БЬЯРНЕ СТРАУСТРУП

Создатель языка C++

Описан C++11 и C++14

Программирование

Принципы и практика с использованием C++

ВТОРОЕ ИЗДАНИЕ

Programming
Principles and Practice Using C++

Second edition

Bjarne Stroustrup

Addison
Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Программирование

Принципы и практика с использованием C++

Второе издание

Бьярне Страуструп

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
С83
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Страуструп, Бьярне.

С83 Программирование: принципы и практика с использованием C++, 2-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 1328 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1949-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Бьярне Страуструп

Программирование: принципы и практика с использованием C++

2-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 26.10.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 107.07. Уч.-изд. л. 66.7.
Тираж 300 экз. Заказ № 6311

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1949-6 (рус.)
ISBN 978-0-321-99278-9 (англ.)

Оглавление

Содержание

Предисловие

К черту мины!
Полный вперед!
— Адмирал Фаррагут

Программирование — это искусство выражать решения задач так, чтобы компьютер мог их осуществить. Основные усилия программиста направлены на то, чтобы найти и уточнить решение, причем довольно часто полное понимание задачи приходит лишь в ходе программирования ее решения.

Эта книга предназначена для тех, кто еще никогда не программировал, но готов учиться этому ценой тяжелого и упорного труда. Она поможет овладеть главными принципами и приобрести практический опыт программирования на языке C++. Моя цель заключается в том, чтобы изложить достаточный объем сведений и научить вас решать простые и полезные задачи по программированию с помощью самых лучших и современных методов. Если вы учитесь на первом курсе университета, то можете изучать эту книгу на протяжении семестра (в предположении, что у вас имеется еще четыре курса средней сложности). Если вы изучаете программирование самостоятельно, то не ждите, что сможете освоить этот курс за меньшее время (ну, при условии, что вы будете работать по 15 часов в неделю, возможно, вам хватит 14 недель).

Три месяца могут показаться долгими, но объем курса довольно велик, и первые простые программы вы сможете написать, проработав над книгой около часа. Кроме того, сложность материала постепенно возрастает: в каждой главе вводятся новые полезные понятия, которые иллюстрируются реальными примерами. Способность выражать свои идеи на языке программирования — т.е. умение объяснять компьютеру, что от него требуется — будет постепенно развиваться у вас по мере изучения. Я никогда не говорю "изучайте теорию в течение месяца, а затем проверьте, сможете ли вы ее применить на практике".

Почему многие хотят научиться программировать? Современная цивилизация основана на компьютерных программах. Не зная, как работают

---

[1] Фаррагут Дэвид Глазго (1801–1870) — первый адмирал США, герой гражданской войны, воевал за северян. В ходе сражения за порт Мобил провел свой корабль через заминированный проход.

эти программы, вы будете вынуждены верить в "волшебство", и многие интересные, выгодные и социально полезные сферы деятельности останутся для вас закрытыми. Когда я говорю о программировании, то подразумеваю весь спектр компьютерных программ — от программ для персональных компьютеров с графическим пользовательским интерфейсом, программ для инженерных вычислений и встроенных систем управления (например, в цифровых видеокамерах, автомобилях и мобильных телефонах) до приложений, предназначенных для манипулирования текстами. Как и математика, программирование — на высоком уровне — представляет собой полезное интеллектуальное упражнение, оттачивающее мыслительные способности. Однако благодаря обратной связи с компьютером программирование носит более конкретный характер, чем многие области математики, а значит, доступно более широкому кругу людей. С помощью программирования можно заработать и изменить мир к лучшему. Кроме того, программирование — довольно увлекательное занятие.

Почему C++? Потому что невозможно научиться программировать, не зная ни одного языка программирования, а язык C++ поддерживает основные концепции и методы, используемые в реальных компьютерных программах. Язык C++ является одним из наиболее широко распространенных языков программирования. Он применяется во многих прикладных сферах. Программы, написанные на языке C++, можно встретить всюду — и на дне океана и на поверхности Марса. Кроме того, существует точный и полный общедоступный международный стандарт языка C++, не защищенный правом собственности. Качественные и/или бесплатные реализации этого языка доступны для любых компьютеров. Большинство концепций программирования, которые вы изучите с помощью языка C++, можно непосредственно использовать и в других языках, таких как C, C#, Fortran и Java. Наконец, я просто люблю этот язык за элегантность и эффективность кода.

Эту книгу нельзя назвать самым простым введением в программирование. Собственно, эту цель я перед собой не ставил. Я просто хотел написать простую и понятную книгу, с помощью которой можно было бы освоить азы практического программирования. Это довольно амбициозная цель, поскольку современное программное обеспечение в большой степени основывается на методах, изобретенных совсем недавно, буквально несколько лет назад.

Надеюсь, что вы — люди ответственные и хотите создавать программы, предназначенные для других пользователей, стараясь обеспечить при этом их высокое качество. Иначе говоря, я предполагаю, что вы желаете достичь определенной степени профессионализма. По этой причине в книге собраны действительно нужные программисту темы, а не просто самые легкие для обучения начинающего. В ней описаны методы, необходимые для правильного программирования, а также приведены связанные с ними

понятия, средства языка и упражнения, которые, я надеюсь, вы обязательно выполните. Люди, интересующиеся лишь игрушечными программами, извлекут из книги намного меньше, чем в нее заложено. С другой стороны, я бы не хотел, чтобы вы растрачивали свое время на материал, который редко находит применение на практике. Если в книге поясняется какая-то идея, значит, я считаю, что она почти наверняка понадобится в реальных приложениях.

Если вы хотите использовать результаты работы других людей, не вникая в детали и не желая добавлять к ним собственный код, то эта книга не для вас. Если это так, подумайте, не следует ли вам выбрать другую книгу и другой язык программирования. Кроме того, задумайтесь над тем, почему вы придерживаетесь такой точки зрения и соответствует ли она вашим потребностям. Люди часто переоценивают сложность программирования, а также его важность. Я бы не хотел вызывать у читателей отвращение к программированию из-за несоответствия между их потребностями и содержанием книги. Существует множество областей мира "информационных технологий", в которых совершенно не требуется знание программирования. Напоминаю, что эта книга предназначена для тех, кто хочет писать или понимать нетривиальные программы.

Благодаря структуре и назначению книги ее могут также использовать люди, уже знакомые с основами языка C++ или владеющие другим языком программирования и желающие изучить C++. Если вы попадаете в одну из этих категорий, то мне сложно предположить, сколько времени вам понадобится на чтение этой книги, но я настоятельно рекомендую обязательно выполнить упражнения. Это поможет решить широко распространенную задачу: адаптировать программы, написанные в старом стиле, с учетом более современных технологий. Если вы овладели языком C++ с помощью одного из традиционных способов обучения, то, возможно, найдете нечто удивительное и полезное в первых шести главах. Здесь рассматриваются темы, которые никак нельзя назвать "C++ времен вашего отца" (если, конечно, ваша фамилия не Страуструп).

Изучение программирования основано на написании программ. Этим программирование похоже на другие виды деятельности, требующие практических занятий. Невозможно научиться плавать, играть на музыкальном инструменте или водить автомобиль, просто прочитав учебник, — необходима практика. Точно так же невозможно научиться программировать, не прочитав и не написав большое количество программ. Основное внимание в книге сосредоточено на программах, которые сопровождаются пояснениями и диаграммами. Вы должны понять идеи, концепции и принципы программирования, а также овладеть языковыми конструкциями, необходимыми для их выражения. Это очень важно, но само по себе не может дать практического опыта программирования. Для того чтобы приобрести такой опыт, следует выполнить упражнения, используя инструментарий

для редактирования, компиляции и выполнения программ. Вы должны делать собственные ошибки и учиться их исправлять. Заменить разработку собственных программ нельзя ничем. Кроме того, это так увлекательно!

С другой стороны, программирование — это нечто большее (гораздо большее!), чем просто следование некоторым правилам и чтению справочника. Эта книга преднамеренно не фокусируется на синтаксисе языка C++. Для того чтобы стать хорошим программистом, необходимо понимать основные идеи, принципы и методы. Только хорошо разработанный код имеет шанс стать частью правильной, надежной и простой в эксплуатации системы. Помимо прочего, основы — это то, что останется даже после того, как современные языки и средства программирования будут усовершенствованы или сойдут с арены.

Что можно сказать о кибернетике, разработке программного обеспечения, информационных технологиях и т.д.? Сводятся ли эти отрасли знаний к программированию? Разумеется, нет! Программирование — всего лишь один из фундаментальных предметов, лежащих в основе всех областей, связанных с использованием компьютеров. Оно занимает свое законное место в курсе компьютерных наук. Я привожу в книге краткий обзор основных понятий и методов, связанных с алгоритмами, структурами данных, пользовательским интерфейсом и программным обеспечением. Однако эта книга не может заменить подробного и сбалансированного учебника по указанным темам.

Программа может быть и прекрасной, и полезной. Надеюсь, эта книга поможет вам понять эту истину. Я старался объяснить, какие программы можно назвать прекрасными, изложить основные принципы их разработки и помочь овладеть практическими навыками по разработке таких программ. Удачи в программировании!

Обращение к студентам

Обучение по этой книге уже прошли более тысячи студентов Техасского университета агрокультуры и машиностроения (Texas A&M University). Из них около 60% уже имели опыт программирования, а остальные 40% до обучения не написали ни одной строчки программы в своей жизни. Большинство из них вполне успешно справились с заданиями, значит, справитесь и вы.

Вы не обязаны читать эту книгу как часть учебного курса. Я предполагаю, что эта книга будет широко использоваться для самообучения. Однако, независимо от того, учитесь ли вы в университете или овладеваете программированием самостоятельно, постарайтесь работать в группе. Программирование часто совершенно неправильно считают занятием одиночек. Большинство людей лучше работают и быстрее обучаются в кол-

лективе, имеющем общую цель. Совместное обучение и обсуждение задач с друзьями ни в коем случае не является обманом преподавателя! Это один из наиболее эффективных, а также наиболее приятных способов совершенствования своего профессионального мастерства. Кроме того, коллективная работа приучает студентов правильно выражать свои идеи, что является одним из наиболее эффективных способов самопроверки и запоминания. Не обязательно искать собственные решения давно известных задач, связанных с языком программирования или особенностями сред для разработки программ. Однако не следует также обманывать себя, не выполняя упражнения (даже если преподаватель их не проверяет). Помните: программирование (помимо всего прочего) — это практический навык, которым следует овладеть. Если вы не пишете программ (т.е. не выполняете упражнения, приведенные в конце каждой главы), то чтение книги сведется к бессмысленному теоретическому занятию.

У большинства студентов — особенно у хороших студентов — иногда возникает вопрос, стоит ли так тяжело работать. Если этот вопрос возник и у вас — сделайте перерыв, перечитайте предисловие и просмотрите главу 1, а также главу 22. В этих главах я попытался объяснить, чем меня восхищает программирование и почему я считаю, что именно программирование способствует улучшению мира. Если вас интересуют мои педагогические принципы, прочитайте главу 0.

Возможно, вы сочтете книгу слишком большой. Частично это объясняется тем, что я либо многократно повторяю объяснения, либо иллюстрирую их дополнительными примерами, вместо того чтобы заставлять читателей удовлетвориться одним-единственным толкованием. Кроме того, вторая часть книги представляет собой справочник и содержит "дополнительный материал", позволяющий читателям углубить свои знания в конкретных областях программирования, например в области программирования встроенных систем, анализа текстов или математических вычислений.

Пожалуйста, сохраняйте терпение. Изучение новых идей и приобретение важных практических навыков требуют времени, но результат стоит затраченных усилий.

Обращение к преподавателям

Нет, это не традиционный курс по компьютерным наукам, принятый в США (Computer Science 101). Эта книга о том, как создать работающее программное обеспечение. Поэтому за ее рамками осталось многое из того, что обычно включается в учебник по компьютерным наукам (сложность алгоритмов по Тьюрингу, конечные автоматы, дискретная математика, грамматики Хомского и т.д.). В книге проигнорирована даже тема,

связанная с аппаратным обеспечением, поскольку я полагаю, что студенты с детства умеют работать с компьютерами. В книге даже не упоминается большинство важных тем из области компьютерных наук. Это книга о программировании (а точнее, о том, как разработать программное обеспечение), поэтому в ней нет того, что обычно включается в традиционные учебники. Поскольку компьютерные науки невозможно изложить в рамках одного курса, я не стал распылять внимание. Если вы будете использовать эту книгу как часть курса по компьютерным наукам, вычислительной технике, электротехнике (большинство наших первых студентов специализировались именно по электротехнике), информатике или какой-либо иной научной дисциплине, то предлагаю выделить ее в виде самодостаточного введения.

Пожалуйста, прочитайте главу 0, чтобы понять мои педагогические принципы, общий подход и т.д. Я прошу вас передать эти идеи своим студентам.

Стандарт ISO

C++ определяется стандартом ISO. Первый стандарт ISO был принят в 1998 году, так что эта версия C++ известна как C++98. Первое издание этой книги я писал во время работы над стандартом C++11, и наибольшим разочарованием была недоступность новых возможностей (таких, как новый стиль инициализации, цикл `for` для диапазона, семантика перемещения, лямбда-выражения) для упрощения изложения в книге принципов и методов. Однако, работая над книгой, я постоянно "держал в уме" C++11, поэтому новое издание книги писалось относительно легко. На момент написания текущим стандартом являлся C++11, а возможности будущего ISO-стандарта, C++14, постепенно находят свой путь в основные реализации C++. Язык, используемый в этой книге, — C++11 с некоторыми возможностями C++14. Например, если ваш компилятор жалуется на код

```
vector<int> v1;
vector<int> v2 {v1}; // Копирование в стиле C++14
```

замените этот текст на

```
vector<int> v1;
vector<int> v2 = v1; // Копирование в стиле C++98
```

Если ваш компилятор не поддерживает C++11, возьмите более новый компилятор. К счастью, современные компиляторы C++ доступны у множества производителей (см. `www.stroustrup.com/compilers.html`). Изучение программирования с использованием старых, не поддерживающих новые возможности версий может оказаться неоправданно тяжелым занятием.

Поддержка

Веб-сайт поддержки книги, находящийся по адресу `www.stroustrup.com/Programming`, содержит множество дополнительных материалов для обучения программированию с помощью данной книги. Со временем этот материал, по-видимому, будет уточняться, но в данный момент читатели найдут там следующие материалы.

- Слайды лекций по этой книге
- Справочник преподавателя
- Заголовочные файлы и реализации библиотек, использованных в книге
- Тексты программ, используемых в книге в качестве примеров
- Решения избранных упражнений
- Потенциально полезные ссылки
- Список найденных ошибок

Любые замечания и предложения по улучшению книги будут приняты с благодарностью.

Благодарности

Я особенно благодарен моему коллеге Лоуренсу "Питу" Петерсену (Lawrence "Pete" Petersen) за то, что он вдохновил меня взяться за обучение новичков и за полезные практические советы по преподаванию. Без его помощи первый вариант этого курса оказался бы неудачным. Мы работали вместе над первым вариантом курса, которому посвящена эта книга, а затем совершенствовали ее, учитывая полученный опыт. Местоимение "мы", использованное в книге, означает "Пит и я".

Выражаю признательность студентам, ассистентам и преподавателям Техасского университета агрокультуры и машиностроения (курсы ENGR 112, ENGR 113 и CSCE 121), которые вольно или невольно помогли написать эту книгу, а также Уолтеру Догерити (Walter Daugherity), Хьюнюнгу Ли (Hyunyoung Lee), Терезе Лейк (Teresa Leyk), Ронни Уарду (Ronnie Ward) и Дженнифер Велч (Jennifer Welch), прослушавшим этот курс. Кроме того, я благодарен Дэмиану Дечеву (Damian Dechev), Трейси Хэммонд (Tracy Hammond), Арне Толструпу Мэдсену (Arne Tolstrup Madsen), Габриэлю Дос Рейосу (Gabriel Dos Reis), Николасу Страуструпу (Nicholas Stroustrup), Дж. К. ван Винкелю (J. C. van Winkel), Грэгу Версундеру (Greg Versoonder), Ронни Уарду (Ronnie Ward) и Леору Зольману (Leor Zolman) за конструктивные замечания к рукописи книги. Большое спасибо Могенсу Хансену (Mogens Hansen) за объяснение принципов работы программного

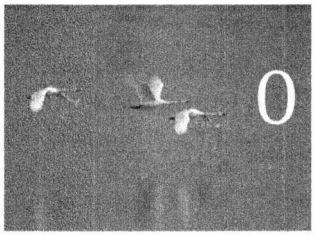
обеспечения по управлению двигателем, а также Элу Ахо (Al Aho), Стивену Эдвардсу (Stephen Edwards), Брайану Кернигану (Brian Kernighan) и Дэйзи Нгуен (Daisy Nguyen) за то, что помогли мне спрятаться от всего, что могло отвлечь от работы над книгой летом.

Благодарю Арта Вершульца (Art Werschulz) за множество конструктивных замечаний, основанных на использовании им первого издания данной книги в качестве учебника для курса в Университете Фордхэма в Нью-Йорке, и Ника Маклареna (Nick Maclaren) за детальные примечания к упражнениям, которые он сделал при использовании первого издания книги в Кэмбриджском университете. Его студенты резко отличались по уровню знаний и профессиональным требованиям от первокурсников Техасского университета агрокультуры и машиностроения.

Благодарю рецензентов, которых издательство Addison-Wesley подыскало для меня. Их замечания, основанные на их опыте преподавания C++ или информатики в колледже, оказались неоценимыми. Это Ричард Энбоди (Richard Enbody), Дэвид Густафсон (David Gustafson), Рон Мак-Карти (Ron McCarty) и К. Нараянасвами (K. Narayanaswamy). Я признателен также моему редактору Питеру Гордону (Peter Gordon) за многочисленные комментарии и (не в последнюю очередь) за его терпение. Большое спасибо техническому персоналу издательства Addison-Wesley. Они много сделали для повышения качества книги. Это корректор Линда Бигли (Linda Begley), верстальщик Ким Арни (Kim Arney Keane), художник Роб Мохар (Rob Mauhar), технический редактор Джулия Нахил (Julie Nahil) и литературный редактор Барбара Вуд (BarbaraWood).

Я признателен переводчикам первого издания, которые помогли выявить массу неточностей и неясностей. В первую очередь, хочу отметить переводчиков книги на французский язык Луи Джоли (Lonc Joly) и Мишеля Мишо (Michel Michaud), благодаря которым стало возможным значительно повысить качество второго издания.

Я хотел бы также поблагодарить Брайана Кернигана (Brian Kernighan) и Дуга Мак-Илроя (Doug McIlroy) за очень высокие стандарты качества, установленные ими для программирования, а также Денниса Ритчи (Dennis Ritchie) и Кристена Нийгарда (Kristen Nygaard) за ценные уроки по практической разработке языков программирования.



0

Обращение к читателям

Если карта не совпадает с местностью, доверяй местности.
Швейцарская армейская поговорка

Эта глава содержит разнообразную информацию; ее цель — дать представление о том, что можно ожидать от остальной части книги. Пожалуйста, пролистайте ее и прочтите то, что покажется вам интересным. Для преподавателей полезной будет большая часть книги. Если же вы изучаете книгу без помощи хорошего преподавателя, то не пытайтесь понять все, что написано в этой главе; просто взгляните на раздел "Структура книги" и первую часть раздела "Принципы обучения и изучения". Возможно, вы захотите вернуться и перечитать эту главу, когда научитесь писать и выполнять небольшие программы.

0.1. Структура книги

Книга состоит из четырех частей и нескольких приложений.

- *В части I, "Основы"*, описаны фундаментальные концепции и методы программирования на примере языка C++ и библиотек, необходимых для начала разработки программ. К этим концепциям относятся система типов, арифметические операции, управляющие конструкции, обработка ошибок, а также разработка, реализация и использование функций и пользовательских типов.
- *В части II, "Ввод и вывод"*, описаны способы ввода числовых и текстовых данных с клавиатуры и из файлов, а также вывода результатов на экран и в файлы. Кроме того, в ней показано, как вывести числа, текст и геометрические фигуры в виде графической информации, а также как ввести данные в программу с помощью графического пользовательского интерфейса (GUI).
- *Часть III, "Данные и алгоритмы"*, посвящена контейнерам и алгоритмам из стандартной библиотеки C++ (standard template library — STL). В ней продемонстрированы реализация и использование контейнеров (таких, как `vector`, `list` и `map`) с помощью указателей, массивов, динамической памяти, исключений и шаблонов. Кроме того, описаны разработка и использование алгоритмов из стандартной библиотеки (таких, как `sort`, `find` и `inner_product`).
- *Часть IV, "Дополнительные темы"*, посвящена идеям и истории программирования на примерах матричных вычислений, обработки текста, тестирования и встроенных систем, а также краткому описанию языка C.
- *Приложения* содержат полезную информацию, которая была пропущена в тексте по дидактическим причинам. В частности, приводится краткий обзор языка C++ и возможностей стандартной библиотеки,

а также продемонстрированы принципы работы с интегрированными средами разработки (integrated development environment — IDE) и библиотекой графического пользовательского интерфейса (graphical user interface — GUI).

К сожалению, программирование нельзя так просто разделить на четыре четкие области. По этой причине предложенная классификация является довольно грубой, хотя мы считаем ее полезной (иначе не стали бы ее предлагать). Например, операции ввода в книге используются намного раньше детального описания стандартных потоков ввода-вывода в языке C++. Как только для описания какой-то идеи нам требуется упомянуть несколько тем, мы предпочитаем изложить минимум информации, а не отсылать читателя к подробному изложению темы в другом месте. Строгая классификация больше нужна для справочников, чем для учебников.

Порядок изложения определяется методами программирования, а не языковыми конструкциями (см. раздел 0.2). Обзор свойств языка содержится в приложении А.

Для облегчения работы читателей, впервые читающих книгу и еще не знающих, какая информация является действительно важной, мы используем три вида пиктограмм, которые должны привлечь внимание.

Пиктограмма "Концепции и методы" (как в абзаце выше).

Пиктограмма "Совет".

Пиктограмма "Предупреждение".

0.1.1. Общие принципы

В книге я обращаюсь к вам непосредственно. Это проще и понятнее, чем принятое в научных работах косвенное обращение в третьем лице. Под местоимением "вы" я подразумеваю вас, читатель, а под местоимением "мы" — себя и преподавателей или нас с вами, работающих вместе над решением задачи, как будто мы с вами находимся в одной комнате.

Эту книгу следует читать главу за главой, от начала до конца. Довольно часто у вас будет возникать желание вернуться к какому-либо разделу и перечитать его во второй или в третий раз. На самом деле это единственное разумное поведение, так как со временем некоторые детали стираются в памяти. В таких случаях вы обязательно рано или поздно постараетесь их освежить. Однако, несмотря на предметный указатель и перекрестные ссылки, это не та книга, которую можно

открыть на любой странице и начинать читать, рассчитывая на успех. Каждый раздел и каждая глава требуют твердого знания материала, изложенного в предыдущих разделах и главах.

Каждая глава является вполне самодостаточной единицей, т.е. ее в принципе можно прочесть в один присест (что, конечно, не всегда возможно из-за напряженного расписания занятий). Это один из основных критериев разделения текста на главы. Кроме того, каждая глава содержит упражнения и задачи, а также посвящена конкретной концепции, идее или методу. Некоторые главы получились слишком длинными, поэтому понимать выражение "за один присест" следует скорее в смысле логического единства главы, а не буквально. В частности, поразмышляв над контрольными вопросами, разобрав примеры и выполнив несколько упражнений, вы почти наверняка поймете, что вам следует еще раз перечитать какие-то разделы, и на это может уйти несколько дней. Мы объединили главы в части, посвященные основным темам, например вводу-выводу. Эти части удобны для проведения контрольных опросов.

Об учебниках часто говорят "Он ответил на все мои вопросы сразу, как только я о них подумал!" Это типично для простых вопросов, и первые читатели рукописи этой книги заметили это. Однако этот принцип не может быть всеобщим. Мы поднимаем вопросы, которые новичку вообще не могут прийти в голову. Наша цель — поставить вопросы, необходимые для написания качественных программ, предназначенных для других людей, и ответить на них. Научить задавать правильные (часто сложные) вопросы необходимо для того, чтобы студент стал думать, как программист. Задавать простые и очевидные вопросы очень удобно, но это не поможет стать программистом.

Мы стараемся уважать ваш интеллект и учитываем затраты вашего времени. В изложении мы ценим профессионализм, а не красивости, поэтому некоторые вещи недоговариваем, а не разжевываем. Мы стараемся не преувеличивать важность методов программирования или языковых конструкций, но не следует недооценивать и такие простые утверждения, как, например, "Это свойство часто оказывается полезным". Если мы подчеркиваем, что некий материал является важным, значит, если вы не освоите его, то рано или поздно вы потеряете из-за этого много дней. Мы шутим намного меньше, чем хотели бы: опыт показывает, что у людей совершенно разное чувство юмора и попытки шутить могут лишь запутать изложение.

Мы не претендуем на идеальность наших идей или инструментов. Ни один инструмент, ни одна библиотека и ни один метод не может решить все проблемы, возникающие у программиста. В лучшем случае они помогут разработать и реализовать ваше решение. Мы очень старались избегать "святой лжи", т.е. отказались от упрощенных объяснений, которые легко и просто понять, но которые на самом деле

неверны в контексте реальных языков и задач. С другой стороны, эта книга — не справочник; более точное и полное описание языка C++ изложено в книге Страуструпа *The C++ Programming Language, Fourth Edition* (Addison-Wesley, 2013) и в стандарте ISO C++.

0.1.2. Упражнения, задачи и т.п.

Программирование — это не просто интеллектуальная деятельность, поэтому для овладения этим искусством необходимо писать программы. Мы предлагаем два уровня практического программирования.

- *Задания.* Простые задачи, предназначенные для отработки практических, почти механических навыков. Задания обычно подразумевают последовательность модификаций простой программы. Вы должны выполнить каждое задание. Задания не требуют глубокого понимания, ума или инициативы. Мы рассматриваем их как очень важную часть книги. Если вы не выполните задания, то не поймете материал, изложенный в книге.
- *Упражнения.* Одни упражнения тривиальны, другие очень сложны, но большинство из них предназначено для того, чтобы разбудить ваши инициативу и воображение. Если вы серьезный человек, то выполните хотя бы несколько упражнений. Попробуйте это сделать хотя бы для того, чтобы понять, насколько это трудно для вас. Затем выполните еще несколько упражнений. Так постепенно вы справитесь с большинством из них. Эти упражнения требуют не столько выдающихся умственных способностей, сколько изобретательности. Однако мы надеемся, что они достаточно трудны, чтобы стимулировать ваше самолюбие и занять все ваше свободное время. Мы не рассчитываем, что вы решите все задачи, но советуем попытаться.

Кроме того, рекомендуем каждому студенту принять участие в разработке небольшого проекта (или крупного, если будет время). Эти проекты предназначены для того, чтобы написать законченную полезную программу. В идеале проекты должны создаваться небольшими группами разработчиков (например, тремя программистами), работающих вместе около месяца и осваивающих главы части III, "Данные и алгоритмы". Большинство студентов получают удовольствие именно от работы над проектом, который связывает людей друг с другом. Одни люди предпочтут отложить книгу в сторону и решать задачи, еще не дойдя до конца главы; другие захотят дочитать до конца и лишь затем приступить к программированию. Для того чтобы поддержать студентов, желающих программировать сразу, мы предлагаем простые практические задания, которые озаглавлены **Попробуйте**. Эти задания являются естественными составными частями

книги. По существу, эти задания относятся к упражнениям, но сфокусированы на узкой теме, которая изложена перед их формулировкой. Если вы пропустите это задание — например, потому, что поблизости нет компьютера или вас слишком увлекло чтение книги, — вернитесь к нему, когда начнете выполнять упражнения; задания **Попробуйте** либо являются частью упражнений, либо дополняют их.

В конце каждой главы вы найдете контрольные вопросы. Они предназначены для закрепления основных идей, объясняемых в главе. Эти вопросы можно рассматривать как дополнения к задачам. В то время как задачи посвящены практическим аспектам программирования, контрольные вопросы позволяют сформулировать идеи и концепции. Этим они напоминают интервью.

Раздел "Термины" в конце каждой главы представляет собой часть словаря по программированию и языку C++. Если хотите понимать, что говорят о программировании, и свободно выражать собственные идеи, запомните значения этих слов.

Повторение — мать учения. Идеальный студент должен повторить каждую важную идею как минимум дважды, а затем закрепить ее с помощью упражнений.

0.1.3. Что дальше

Станете ли вы профессиональным программистом или экспертом по языку C++, прочитав эту книгу? Конечно, нет! Настоящее программирование — это тонкое, глубокое и очень сложное искусство, требующее знаний и технических навыков. Рассчитывать на то, что за четыре месяца вы станете экспертом по программированию, можно с таким же успехом, как и на то, что за полгода или даже год вы полностью изучите биологию, математику или иностранный язык (например, китайский, английский или датский) или научитесь играть на виолончели. Если подходить к изучению книги серьезно, то можно ожидать, что вы сможете писать простые полезные программы, читать более сложные программы и получите хорошие теоретическую и практическую основы для дальнейшей работы.

Прослушав этот курс, лучше всего поработать над реальным проектом. Еще лучше параллельно с работой над реальным проектом приступить к чтению какой-нибудь книги профессионального уровня (например, Bjarne Stroustrup, *The C++ Programming Language, Special Edition* (Addison-Wesley, 2000)), более специализированной книги, связанной с вашим проектом (например, документации по библиотеке Qt для разработки графического пользовательского интерфейса GUI или справочника по библиотеке ACE для параллельного программирования, или учебника, посвященного конкретному аспекту языка C++, например Кёниг Э., Му Б. *Эффективное*

программирование на C++. — М.: Издательский дом "Вильямс", 2002. — 384 с.; Саттер Г. *Решение сложных задач на C++*. — М.: Издательство Вильямс, 2002. — 400 с.; Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — Спб.: Питер, 2004. — 366 с.). Полный список рекомендуемых книг приведен в разделе 0.6 и в разделе "Библиография" в конце книги.

В конечном итоге вам придется приступить к изучению некоторого другого языка программирования. Невозможно стать профессионалом в области программного обеспечения (даже если программирование не является вашей основной специальностью), зная только один язык программирования.

0.2. Принципы обучения и изучения

Чему мы хотим вас научить и как собираемся организовать процесс обучения? Мы попытались изложить минимальный объем концепций, методов и инструментов, необходимых для эффективного программирования. Их список приведен ниже.

- Организация программ
- Отладка и тестирование
- Проектирование классов
- Вычисления
- Проектирование функций и алгоритмов
- Графика (только двумерная)
- Графические пользовательские интерфейсы
- Обработка текста
- Регулярные выражения
- Файлы и потоки ввода-вывода (I/O)
- Управление памятью
- Научные/числовые/инженерные вычисления
- Принципы проектирования и программирования
- Стандартная библиотека языка C++
- Стратегии разработки программного обеспечения
- Приемы программирования на языке C

Эти темы охватывают процедурное программирование (его типичным представителем является язык C), а также абстракцию данных, объектно-ориентированное и обобщенное программирование. Основным предметом

книги является именно *программирование*, т.е. идеи, методы и средства выражения этих идей с помощью программ. Нашим основным инструментом является язык C++, поэтому мы довольно подробно описываем его многочисленные возможности. Однако следует помнить, что язык C++ — это просто инструмент, а не основной предмет изучения этой книги. Иначе говоря, книга посвящена программированию с помощью языка C++, а не языку C++ с небольшим количеством теории.

Каждая тема, которую мы излагаем, преследует цель описать метод, концепцию или принцип, а также практический язык программирования или свойство библиотеки. Например, для иллюстрации классов и концепции наследования мы используем систему двумерной графики. Это позволит сэкономить место (и ваше время), а также продемонстрировать, что программирование не сводится к простому связыванию фрагментов кода друг с другом, чтобы как можно быстрее получить результат. Основным источником таких "примеров двойного назначения" является стандартная библиотека языка C++. Некоторые из этих примеров имеют даже тройное назначение. Например, мы рассматриваем класс `vector` из стандартной библиотеки, используем его для иллюстрации полезных методов проектирования и демонстрируем многочисленные приемы программирования, позволяющие его реализовать. Одна из целей — показать, как реализованы основные возможности библиотеки и как они отображаются на аппаратное обеспечение. Мы настаиваем на том, что профессионал должен понимать устройство инструментов, с которыми он работает, а не считать их волшебной палочкой.

Одни темы покажутся некоторым программистам более интересными, чем другие. Однако мы советуем не предвосхищать свои потребности (как вы можете знать, что вам понадобится в будущем?) и хотя бы просмотреть каждую главу. Если вы используете книгу как учебник, а не самоучитель, то ваш преподаватель сам определит выбор глав.

Наш подход можно назвать глубинным, конкретным или концептуальным. Вначале мы быстро (ну, хорошо — относительно быстро, в главах 1–11) описываем набор навыков, необходимых для написания небольших практических программ. При этом мы описываем много инструментов и приемов, не вдаваясь в детали. Мы акцентируем внимание на простых конкретных программах, поскольку конкретное усваивается быстрее, чем абстрактное. Большинство людей используют именно такой способ обучения. Не рассчитывайте на то, что уже на ранних стадиях обучения вы поймете все до малейших деталей. В частности, пытаясь сделать нечто, отличающееся от того, что только что работало, вы обнаружите "загадочные" явления. Впрочем, попытайтесь! И пожалуйста, не забывайте выполнять упражнения и решать задачи, которые мы предлагаем. Помните, что на первых порах у вас просто еще нет достаточно знаний

и опыта, чтобы понять, что является простым, а что сложным; выявляйте недостатки и учитесь на них.

Первый этап мы пройдем в быстром темпе. Мы хотим как можно быстрее достичь пункта, после которого вы сможете писать собственные интересные программы. Кое-кто говорит "Мы должны двигаться медленно и осторожно; прежде чем научиться бегать, мы должны научиться ходить!" Но где вы видели ребенка, который учился бы именно ходить, а не бегать? На самом дели дети бегают, пока не научатся контролировать свою скорость. Точно так же мы сначала быстренько, иногда ненадолго останавливаясь, научимся программировать, а уж потом притормозим, чтобы глубже разобраться и понять, как все это работает. Мы должны научиться бегать раньше, чем ходить!

Ни в коем случае не следует зацикливаться на попытках досконально изучить какие-то детали языка или метода. Разумеется, вы можете заучить все встроенные типы данных в языке C++ и все правила их использования. Конечно, после этого вы можете чувствовать себя знатоком. Однако это не сделает вас программистом. Пренебрежение деталями может вызвать у вас ощущение недостатка знаний, но это быстрейший способ, позволяющий научиться писать хорошие программы. Обратите внимание на то, что именно наш подход, по существу, используется при обучении детей иностранным языкам. Если вы зайдете в тупик, советуем искать помощи у преподавателей, друзей, коллег и т.п. Не забывайте, что в первых главах нет ничего принципиально сложного. Однако многое будет незнакомым и поэтому может показаться сложным.

Позднее мы углубим ваши первоначальные навыки, чтобы расширить базу ваших знаний и опыта. Для иллюстрации концепций программирования мы используем упражнения и задачи.

Основной упор в книге делается на идеи и причины. Людям нужны идеи, чтобы решать практические задачи, т.е. находить правильные и принципиальные решения. Необходимо понимать подоплеку этих идей, т.е. знать, почему именно этими, а не другими принципами следует руководствоваться, а также чем это может помочь программистам и пользователям программ. Никого не может удовлетворить объяснение "потому что потому". Кроме того, понимание идей и причин позволит вам обобщить их в новых ситуациях и комбинировать принципы и средства для решения новых задач. Знание причин является важной частью программистских навыков. И наоборот, формальное знание многочисленных плохо понятых правил и конструкций языка программирования является источником многих ошибок и приводит к колоссальной потере времени. Мы ценим ваше время и не хотим его тратить понапрасну.

Многие технические детали языка C++ изложены в приложениях и справочниках, где их можно при необходимости найти. Мы считаем, что вы способны самостоятельно найти заинтересовавшую вас информацию.

Используйте для этого предметный указатель и содержание. Не забывайте также об Интернете. Однако помните, что не каждой веб-странице следует слепо доверять. Многие веб-сайты, выглядящие авторитетными источниками знаний, созданы новичками или просто пытаются что-то кому-то продать. Некоторые веб-сайты просто устарели. Мы собрали коллекцию полезных ссылок и фактов на нашем веб-сайте `www.stroustrup.com/Programming`.

Пожалуйста, не придирайтесь к "реалистичности" примеров. Идеальный пример — это максимально короткая и простая программа, ярко иллюстрирующая свойство языка, концепцию или прием. Большинство реальных примеров являются намного более запутанными, чем наши, и не содержат необходимых комбинаций идей, которые мы хотели бы продемонстрировать. Успешные коммерческие программы, содержащие сотни тысяч строк, основаны на технических приемах, которые можно проиллюстрировать дюжиной программ длиной по 50 строк. Самый быстрый путь к пониманию реальной программы ведет через хорошее знание теоретических основ.

С другой стороны, мы не используем для иллюстрации своих идей красивые примеры с симпатичными зверюшками. Наша цель — научить вас писать реальные программы, которые будут использоваться реальными людьми. По этой причине каждый пример, не относящийся к технической стороне языка программирования, взят из реальной жизни. Мы стараемся обращаться к читателям, как профессионалы к (будущим) профессионалам.

0.2.1. Порядок изложения

Существует множество способов обучения программированию. Совершенно очевидно, что мы не придерживаемся популярного принципа "способ, которым я научился программировать, является наилучшим способом обучения". Для облегчения процесса обучения мы сначала излагаем темы, которые считались сложными еще несколько лет назад. Мы стремились к тому, чтобы излагаемые темы вытекали из поставленных задач и плавно переходили одна в другую по мере повышения уровня ваших знаний. По этой причине книга больше похожа на повествование, а не на словарь или справочник.

Невозможно одновременно изучить все принципы, методы и свойства языка, необходимые для создания программ. Следовательно, необходимо выбрать подмножество принципов, методов и свойств, с которых следует начинать обучение. В целом любой учебник должен вести студентов через набор таких подмножеств. Мы понимаем свою ответственность за выбор этих тем. Поскольку все темы охватить невозможно, на каждом этапе обучения мы должны выбирать; тем не менее то, что останется за рамками нашего внимания, не менее важно, чем то, что будет включено в курс.

Для контраста, возможно, будет полезно привести список подходов, которые мы отвергли.

- "*Сначала следует изучить язык C*". Этот подход к изучению языка C++ приводит к ненужной потере времени и приучает студентов к неправильному стилю программирования, вынуждая их решать задачи, имея в своем распоряжении ограниченный набор средств, конструкций и библиотек. Язык C++ предусматривает более строгую проверку типов, чем язык C, а стандартная библиотека лучше соответствует потребностям новичков и позволяет применять исключения для обработки ошибок.
- "*Снизу вверх*". Этот подход отвлекает от изучения хороших и эффективных стилей программирования. Вынуждая студентов решать проблемы, ограничиваясь совершенно недостаточными языковыми конструкциями и библиотеками, он приучает их к плохим и слишком затратным способам программирования.
- "*Если вы что-то описываете, то должны делать это исчерпывающим образом*". Этот подход приводит к только что отвергнутому подходу "снизу вверх" (заставляя студентов все глубже и глубже погружаться в технические детали). В результате новички тонут в море технических подробностей, на изучение которых им потребуются годы. Если вы умеете программировать, то техническую информацию найдете в справочниках. Документация хороша сама по себе, но совершенно не подходит для первоначального изучения концепций.
- "*Сверху вниз*". Этот подход, предусматривающий переход от формулировки принципа к его техническим подробностям, отвлекает читателей от практических аспектов программирования и заставляет концентрироваться на высокоуровневых концепциях еще до того, как они поймут, зачем они нужны. Например, никто просто не в состоянии правильно оценить принципы разработки программного обеспечения, пока не поймет, как легко делать ошибки и как трудно их исправлять.
- "*Сначала следует изучать абстракции*". Фокусируясь лишь на основных принципах и защищая студентов от ужасной реальности, этот подход может вызвать у них пренебрежение реальными ограничениями, связанными с практическими задачами, языками программирования, инструментами и аппаратным обеспечением. Довольно часто этот подход поддерживается искусственными "учебными языками", которые в дальнейшем нигде не используются и (вольно или невольно) дезинформируют студентов о проблемах, связанных с аппаратным обеспечением и компьютерными системами.

- "*Сначала следует изучить принципы разработки программного обеспечения*". Этот подход, как и подход "Сначала следует изучить абстракции", порождает те же проблемы, что и подход "сверху вниз": без конкретных примеров и практического опыта вы просто не сможете оценить важность абстракций и правильного выбора методов разработки программного обеспечения.
- "*С первого дня следует изучать объектно-ориентированное программирование*". Объектно-ориентированное программирование — один из лучших методов организации программ, но это не единственный эффективный способ программирования. В частности, мы считаем, что сначала необходимо изучить типы данных и алгоритмы и лишь потом переходить к разработке классов и их иерархий. Мы с первого дня применяем пользовательские типы (то, что некоторые люди называют объектами), но не углубляемся в устройство класса до главы 6 и не демонстрируем иерархию классов до главы 12.
- "*Просто верьте в магию*". Этот подход основан на демонстрации мощных инструментов и методов без углубления в технические подробности. Он заставляет студентов угадывать — как правило, неправильно, — что же происходит в программе, с какими затратами это связано и где это можно применить. В результате студент выбирает лишь знакомые ему шаблоны, что мешает дальнейшему обучению.

Естественно, мы вовсе не имеем в виду, что все эти подходы совершенно бесполезны. Фактически мы даже используем некоторые из них при изложении некоторых тем. Однако в целом мы отвергаем их как общий способ обучения программированию, полезному для реального мира, и предлагаем альтернативу: конкретное и глубокое обучение с акцентом на концепции и методы.

0.2.2. Программирование и языки программирования

В первую очередь, мы учим программированию, а выбранный язык программирования рассматриваем лишь как вспомогательное средство. Выбранный нами способ обучения может опираться на любой универсальный язык программирования. Наша главная цель — помочь вам понять основные концепции, принципы и методы. Однако эту цель нельзя рассматривать изолированно. Например, языки программирования отличаются друг от друга деталями синтаксиса, возможностями непосредственного выражения разных идей, а также средствами технической поддержки. Тем не менее многие фундаментальные методы разработки безошибочных программ, например написание логически простого кода (главы 5 и 6), выявления инвариантов (раздел 9.4.3) и отделения интер-

фейса от реализации (разделы 9.7, 14.1 и 14.2), во всех языках программирования практически одинаковы.

Методы программирования и проектирования следует изучать на основе определенного языка программирования. Проектирование, программирование и отладка не относятся к навыкам, которыми можно овладеть абстрактно. Вы должны писать программы на каком-то языке и приобретать практический опыт. Это значит, что вы должны изучить основы какого-то языка программирования. Мы говорим "основы", так как времена, когда все основные промышленные языки программирования можно было изучить за несколько недель, ушли в прошлое. Для обучения мы выбрали подмножество языка C++, которое лучше всего подходит для разработки хороших программ. Кроме того, мы описываем свойства языка C++, которые невозможно не упомянуть, поскольку они либо необходимы для логической полноты, либо широко используются в сообществе программистов.

0.2.3. Переносимость

Как правило, программы на языке C++ выполняются на разнообразных компьютерах. Основные приложения на языке C++ выполняются на компьютерах, о которых мы даже не слышали! По этой причине мы считаем переносимость программ и возможность их выполнения на компьютерах с разной архитектурой и операционными системами одним из самых важных свойств. Практически каждый пример в этой книге не только соответствует стандарту ISO Standard C++, но и обладает переносимостью. Если иное не указано явно, представленные в книге программы могут быть выполнены с помощью любого компилятора языка C++ и протестированы на разных компьютерах и под управлением разных операционных систем.

Процесс компиляции, редактирования связей и выполнения программ на языке C++ зависит от операционной системы. Было бы слишком неудобно постоянно описывать детали устройства этих систем и компиляторов каждый раз при ссылке на выполнение программы. Наиболее важная информация, необходимая для использования интегрированной среды разработки программ Visual Studio и компилятора Microsoft C++ под управлением операционной системы Windows, приведена в приложении B.

Если вы испытываете трудности при работе с популярными, но слишком сложными интегрированными средами разработки программ, предлагаем использовать командную строку; это удивительно просто. Например, для того чтобы скомпилировать, отредактировать связи и выполнить простую программу, состоящую из двух исходных файлов, `my_file1.cpp` и `my_file2.cpp`, с помощью компилятора GNU C++ под управлением операционной системы Unix или Linux, выполните две команды:

```
g++ -o my_program my_file1.cpp my_file2.cpp
my_program
```

Да, этого действительно достаточно.

0.3. Программирование и компьютерные науки

Можно ли свести компьютерные науки только к программированию? Разумеется, нет! Единственная причина, по которой мы поставили этот вопрос, заключается в том, что люди часто заблуждаются по этому поводу. Мы затрагиваем множество тем, связанных с компьютерными науками, например алгоритмы и структуры данных, но наша цель — научить программировать, т.е. разрабатывать и выполнять программы. Это одновременно и шире, и уже, чем общепринятая точка зрения на компьютерные науки.

- *Шире*, так как программирование связано с множеством технических знаний, которые, как правило, не относятся ни к какой научной дисциплине.
- *Уже*, т.е. мы не стремились к систематическому изложению основ компьютерных наук.

Цель этой книги — частично охватить курс компьютерных наук (если вы собираетесь стать специалистом в этой области), изложить введение в методы разработки и эксплуатации программного обеспечения (если вы планируете стать программистом или разработчиком программного обеспечения) и, вообще говоря, заложить основы более общего курса.

Тем не менее, несмотря на то, что изложение опирается на компьютерные науки и их основные принципы, следует подчеркнуть, что мы рассматриваем программирование как совокупность практических навыков, основанных на теории и опыте, а не как науку.

0.4. Творческое начало и решение задач

Основная цель книги — помочь вам выражать свои идеи в программах, а не научить придумывать эти идеи. Кроме того, мы приводим множество примеров решения задач, как правило, с помощью анализа, за которым следует последовательное уточнение решения. Мы считаем, что программирование само по себе является формой решения задач: только полностью поняв задачу и ее решение, можно написать правильную программу; и только через создание и тестирование программ можно прийти к полному пониманию задачи. Таким образом, программирование является неотъемлемой частью процесса познания. Однако мы стараемся продемонстрировать это на примерах, а не путем "проповеди" или подробного описания процесса решения задач.

0.5. Обратная связь

Идеальных учебников не существует; потребности разных людей очень различаются. Однако мы старались сделать эту книгу и сопровождающие ее материалы как можно лучшими. Для этого нам необходима обратная связь; хороший учебник невозможно написать в изоляции от читателей. Пожалуйста, сообщите нам об ошибках, опечатках, неясных местах, пропущенных объяснениях и т.п. Мы также будем благодарны за постановку более интересных задач, за формулировку более ярких примеров, за предложения тем, которые следует удалить или добавить, и т.д. Конструктивные комментарии помогут будущим читателям. Все найденные ошибки будут опубликованы на веб-сайте `www.stroustrup.com/Programming`.

0.6. Библиографические ссылки

Кроме публикаций, упомянутых в главе, ниже приведен список работ, которые могут оказаться полезными.

Becker, Pete, ed. *The C++ Standard*. ISO/IEC 14882:2011.

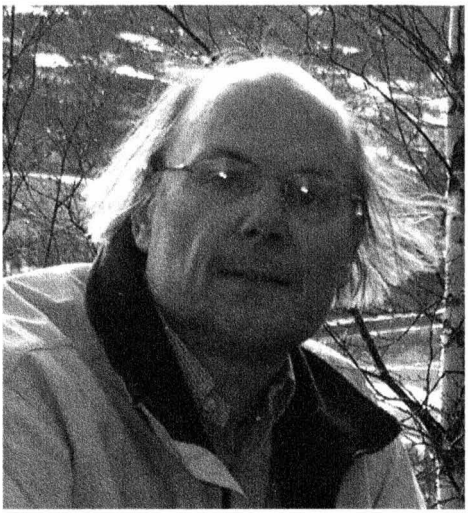
1. Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall, 2008. ISBN 0132354160.
2. Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
3. Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
4. Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2001. ISBN 0201604647.
5. Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. ISBN 0201795256.
6. Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
7. Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.
8. Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
9. Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 0321958314.
10. Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 1999. ISBN 0201615622.

Более полный список библиографических ссылок приведен в конце книги.

0.7. Биографии

Вы можете вполне резонно спросить "Кто эти люди, которые собираются учить нас программировать?" Для того чтобы вы поняли это, мы приводим некоторую биографическую информацию. Я, Бьярне Страуструп, написал эту книгу и вместе с Лоуренсом "Питом" Петерсеном на ее основе разработал университетский вводный курс программирования.

Бьярне Страуструп



Я разработал и впервые реализовал язык программирования C++. В течение последних сорока лет я использовал этот и многие другие языки программирования для решения многочисленных задач. Я люблю элегантные и эффективные программы, предназначенные для сложных приложений, таких как управление роботами, графические системы, игры, анализ текста и компьютерные сети. Я учил проектированию, программированию и языку C++ людей с разными способностями и интересами. Кроме того, я являюсь членом-основателем Комитета ISO по стандартизации языка C++, в котором возглавляю рабочую группу по эволюции языка.

Это моя первая книга, представляющая собой вводный курс. Мои другие книги, такие как "*Язык программирования C++*" и "*Дизайн и эволюция C++*", предназначены для опытных программистов.

Я родился в семье рабочих в Архусе, Дания, и получил магистерскую степень по математике и компьютерным наукам в местном университете. Докторскую степень по компьютерным наукам я получил в Кембридже, Англия. Около двадцати пяти лет я работал в компании AT&T, сначала в знаменитом Исследовательском компьютерном центре лабораторий Белла (Computer Science Research Center of Bell Labs; именно там были изобретены операционная система Unix, языки C и C++, а также многое другое), а позднее — в подразделении AT&T Labs–Research.

Я являюсь членом Национальной технической академии США (U.S. National Academy of Engineering), Ассоциации по вычислительной технике (Association for Computing Machinery — ACM), Института инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers — IEEE). Я был первым специалистом по компьютерным наукам,

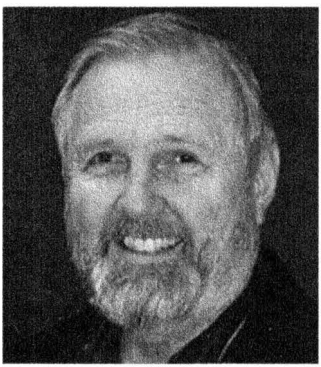
получившим в 2005 году премию Уильяма Проктера за научные достижения (William Procter Prize for Scientific Achievement), которую присуждает научное общество Sigma Xi.

В 2010 году я получил старейшую и почетнейшую награду моей альма-матер, присуждаемую за вклад в науку выпускникам университета, — *Rigmor og Carl Holst-Knudsens Videnskapspris*. В 2013 году мне присвоено звание почетного доктора информатики Университета информационных технологий, механики и оптики в Санкт-Петербурге, Россия.

Работа занимает не все мое время. Я женат, у меня двое детей. Один из них стал врачом, а другой учится в аспирантуре. Я читаю много книг (исторические повести, научную фантастику, детективы и книги о текущих событиях) и люблю музыку (включая классику, рок, блюз и кантри). Застолья с друзьями составляют существенную часть моей жизни. Я люблю посещать интересные места и интересных людей по всему миру. Для того чтобы застолья проходили без последствий, я бегаю по утрам.

Дополнительную информацию обо мне можно найти на веб-сайте по адресу `www.stroustrup.com`. В частности, там вы узнаете, как правильно произносится мое имя[2].

Лоуренс "Пит" Петерсен



В конце 2006 года Пит представлялся так: "Я — учитель. Почти двадцать лет я преподаю языки программирования в Техасском университете агрокультуры и машиностроения (Texas A&M). Студенты пять раз выдвигали меня на присуждение премий за успехи в преподавании (Teaching Excellence Awards), и в 1996 году я получил премию за достижения в преподавании (Distinguished Teaching Award) от Ассоциации выпускников Технического колледжа (Alumni Association for the College of Engineering). Я участвую в программе усовершенствования преподавания (Wakonse Program for Teaching Excellence), а также являюсь членом Академии усовершенствования учителей (Academy for Educator Development).

Будучи сыном офицера, я легок на подъем. Получив степень по философии в университете Вашингтона, я двадцать два года прослужил в армии полевым артиллерийским офицером и аналитиком-исследователем по

[2] На веб-странице `http://www.stroustrup.com/bs_faq.html` автор очень подробно объясняет, что его норвежское имя правильно произносится как Беарне или, в крайнем случае, Бьярне, а не Бьорн и не Бьёрн, а фамилия читается как Строуструп, а не Страуструп. Однако по историческим причинам мы придерживаемся принятой в русскоязычной литературе транскрипции. — *Примеч. ред.*

опытной эксплуатации. С 1971 по 1973 год я прошел Высшие курсы полевых артиллерийских офицеров в Форт-Силле, Оклахома (Field Artillery Officer's Advanced Course at Fort Sill, Oklahoma). В 1979 я помог организовать Учебный центр офицеров-испытателей и с 1978 по 1981 год и с 1985 по 1989 год работал ведущим преподавателем на девяти разных должностях в разных регионах США.

В 1991 году я создал небольшую компанию, разрабатывавшую программное обеспечение для университетов вплоть до 1999 года. Мои интересы сосредоточены в области преподавания, проектирования и разработки программного обеспечения, предназначенного для реальных людей. Я получил магистерскую степень по техническим наукам в Технологическом институте штата Джорджия (Georgia Tech), а также магистерскую степень по педагогике в Техасском университете агрокультуры и машиностроения. Я также прошел программу подготовки магистров по микрокомпьютерам. Моя докторская диссертация по информатике и управлению написана в Техасском университете агрокультуры и машиностроения.

С женой Барбарой мы живем в г. Брайан, штат Техас. Я люблю путешествовать, ухаживать за садом и принимать гостей. Мы стараемся проводить как можно больше времени с нашими сыновьями и их семьями, особенно с внуками Ангелиной, Карлосом, Тесс, Эйвери, Николасом и Джорданом".

К несчастью, в 2007 году Пит умер от рака легкого. Без него этот курс никогда не достиг бы успеха.

Послесловие

Большинство глав завершается коротким послесловием, в котором излагается определенная точка зрения на информацию, изложенную в данной главе. Мы сделали это, хотя понимали, что она может ошеломить читателей (и часто на самом деле приводит их в замешательство) и что полностью уяснить ее можно, лишь выполнив упражнения и прочитав следующие главы (в которых будут применяться указанные идеи). Не паникуйте, расслабьтесь. Это вполне естественно и понятно. Вы не можете стать экспертом за один день, но, проработав книгу, можете стать вполне компетентным программистом. Кроме того, вы найдете в книге много фактов, примеров и приемов, которые многие программисты считают чрезвычайно интересными и поучительными.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

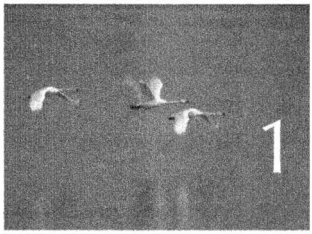
Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр 1
в Украине: 03150, Киев, а/я 152



Компьютеры, люди и программирование

Специализация нужна только насекомым.

— Р.Э. Хайнлайн (R.A. Heinlein)

В этой главе излагаются темы, которые, по нашему мнению, делают программирование важным, интересным и радостным занятием. Мы также описываем несколько фундаментальных идей и принципов, надеясь развеять множество распространенных мифов о программировании и программистах. Пока эту главу достаточно просто просмотреть, но мы рекомендуем вернуться к ней впоследствии, когда вы начнете самостоятельно программировать и засомневаетесь, а стоит ли этим заниматься вообще.

1.1. Введение

Как и любой процесс обучения, преподавание программирования сводится к дилемме о яйце и курице. Мы хотим поскорее начать работу, но одновременно желаем объяснить, почему именно эти темы выбрали для изучения. Мы хотим передать вам практические навыки, но одновременно хотим убедить вас, что это не причуда, а необходимость. Мы не желаем терять время, но при этом не хотим подгонять вас и читать проповеди. Пока что отнеситесь к данной главе, как к обычному интересному тексту, а затем можете вернуться к ней для того, чтобы освежить в памяти.

Эта глава выражает нашу личную точку зрения на то, что мы считаем интересным и важным для программирования. В ней изложены причины, по которым мы вот уже много десятков лет занимаемся этим делом. Она должна помочь вам понять, в чем заключаются наши основные цели и какими бывают программисты. Учебники для начинающих неизбежно содержат массу прописных истин. В этой главе мы закрываем глаза на технические детали и предлагаем рассмотреть картину в целом. Почему программирование является достойным занятием? Какую роль играет программирование в нашей цивилизации? В каких областях программисты могут сделать вклад, которым могли бы гордиться? Какие задачи остались пока нерешенными в области разработки, развертывания и эксплуатации программного обеспечения? Какое место занимает программирование в области компьютерных наук, разработки программного обеспечения, информационных технологий и т.д.? Чем занимаются программисты? Какими навыками они должны владеть?

Для студентов основной причиной изучения какой-либо идеи, метода или главы учебника может быть простое желание получить хорошую оценку на экзамене, но должны же быть и более веские мотивы! Для людей, работающих в области разработки программного обеспечения, наиболее важной причиной изучения какой-либо идеи, метода или главы учебника может быть желание узнать нечто, что поможет ему заслужить одобрение босса, от которого зависит повышение зарплаты, продвижение по службе

и увольнение, — но должны же быть и более веские мотивы! Лучше всего мы работаем, когда чувствуем, что наша работа делает мир лучше и помогает людям. Для задач, которые мы решаем годами (из которых, собственно, и складывается карьера профессионала), жизненно важными являются идеалы и более абстрактные идеи.

Жизнедеятельность нашей цивилизации зависит от программного обеспечения. Улучшение программного обеспечения и поиск новых областей для его применения позволят улучшить жизнь многих людей. Программирование играет в этом очень важную роль.

1.2. Программное обеспечение

Хорошее программное обеспечение невидимо. Вы не можете его потрогать, взвесить или стукнуть. Программное обеспечение — это совокупность программ, выполняемых на определенном компьютере. Иногда мы можем пощупать этот компьютер, но чаще можем увидеть лишь устройство, содержащее этот компьютер, например телефон, фотоаппарат, тестер, автомобиль или воздушную турбину. Мы способны воспринимать лишь результаты работы программного обеспечения. Кроме того, если работа программного обеспечения не соответствует нашим ожиданиям или потребностям, это может стать поводом для беспокойства.

Сколько компьютеров существует в мире? Мы не знаем; по крайней мере миллиарды. Компьютеров в мире больше, чем людей. Надо учесть все серверы, настольные компьютеры, смартфоны, планшеты и компьютеры, встроенные в различные изделия.

Сколько компьютеров (более или менее непосредственно) вы используете каждый день? Например, в моем автомобиле установлено более тридцати компьютеров, в мобильном телефоне — два, в MP3-плеере — один и еще один — в видеокамере. Кроме того, у меня есть еще ноутбук (на котором я набирал текст, который вы сейчас читаете), а также настольный компьютер. Контроллер кондиционера, который летом поддерживает комфортную температуру и влажность, также представляет собой простой компьютер. В нашем университете компьютер управляет работой эскалатора. Если вы пользуетесь современными телевизорами, то обязательно найдете в нем хотя бы один компьютер. Переходя с одной веб-страницы на другую, вы соединяетесь с десятками, а возможно, и сотнями серверов через телекоммуникационные системы, состоящие из многих тысяч компьютеров: телефонных коммутаторов, маршрутизаторов и т.д.

Нет, я не храню тридцать ноутбуков на заднем сидении своего автомобиля! Дело в том, что большинство компьютеров выглядят совсем не так, как мы представляем (с дисплеем, клавиатурой, мышью и т.д.); они просто являются небольшим устройством, встроенным в используемое

оборудование. Итак, в моем автомобиле нет ничего похожего на типичный компьютер, нет даже экрана для изображения карты и указания направления движения (хотя такие устройства весьма популярны). Однако двигатель моего автомобиля содержит несколько компьютеров, например, управляющих впрыском топлива и контролирующих его температуру. По крайней мере еще один компьютер встроен в рулевой механизм, в радиосистему и систему безопасности. Я подозреваю даже, что система, открывающая и закрывающая окна, управляется компьютером. В более современные автомобили встроены компьютеры, которые непрерывно контролируют даже давление в шинах.

От скольких компьютеров вы зависите на протяжении дня? Если вы живете в большом городе, то для того, чтобы получить еду, кто-то должен совершить небольшие чудеса, связанные с планированием, транспортировкой и хранением продуктов. Разумеется, управление сетями распределения продуктов компьютеризовано, как и работа любых коммуникационных систем, требующих согласованной работы. Современные фермы также компьютеризованы; на них можно найти компьютеры, используемые как в хлеву, так и в бухгалтерии (для учета возраста, состояния здоровья коров, надоев и т.д.). Фермеры все шире используют компьютеры для делопроизводства, причем количество отчетов, которые необходимо отправлять в разные правительственные агентства, приводит их в отчаянье. О том, что происходит в мире, вы прочтете в газете; разумеется, статья в этой газете будет набрана на компьютерах, сверстана на компьютерах и напечатана (если вы все еще читаете бумажную прессу) на компьютеризованном оборудовании — после передачи в типографию в электронном виде. Если вам потребуется связь с удаленным компьютером, то трафик будет управляться компьютерами, которые попытаются (как правило, тщетно) избежать узких мест. Вы предпочитаете ездить на поезде? Этот поезд тоже будет компьютеризован; некоторые из поездов даже ездят без помощи машинистов, причем большинство бортовых систем поезда (объявления по радио, торможение и продажа билетов) состоит из множества компьютеров. Современную индустрию развлечений (музыку, фильмы, телевидение, театрализованные представления) тоже невозможно представить без использования компьютеров. Даже художественные фильмы производятся с массовым применением компьютеров; музыкальные произведения и фотографии также все чаще основаны на цифровых технологиях (т.е. создаются с помощью компьютеров) как при записи, так и при распространении. Если вы заболеете, то все анализы доктор проведет с помощью компьютеров, записи в медицинских книжках будут внесены в электронном виде, а большинство медицинского оборудования в больницах окажется компьютеризованным. Если вы не живете в лесной избушке без доступа к электричеству, то используете электроэнергию. Нефть обнаруживается, извлекается и транспортируется по системе трубопроводов,

управляемых компьютерами на каждом этапе этого процесса: от погружения бура в землю до подачи топлива на местную насосную станцию. Если вы платите за бензин с помощью кредитной карточки, то снова задействуете огромное количество компьютеров. То же самое можно сказать о добыче и транспортировке угля, газа, а также солнечной и ветряной энергии.

Все приведенные выше примеры были "материальными"; они предусматривали непосредственное использование компьютеров в ежедневной жизнедеятельности. Не менее важной и интересной областью является проектирование. Одежда, которую вы носите, и кофеварка, в которой варите свой любимый кофе, были спроектированы и произведены с помощью компьютеров. Превосходное качество оптических линз в современных фотоаппаратах, а также утонченные формы современных технических устройств и посуды обеспечиваются компьютеризованным проектированием и производством. Производственники, проектировщики, артисты и инженеры, создающие окружающую нас среду, свободны от многих физических ограничений, которые ранее считались непреодолимыми. Если вы заболеете, то даже лекарство, которое выпишет врач, окажется синтезированным с помощью компьютеров.

В заключение отметим, что исследования — т.е. собственно наука — в целом основаны на использовании компьютеров. Телескопы, открывающие секреты далеких звезд, невозможно спроектировать, построить и эксплуатировать без помощи компьютеров, а огромные массивы данных, которые они производят, невозможно проанализировать и понять без компьютерной обработки. Отдельные биологические исследования иногда способны обойтись без широкого использования компьютеров (разумеется, если не учитывать фотоаппараты, устройства хранения данных, телефоны и т.д.), но данные все же необходимо как-то хранить, анализировать, сравнивать с компьютерными моделями и передавать другим ученым. Еще несколько лет назад невозможно было представить, насколько широко будут использоваться компьютеры в современной химии и биологии, включая медицинские исследования. Компьютеры смогли расшифровать геном человека. Точнее говоря, этот геном был расшифрован людьми с помощью компьютеров. Во всех этих примерах компьютер кардинально облегчает работу.

На каждом из упомянутых компьютеров установлено программное обеспечение. Без него компьютеры представляют собой просто дорогую груду кремния, металла и пластика, которую можно использовать лишь в качестве груза, якоря или обогревателя воздуха. Любое программное обеспечение создается людьми. Каждая строка программы, выполняемой компьютером, имеет определенный смысл. Очень странно, что все это работает! Ведь мы говорим о миллиардах строк программ, написанных на сотнях языков программирования. Для того чтобы все это правильно работало, люди затратили массу усилий и применили множество знаний. Нет ни одного устройства, работу которого мы не хотели бы усовершенствовать.

Достаточно просто задуматься о его функционировании, и вы сразу поймете, что именно требуется изменить. В любом случае каждое устройство можно уменьшить (или увеличить), заставить работать быстрее или надежнее, снабдить более широкими возможностями или мощностями, сделать красивее или дешевле. Очень вероятно, что для этого вам придется использовать программирование.

1.3. Люди

Компьютеры созданы людьми и для людей. Компьютер представляет собой универсальное устройство, его можно использовать для решения невероятно широкого спектра задач. Именно благодаря этому программы приносят пользу. Иначе говоря, компьютер — это просто груда железа, пока некто — программист — не напишет программу, делающую что-то полезное. Мы часто забываем о программном обеспечении. Еще чаще забываем о программисте.

Голливуд и другие "масскультовые" источники дезинформации создали программистам весьма негативный имидж. Например, по их мнению, программист — это одинокий толстый отщепенец, не имеющий социальных связей, не отрывающийся от видеоигр и постоянно залезающий в чужие компьютеры. Он (почти все программисты в фильмах — мужчины) стремится либо разрушить весь мир, либо спасти его. Разумеется, в реальном мире существуют люди, отдаленно напоминающие эту карикатуру, но наш опыт показывает, что среди программистов они встречаются не чаще, чем среди адвокатов, полицейских, продавцов автомобилей, журналистов, артистов и политиков.

Подумайте об известных вам компьютерных приложениях. Можете вспомнить приложение, используемое одиночкой в темной комнате? Конечно, нет! Разработка фрагментов программного обеспечения, компьютерных устройств или систем невозможна без совместной работы десятков, сотен и даже тысяч людей, играющих невероятно разнообразные роли. Например, среди них есть программисты, проектировщики, тестировщики, аниматоры, менеджеры фокус-групп, экспериментальные психологи, разработчики пользовательского интерфейса, аналитики, системные администраторы, специалисты по связям с потребителями, звукоинженеры, менеджеры проектов, инженеры по качеству, статистики, разработчики интерфейсов аппаратного обеспечения, специалисты по разработке технических заданий, сотрудники службы безопасности, математики, продавцы, ремонтники, проектировщики сетей, специалисты по методологии, менеджеры по разработке программного обеспечения, специалисты по созданию библиотек программ и т.п. Диапазон ролей огромен и неисчерпаем, при этом их названия варьируются: инженер в одной организации в

другой компании может называться программистом, разработчиком, членом технической команды или архитектором. Существуют даже организации, позволяющие своим сотрудникам самим выбирать, как называются их должности. Не все эти роли непосредственно связаны с программированием. Однако лично нам приходилось встречаться с людьми, основным занятием которых было читать или писать программы. Кроме того, программист (выполняющий одну или несколько из указанных ролей) может некоторое время контактировать со многими людьми из других прикладных областей: биологами, конструкторами автомобилей, адвокатами, продавцами автомобилей, медиками, историками, геологами, астронавтами, авиаконструкторами, менеджерами лесопилок, ракетостроителями, проектировщиками боулингов, журналистами и мультипликаторами (да, этот список — результат личного опыта). Некоторые из них также могут быть некоторое время программистами, а затем занять должность, не связанную с программированием.

Миф о программисте-одиночке — просто выдумка. Люди, предпочитающие самостоятельно выбирать задания, лучше всего соответствующие их способностям, обычно горько жалуются на то, что их часто отвлекают или вызывают на совещания. Люди, предпочитающие контактировать с другими людьми, чувствуют себя намного комфортнее, так как разработка современного программного обеспечения — коллективное занятие. По этой причине социальные связи и навыки общения для программистов имеют намного более высокую ценность, чем считается. Одним из наиболее желательных навыков программиста (*реального программиста*) является умение общаться с разными людьми — на совещаниях, посредством писем и на формальных презентациях. Мы убеждены, что, не завершив один-два коллективных проекта, вы не получите представления о том, что такое программирование и действительно ли оно вам нравится. В частности, мы любим программирование за возможность общаться с приятными и интересными людьми и посещать разные города.

Единственной сложностью является то, что все эти люди имеют разное образование, интересы и привычки, влияющие на производство хорошего программного обеспечения. От этих людей зависит качество нашей жизни — иногда даже сама жизнь. Ни один человек не может играть все роли, упомянутые выше; да и, вообще, ни один разумный человек к этому не стремится. Мы перечислили их, чтобы показать, что вы имеете намного более широкий выбор возможностей, чем можете себе представить. Можете перебирать разные области занятий, выбирая то, что лучше всего соответствует вашим умениям, талантам и интересам.

Мы все время говорим о программистах и программировании, но совершенно очевидно, что программирование — это только часть общей картины. Люди, разрабатывающие корабли или мобильные телефоны, не считают себя программистами. Кроме того, хотя программирование — важная

часть разработки программного обеспечения, разработка программного обеспечения — это не только программирование. Аналогично для большинства товаров разработка программного обеспечения — это важная часть производства, но не все производство.

Мы вовсе не предполагаем, что вы — наш читатель — стремитесь стать профессиональным программистом и провести всю оставшуюся жизнь в написании программ. Даже самые лучшие программисты — в особенности *лучшие* программисты! — тратят большую часть своего времени вовсе не на кодирование программ. Анализ задачи требует серьезных затрат времени и значительных интеллектуальных усилий. Именно за это многие программисты любят программирование. Помимо этого, многие из лучших программистов имеют научные степени по дисциплинам, которые не считаются частью компьютерных наук. Например, если вы работаете над программным обеспечением для исследований генома, ваша работа станет намного эффективнее, если вы будете разбираться в молекулярной биологии. Если же вы работаете над анализом средневековых текстов, то сможете написать гораздо более эффективные программы, если будете знать эту литературу и, возможно, один или несколько древних языков. В частности, люди, утверждающие, что их интересуют только компьютеры и программирование, обычно просто не в состоянии общаться с коллегами-не программистами. Такие люди не только лишены роскоши человеческого общения (собственно, самой жизни), но и, как правило, разрабатывают плохое программное обеспечение.

Итак, в чем же суть нашей точки зрения? Программирование — это набор навыков, требующих интеллектуальных усилий и образующих часть многих важных и интересных технических дисциплин. Кроме того, программирование — это существенная часть окружающего нас мира, поэтому не знать основы программирования — это то же самое, что не знать основ физики, истории, биологии или литературы. Если человек полностью игнорирует программирование, значит, он верит в чудеса и опасен на технической должности. Если вы читали Дилберта (Dilbert), то вспомните образ начальника с волосатыми пальцами и поймете, какой тип менеджера никогда не хотелось бы встретить в своей жизни и (что было бы еще ужаснее) каким руководителем вам никогда не хотелось бы стать. Кроме того, программирование часто бывает увлекательным занятием.

Так для чего же вам может понадобиться программирование? Может быть, вы будете использовать его как основной инструмент своих исследований, не став профессиональным программистом. Возможно, вы будете профессионально общаться с людьми, работая в областях, в которых знание программирования может дать существенное преимущество, например став конструктором, писателем, менеджером или ученым. Может быть, освоив программирование на профессиональном уровне, вы сможете сделать его частью своей работы. Даже если вы станете профессиональным

программистом, маловероятно, что вы не будете знать ничего, кроме программирования.

Вы можете стать инженером, занимающимся конструированием компьютеров, или специалистом по компьютерным наукам, но и в этом случае вы не будете все время программировать. Программирование — это способ выражения своих идей в виде программ. Это помогает решать задачи. Программирование становится совершенно бесполезной тратой времени, если у вас нет идей, которые вы хотели бы выразить, и нет задач, которые стоило бы решить.

Эта книга — о программировании, и мы пообещали научить вас программировать. Зачем же мы подчеркиваем важность остальных предметов и постоянно указываем на ограниченную роль программирования? Хороший программист понимает роль программ и техники программирования в работе над проектом. Хороший программист (в большинстве случаев) — это ответственный командный игрок, стремящийся сделать как можно более весомый вклад в решение общей задачи. Например, представьте себе, что, работая над новым MP3-плеером (например, в составе нового смартфона), я заботился бы лишь о красоте своей программы и количестве технических тонкостей. Вероятно, я стал бы настаивать на том, чтобы моя программа выполнялась на самом мощном компьютере. Я мог бы пренебречь теорией кодирования звука, поскольку эта дисциплина не относится к программированию. Я бы оставался в стенах своей лаборатории и не стал бы встречаться с потенциальными пользователями, у которых, несомненно, дурной музыкальный вкус и которые, конечно, не способны оценить новейшие достижения в области разработки графического пользовательского интерфейса. Вполне вероятно, что результаты моей работы были бы катастрофическими для всего проекта. Чем более мощный компьютер я бы потребовал, тем дороже стал бы MP3-плеер и тем быстрее разряжались бы его батареи питания. Существенную часть цифровой обработки музыки занимает ее кодирование, поэтому пренебрежение современными методами кодирования информации привело бы к завышенным требованиям к объему памяти для каждой песни (кодирование может уменьшить объем песни почти вдвое без потери качества ее звучания). Игнорирование предпочтений потенциальных пользователей — какими бы странными и архаичными они ни казались — обычно приводит к тому, что они выбирают другой продукт. При разработке хороших программ важно понимать потребности пользователей и ограничения, которые необходимо накладывать на программный код. Для того чтобы завершить карикатурный образ программиста, упомянем о тенденции опаздывать и нарушать сроки поставки из-за одержимости деталями и слепой веры в правильность плохо отлаженной программы. Мы желаем вам стать хорошим программистом с широким кругозором. Эти два качества одинаково полезны для общества и являются ключевыми для удовлетворения от своей работы.

1.4. Компьютерные науки

Даже в самом широком смысле программирование является частью более крупной научной дисциплины. Мы можем рассматривать ее как часть компьютерных наук, компьютерной техники, информационных технологий или другой научной дисциплины, связанной с программным обеспечением. Программирование является вспомогательной технологией, используемой как в информатике, так и в технике, физике, биологии, медицине, истории, литературе и других академических областях.

В 1995 году правительство США выпустило так называемую "Синюю книгу", в которой дано следующее определение компьютерных наук: "Систематическое изучение компьютерных систем и вычислений. Комплекс знаний, порождаемых этой дисциплиной, содержит теории компьютерных систем и методов; методы, алгоритмы и инструменты проектирования; методы проверки концепций; методы анализа и верификации; методы представления и реализации знаний". Как и следовало ожидать, Википедия дает более неформальное определение: "Компьютерные науки, или науки о вычислениях, изучают теоретические основы информации и вычислений, а также методы их реализации и приложения в компьютерных системах. Компьютерные науки состоят из многих подобластей; одни сосредоточены на конкретных вычислениях (например, компьютерная графика), другие (например, теория вычислительной сложности) изучают абстрактные проблемы вычислений, третьи связаны с реализацией вычислений. Например, теория языков программирования изучает подходы к описанию вычислений, в то время как компьютерное программирование применяет языки программирования для решения конкретных вычислительных задач".

Программирование — это основной инструмент выражения решений фундаментальных и прикладных задач, допускающий их проверку, уточнение с помощью эксперимента и использование на практике. Программирование — это дисциплина, в которой идеи и теории сталкиваются с реальностью; где компьютерные науки становятся экспериментальными, а не теоретическими и начинают влиять на реальный мир. В этом контексте, как и во многих других аспектах, важно подчеркнуть, что программирование — это средство реализации как практических, так и теоретических методов. Программирование нельзя сводить к простому ремесленничеству: мол, достаточно заполучить хоть какую-то программу, лишь бы она работала.

1.5. Компьютеры повсюду

Никто не знает всего о компьютерах и программах. В этом разделе мы приводим несколько примеров. Может быть, среди них вы найдете тот,

который вам понравится. По крайней мере, мы хотели бы убедить вас, что сфера применения компьютеров — а значит, и программирования — намного шире, чем можно себе представить.

Многие люди представляют себе компьютер как небольшой серый ящик с экраном и клавиатурой. Такие компьютеры, как правило, стоят под письменным столом и хорошо подходят для игр, обмена сообщениями, электронной почты и воспроизведения музыкальных произведений. Другие компьютеры, ноутбуки, используются в самолетах занятыми бизнесменами, просматривающими базы данных, играющими в компьютерные игры и просматривающими видеофильмы. Эта карикатура — всего лишь вершина айсберга. Большинство компьютеров вообще не попадает в поле нашего зрения, и именно от их работы зависит существование нашей цивилизации. Одни компьютеры занимают целые комнаты, а другие не больше монетки. Многие из наиболее интересных компьютеров вообще не имеют непосредственной связи с людьми, осуществляемой с помощью клавиатуры, мыши или других устройств.

1.5.1. С экранами и без них

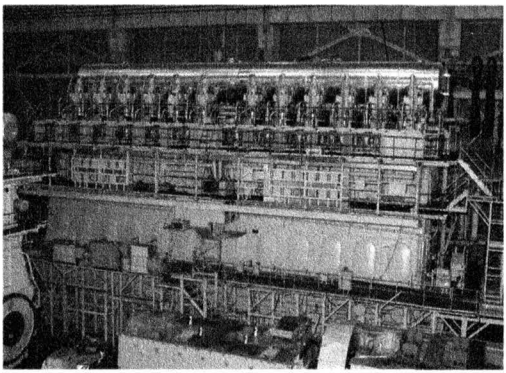
Представление о компьютере как о ящике с экраном и клавиатурой широко распространено и весьма устойчиво. Однако рассмотрим два следующих компьютера.

Оба компьютера можно увидеть непосредственно. Кроме того, будем считать, что они относятся к одной и той же модели, только с разными системами ввода-вывода. На левом устройстве время выводится на маленький экран (напоминающий экраны обычных компьютеров, но только поменьше), а справа отображается на традиционном циферблате, работающем под управлением небольшого электрического моторчика. Их системы ввода-вывода состоят из четырех кнопок (их легко обнаружить на

правом устройстве) и радиоприемника, обеспечивающего синхронизацию с высокоточными атомными часами. Большинство программ, управляющих этими двумя компьютерами, являются общими для них.

1.5.2. Кораблестроение

На этих двух фотографиях изображены крупный дизельный корабельный двигатель и огромный корабль, на котором может быть установлен такой двигатель.



Посмотрим, в каком месте компьютеры и программное обеспечение могли бы сыграть ключевую роль.

- *Проектирование.* Разумеется, как корабль, так и его двигатель проектируются с помощью компьютеров. Список их применения практически бесконечен и включает в себя изготовление архитектурных и инженерных чертежей, общие вычисления, визуализацию помещений и конструкций, а также моделирование их работы.
- *Строительство.* Современные корабли сильно компьютеризованы. Сборка корабля тщательно планируется с помощью компьютеров, а сам процесс осуществляется под управлением компьютеров. Сварка проводится роботами. В частности, современные двухкорпусные танкеры невозможно построить без маленьких роботов-сварщиков, способных проникнуть в пространство между корпусами. Там просто нет места, в которое мог бы протиснуться человек. Разрезание стальных плит для корабля было одним из первых приложений систем компьютерного проектирования и производства CAD/CAM (computer-aided design and computer-aided manufacture).
- *Двигатель.* Имеет электронную систему впрыска топлива и контролируется несколькими десятками компьютеров. Для двигателя мощностью 100 тысяч лошадиных сил (такого, какой изображен на фотографии) это нетривиальная задача. Например, компьютеры, уп-

равляющие двигателем, настраивают топливную смесь, чтобы минимизировать загрязнение. Многие насосы, связанные с двигателем (и другими частями корабля), также управляются компьютерами.

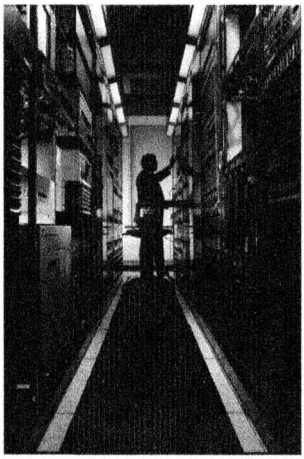
- *Управление.* Корабли предназначены для доставки груза и людей. Составление расписания движения флотилии кораблей — непрерывный процесс (который также выполняется компьютерами), поскольку он зависит от погоды, спроса и предложения, а также от грузоподъемности кораблей и вместимости портов. Существуют даже веб-сайты, с помощью которых можно отслеживать движение торговых кораблей. Корабль, изображенный на фотографии, — крупнейший в мире сухогруз (397 м в длину и 56 м в ширину), но другие крупные современные корабли управляются точно так же.
- *Мониторинг.* Океанские лайнеры в большой степени автономны; иначе говоря, их команды могут справиться с любыми неожиданностями еще до прибытия в следующий порт. Одновременно они являются частью глобальной системы, имеют доступ к достаточно точной метеорологической информации (через компьютеризованные космические спутники). Кроме того, у них имеются устройства глобального позиционирования (global positioning system — GPS), а также радары, управляемые компьютерами. Если команде нужен отдых, то за работой большинства систем (включая двигатель, радар и т.п.) можно следить (через спутник), находясь в центре управления кораблем. Если произойдет нечто необычное или связь будет нарушена, то команда сразу узнает об этом.

Что же произойдет, если один из многих сотен упомянутых компьютеров выйдет из строя? В главе 25 эта тема рассмотрена более подробно. Создание программ для современных кораблей — очень сложная и интересная работа. Кроме того, она приносит пользу. Стоимость транспортировки действительно удивительно мала. Вы оцените это, когда захотите купить какой-нибудь товар, произведенный в другом месте. Морской транспорт всегда был дешевле, чем наземный; в настоящее время одной из причин этого является широкое использование компьютеров и информации.

1.5.3. Телекоммуникации

На этих двух фотографиях изображены телефонный коммутатор и телефон (в который встроены фотоаппарат, MP3-плеер, FM-радиоприемник и веб-браузер).

Посмотрим, в каком месте компьютеры и программное обеспечение могли бы сыграть ключевую роль здесь. Допустим, вы берете в руку телефон и делаете вызов, а человек, которому вы звоните, отвечает вам, и вы



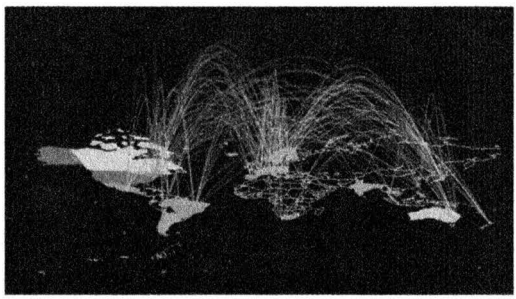
начинаете разговор. А возможно, вы хотите соединиться с автоматизированной справочной системой или отослать фотографию, сделанную с помощью встроенного фотоаппарата, или отправить текстовое сообщение (просто нажав кнопку "отправить" и поручив всю остальную работу телефону). Очевидно, что такой телефон является компьютером. Это особенно ясно, если телефон (как большинство мобильных телефонов) имеет экран и предусматривает больше, чем обычные телефонные услуги, например функции веб-браузера. На самом деле такие телефоны содержат несколько компьютеров: один управляет экраном, другой обеспечивает связь с телефонной станцией, а третий делает что-то еще.

Часть телефона, которая управляет экраном, выполняет функции веб-браузера и решает другие задачи, возможно, покажется пользователям компьютеров наиболее знакомой: она просто запускает графический пользовательский интерфейс. Большинство пользователей даже не представляют, с какой огромной системой соединяется такой маленький телефонный аппарат, выполняя свою работу. Допустим, я нахожусь в Техасе, а вы — в Нью-Йорке, но уже через несколько секунд ваш телефон зазвонит, и я услышу "Алло!" на фоне городского гула. Многие телефоны способны выполнить этот фокус и соединить вас с любой точкой Земли, и вы принимаете это как должное. Как телефон находит вас? Как передается звук? Как этот звук шифруется в пакетах? Ответы на эти вопросы могут заполнить много книг, но в целом для этого необходима согласованная работа аппаратного и программного обеспечения сотен компьютеров, разбросанных по всему миру. Если вам не повезет, то несколько телекоммуникационных спутников (которые сами представляют собой компьютерные системы) также включатся в работу. Мы говорим "не повезет", потому что не можем полностью компенсировать окольный путь длиной 20 тыс. миль; скорость света (а значит, скорость передачи вашего голоса) является конечной (оптоволоконные кабели значительно ускоряют передачу сигнала, будучи

короче и позволяя передавать куда большие потоки данных). Большинство этих функций выполняются отлично; коэффициент надежности основных телекоммуникационных систем достигает 99,9999% (например, они допускают лишь 20 минут простоя за 20 лет). Основные проблемы кроются в линиях связи между вашим мобильным телефоном и ближайшим телефонным коммуникатором.

Существует программное обеспечение, предназначенное для соединения телефонов, кодирования слов в виде пакетов сигналов для последующей передачи по проводам и радиоволнам, для маршрутизации сообщений, исправления любых неполадок, непрерывного мониторинга качества и надежности услуг, а также, конечно, для учета затраченного времени. Даже для простого слежения за физическими устройствами этой системы требуется большой объем сложного программного обеспечения. Кто с кем разговаривает? Какие части образуют новую систему? Когда следует провести превентивный ремонт?

Вероятно, основные телекоммуникационные мировые системы, состоящие из полуавтономных, но взаимосвязанных систем, являются самым крупным и сложным произведением человечества. Для того чтобы подчеркнуть это, напомним, что звонок по мобильному телефону — это не обычный звонок по старому телефону, у которого появилось несколько новых звуков. Он требует согласованной работы многих инфраструктур, являющихся также основой Интернета, банковских и коммерческих систем, кабельного телевидения. Работу телекоммуникаций можно также проиллюстрировать еще несколькими фотографиями.

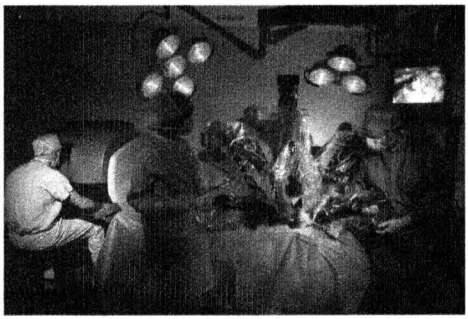


Помещение, изображенное на левой фотографии, представляет собой торговую площадку американской фондовой биржи на Уолл-стрит в Нью-Йорке, а карта демонстрирует часть Интернета (полная карта выглядит слишком запутанной).

Как видите, мы любим цифровую фотографию и используем компьютеры для изображения специальных карт, позволяющих визуализировать информацию.

1.5.4. Медицина

На следующих двух фотографиях продемонстрированы сканер компьютерной аксиальной томографии CAT и операционная для компьютерной хирургии (которая также называется роботохирургией).



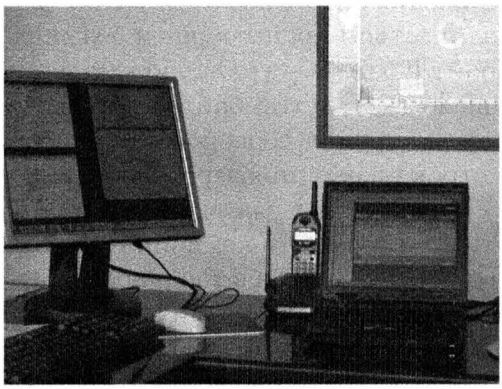
Посмотрим, в каком месте компьютеры и программное обеспечение могли бы сыграть ключевую роль здесь. Сканеры — это в основном компьютеры; излучаемые ими импульсы управляются компьютерами, но получаемая информация представляет собой неразбериху, пока не будет обработана сложными алгоритмами и преобразована в понятные трехмерные изображения соответствующей части тела. Для проведения хирургических операций с помощью компьютеров мы должны продвинуться еще дальше. Существует множество методов визуализации, позволяющих хирургу видеть внутренности пациента при наилучшем увеличении и освещении. С помощью компьютеров хирург может намного точнее оперировать инструментами, чем человеческая рука, и проникать в области, куда обычным способом без дополнительных разрезов дотянуться невозможно. Минимально инвазивная хирургия (лапароскопия) — это яркий пример медицинской технологии, позволяющей уменьшить боль до минимума и сократить время выздоровления миллионов людей. Компьютер может также помочь руке хирурга выполнить более тонкую работу, чем обычно. Кроме того, робототехническая система допускает дистанционное управление, позволяя доктору работать на расстоянии (например, через Интернет). Компьютеры и программы, связанные с этими системами, поразительно сложны и интересны. Разработка пользовательского интерфейса, средств управления оборудованием и методов визуализации в этих системах загрузит работой многие тысячи исследователей, инженеров и программистов на многие десятилетия.

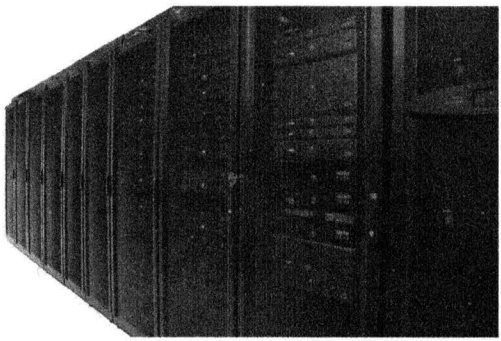
Среди медиков идет дискуссия о том, какой именно новый инструмент оказался наиболее полезным. Сканер компьютерной аксиальной томографии? Сканер магниторезонансной томографии? Аппараты для автоматического анализа крови? Ультразвуковые установки с высоким разрешением? Персональные информационные устройства? К удивлению многих,

"победителем" в этом "соревновании" стали устройства, обеспечивающие непрерывный доступ к записям о состоянии пациента. Знание истории болезни пациента (заболевания, которые он перенес, виды медицинской помощи, которая ему была оказана, аллергические реакции, наследственные проблемы, общее состояние здоровья, текущее лечение и т.д.) упрощает диагностику и минимизирует вероятность ошибок.

1.5.5. Информация

На следующих двух фотографиях изображены обычные персональные компьютеры и группа серверов.





Мы сосредоточились на аппаратных устройствах по вполне очевидным причинам: никто не в состоянии увидеть, потрогать или услышать программное обеспечение. Поскольку показать фотографию программы невозможно, мы демонстрируем оборудование, которое ее выполняет. Однако многие виды программного обеспечения непосредственно работают с информацией. Итак, рассмотрим обычное использование обычных компьютеров, выполняющих обычное программное обеспечение.

Группа серверов — это совокупность компьютеров, обеспечивающих веб-сервис. Спецификации таких групп, используемых крупнейшими организациями типа Google, Amazon или Microsoft, постоянно изменяются (так что информация, которую вы найдете о них в Интернете, будет устаревшей). Однако в любом случае эта система посложнее, чем организация вычислений на нескольких лэптопах.

- Google использует около миллиона серверов (каждый куда мощнее вашего лэптопа), расположенных в "центрах обработки данных" (data center) общим числом от 25 до 50.
- Такие центры обработки данных располагаются в помещениях, которые могут иметь размер $60 \times 100\,м^2$ и более.

- В 2011 году *New York Times* сообщила, что постоянное потребление электроэнергии центрами обработки данных Google — около 260 МВт (почти такое же количество электроэнергии потребляет Лас-Вегас).
- Пусть сервер представляет собой четырехъядерную машину с частотой 3 ГГц и 24 Гбайт памяти. Тогда можно говорить о вычислительной мощности порядка 12×10^{15} Гц с памятью 24×10^{15} 8-битовых байт, ну а диск объемом 4 Тбайта на сервер дает общий объем хранилища, равный 4×10^{18} байт.

Мы могли несколько занизить наши оценки, а к тому времени, когда книга попадет к вам в руки, это будут наверняка заниженные величины. Как нам представляется, требование снижения энергозатрат будет вести к применению архитектур с большим количеством процессоров и большим количеством ядер в процессоре. Гигабайт (Гбайт) — это около миллиарда символов. Терабайт (Тбайт) — это около тысячи гигабайтов, т.е. около триллиона символов. За прошедшее время группа серверов Google стала намного больше. Это достаточно экстремальный пример, но почти каждая крупная компания использует веб-программы для взаимодействия с пользователями и клиентами. Достаточно вспомнить компании Amazon (книжная и другая торговля), Amadeus (продажа авиабилетов и аренда автомобилей) и eBay (интернет-аукционы). Миллионы небольших компаний, организаций и частных лиц также работают в сети веб. Большинство из них не используют собственное программное обеспечение, но многие все же пишут свои программы, которые часто бывают совсем не тривиальными.

Более традиционным является использование компьютеров для ведения бухгалтерии, учета заказов, платежей и счетов, управления запасами, учета кадров, ведения баз данных, хранения записей о студентах, персонале, пациентах и т.п. Эти записи хранят практически все организации (коммерческие и некоммерческие, правительственные и частные), поскольку они составляют основу их работы. Компьютерная обработка таких записей выглядит просто: в большинстве случаев информация (записи) просто записывается в память компьютера и извлекается из его памяти, и очень редко обрабатывается с его помощью. Приведем некоторые примеры.

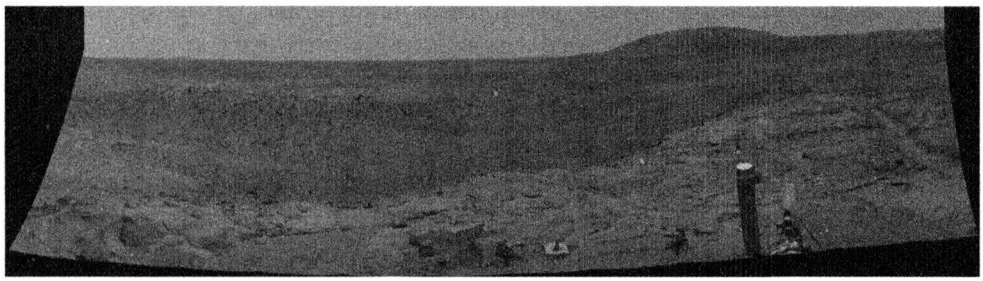
- Не откладывается ли вылет самолета рейсом в Чикаго в 12:30?
- Болел ли Гильберт Салливан корью?
- Поступила ли на склад кофеварка, которую заказал Хуан Вальдез?
- Какую кухонную мебель купил Джек Спрат в 1996 году и покупал ли он ее вообще?
- Сколько телефонных звонков поступило из зоны 212 в августе 2012 года?
- Сколько кофейных чашек было продано в январе и чему равна их совокупная стоимость?

Из-за крупного масштаба баз данных эти системы весьма сложны. К тому же ответы на вопросы следует давать быстро (часто на протяжении не более двух секунд) и правильно (по крайней мере, почти всегда). Сегодня трудно кого-то удивить терабайтами данных (байт — это единица памяти, необходимая для хранения обычного символа). Эта традиционная обработка данных часто сочетается с доступом к базам данных через веб.

Этот вид использования компьютеров часто называют *обработкой информации* (information processing). Он сосредоточен на данных — как правило, на крупных объемах данных — и создает интересные проблемы, связанные с организацией и передачей информации, а также со сжатым представлением огромных массивов данных: пользовательский интерфейс представляет собой важный аспект обработки данных. Например, представьте себя на месте исследователя средневековой литературы (скажем, "*Кентерберийских рассказов*" Чосера или "*Дон Кихота*" Сервантеса) и подумайте, каким образом можно было бы выяснить, кто именно из многих десятков гипотетических авторов на самом деле написал анализируемый текст. Для этого пришлось бы выполнить поиск по тексту, руководствуясь множеством критериев, сформулированных литературоведами, а также вывести результаты, позволяющие выявить скрытые особенности этих произведений. Размышляя об анализе текстов, вы непременно вспомните и о сегодняшних публикациях: нынче любая статья, книга, брошюра, газета производится на компьютере. Разработка программного обеспечения, облегчающего публикацию текстов, для большинства людей остается задачей, не имеющей удовлетворительного решения.

1.5.6. Вид сверху

Говорят, что палеонтологи способны полностью реконструировать динозавра, описать его образ жизни и естественную среду, изучив лишь одну его маленькую косточку. Возможно, это покажется преувеличением, но иногда полезно изучить простой артефакт и подумать, какие следствия он влечет. Посмотрите на фотографию марсианского ландшафта, сделанную марсоходом NASA.



Для того чтобы заниматься космонавтикой, стоит стать хорошим программистом. В многочисленных космических программах участвует огромное количество программистов, хорошо знающих физику, математику, электротехнику, механику, медицинскую технику и тому подобное, т.е. все научные дисциплины, лежащие в основе исследования космоса. Управление двумя марсоходами — одно из крупнейших достижений нашей цивилизации. Один из них (*Spirit*) отсылал данные на Землю 6 лет, а второй (*Opportunity*) на момент написания этой книги продолжает работать и отметил десятилетие пребывания на Марсе в январе 2014 года. Заметим, что рассчитаны они были на работу в течение трех месяцев.

Эта фотография пришла на Землю по каналу связи с 25-минутной задержкой; при этом большое количество искушенных программистов и талантливых математиков сделали все возможное, чтобы эта картинка была закодирована минимальным количеством битов без потери хотя бы одного бита. На Земле фотография обработана с помощью алгоритмов, восстанавливающих цвет и минимизирующих искажения, возникающие из-за несовершенства оптических приборов и электронных сенсоров.

Программы управления марсоходами, конечно, являются компьютерными программами, — двигатель марсохода работает автономно и круглосуточно, подчиняясь командам, посылаемым с Земли за день до их выполнения. Передача команд управляется программами.

Операционные системы, используемые на разных компьютерах, вовлеченных в управление марсоходами, передачей команд и реконструкцией фотографий, ничем не отличаются от приложений, используемых при написании этой главы. Компьютеры, на которых запускаются эти программы, в свою очередь, разработаны и созданы с помощью систем компьютерного проектирования и производства CAD/CAM. Микросхемы, входящие в состав этих компьютеров, произведены на компьютеризованных сборочных линиях с использованием высокоточных инструментов, причем сами эти инструменты спроектированы и созданы с помощью компьютеров (и программного обеспечения). Управление качеством этого долгого процесса конструирования связано с серьезными вычислениями. Все эти программы написаны людьми на языках программирования высокого уровня и переведены в машинный код компиляторов, которые сами являются программами. Многие из этих программ взаимодействуют с пользователями с помощью графического пользовательского интерфейса и обмениваются данными через потоки ввода-вывода.

Кроме того, большое количество программистов занимаются обработкой изображений (в том числе обработкой фотографий, поступающих с марсохода), анимацией и редактированием фотографий (по сети веб "гуляют" варианты марсианских фотографий, на которых изображены "марсиане").

1.5.7. И что?

Какое отношение все эти превосходные и сложные приложения и системы программного обеспечения имеют к изучению программирования и языка C++? Связь довольно очевидная — для того чтобы специалисты могли успешно выполнять такие проекты, они должны хорошо знать программирование и языки программирования. Кроме того, каждый пример в этой главе связан с языком C++ и по крайней мере с одним из методов программирования, описанных в книге. Да, программы, написанные на C++, работают и в MP3-плеерах, и на кораблях, и в воздушных турбинах, и на Марсе, и в проекте по расшифровке генома человека. Многие другие приложения, созданные с использованием языка C++, вы найдете на веб-странице `www.stroustrup.com/applications.html`.

1.6. Идеалы программистов

Чего мы ждем от наших программ вообще? Чего хотим от конкретной программы в частности? Мы хотим от программ *корректности* (чтобы программа работала правильно) и *надежности*. Если программа не делает то, что от нее требуется, или работает ненадежно, то в лучшем случае это серьезный нюанс, а в худшем — опасность. При этом мы хотим, чтобы программа была *хорошо спроектирована*, т.е. удовлетворяла наши реальные потребности; на самом деле совершенно не важно, работает ли программа правильно, если она делает не то, что задумано, или правильно выполняет задание, но способ, которым она это делает, вызывает тревогу. Кроме того, мы хотим, чтобы программа была экономичной; возможно, я предпочел бы ездить на роллс-ройсе или летать на корпоративном самолете, но пока я не миллиардер, я должен учитывать стоимость этого удовольствия.

Именно эти аспекты программного обеспечения (оборудования, систем) могут быть по достоинству оценены не программистами. Они должны служить для программистов идеалами, которые следует иметь в виду постоянно, если мы хотим разработать качественное программное обеспечение, в особенности на ранних стадиях проектирования. Мы должны также учитывать требования к самому коду: он должен быть легко *поддерживаемым*; т.е. его структура должна быть такой, чтобы любой другой программист мог понять его и внести свои изменения. Успешная программа "живет" долго (часто десятки лет), постоянно изменяясь. Например, она может быть выполнена на новом аппаратном обеспечении, получить новые возможности, адаптироваться к новым средствам ввода-вывода (экраны, видео, звук), взаимодействовать с пользователями на новых естественных языках и т.д. Невозможно модифицировать только неправильную программу. Для удобства поддержки программа должна быть

относительно простой, а ее код должен непосредственно выражать идеи, лежащие в ее основе. Сложность — враг простоты и удобства — может быть присуща самой проблеме (в этом случае мы должны просто как-то с ней справиться), но она может быть и следствием неудачного выражения идей, заложенных в программе. Мы должны избегать этого, придерживаясь хорошего стиля программирования, — стиль имеет значение!

Звучит довольно просто, но это далеко не так. Почему? В программировании, по существу, нет ничего сложного: вы просто сообщаете компьютеру, что вы от него хотите. Почему же оно может потребовать большого напряжения сил? Ведь в компьютерах тоже нет ничего сложного; они просто выполняют определенные наборы операций, например складывают два числа и выбирают следующую инструкцию в зависимости от результата их сравнения. Проблема заключается в том, что мы не используем компьютеры для решения простых задач. Мы хотим, чтобы они решали задачи, которые нам самим не под силу, но при этом забываем, что вычислительные машины — это придирчивые, ничего не прощающие и безмолвные существа. Кроме того, мир устроен гораздо более сложно, чем нам кажется, поэтому мы часто просто не представляем, к каким последствиям могут привести наши запросы. Мы просто хотим, чтобы программа "делала что-то вроде этого", и не вникаем в техническое детали. А еще мы часто опираемся на некий "здравый смысл". К сожалению, даже среди людей встречаются разные точки зрения на здравый смысл, а уж у компьютеров его вообще нет (хотя некоторые действительно хорошо спроектированные программы могут имитировать здравый смысл в конкретных, подробно изученных ситуациях).

Такой образ мышления приводит к заключению, что "программирование — это понимание": если вы можете запрограммировать задачу, значит, понимаете ее. И наоборот, если вы глубоко разобрались в задаче, то сможете написать и программу для ее решения. Иначе говоря, программирование можно рассматривать как часть усилий по исследованию проблемы. Программы — это точное представление нашего понимания проблемы. Когда вы программируете, то проводите много времени, пытаясь понять задачу, которую хотите автоматизировать.

Процесс разработки программ можно разделить на четыре этапа.

- *Анализ.* В чем заключается задача? Чего хочет пользователь? Что требуется пользователю? Что может позволить себе пользователь? Какая степень надежности нам необходима?
- *Проектирование.* Как решить задачу? Какую структуру должна иметь система? Из каких частей она должна состоять? Каким образом эти части будут взаимодействовать? Каким образом система будет взаимодействовать с пользователем?

- *Программирование.* Выражение решения задачи (проекта) в исходном тексте. Написание программы с учетом всех установленных ограничений (по времени, объему, финансам, надежности и т.д.). Проверка, что программа работает правильно и удобна в сопровождении.
- *Тестирование.* Проверка, что во всех предусмотренных ситуациях система работает правильно.

Программирование и тестирование часто называют *реализацией*. Очевидно, что описанное разделение разработки программ на четыре части является упрощением. По указанным четырем темам написаны толстые книги, и еще больше книг написано о том, как эти темы взаимосвязаны между собой. Следует помнить, что эти стадии проектирования не являются независимыми и на практике не следуют именно в таком порядке. Обычно мы начинаем с анализа, но обратная связь на этапе тестирования влияет на программирование; проблемы, возникающие на этапе программирования, могут свидетельствовать о проблемах, не решенных на этапе проектирования; в свою очередь, проектирование может выявить аспекты, не учтенные на этапе анализа. На самом деле функционирование системы обычно сразу же выявляет слабость анализа.

Чрезвычайно важным обстоятельством является *обратная связь*. Мы учимся на ошибках и уточняем свое поведение, основываясь на том, чему мы учимся. Это очень важно для эффективной разработки программного обеспечения. В работе над любым крупным проектом нам неизвестна вся информация о проблеме и ее решении, пока мы не приступаем к делу. Конечно, опробовать идеи и проанализировать обратную связь можно и на этапе программирования, но на более ранних стадиях разработки это можно сделать намного легче и быстрее, записав идеи, проработав их и испытав на друзьях. По нашему мнению, наилучшим инструментом проектирования является меловая доска (если вы предпочитаете химические запахи, а не запах мела, то можете использовать доску для фломастеров). По возможности никогда не проектируйте в одиночку! Никогда не начинайте писать программу, пока не опробуете свои идеи, объяснив их кому-то еще. Обсуждение проекта и методов проектирования с друзьями, коллегами, потенциальными пользователями и другими людьми следует проводить еще до того, как вы сядете за клавиатуру. Просто удивительно, как много можно узнать, просто попытавшись объяснить свою идею словами. Помимо всего прочего, программа — это всего лишь средство выражения идей в виде кода.

Аналогично, попав в тупик при реализации программы, оторвитесь от клавиатуры. Подумайте о самой задаче, а не о своем неполном решении этой задачи. Поговорите с кем-нибудь: объясните, что вы хотели и почему программа не работает. Просто удивительно, как часто можно найти

решение, просто кому-то подробно объяснив задачу. Не занимайтесь отладкой программ (поиском ошибок) в одиночку, если у вас есть возможность делать это с кем-то!

В центре внимания нашей книги лежат реализация и особенно программирование. Мы не учим решать задачи, заваливая вас грудой примеров и решений. Часто новую задачу можно свести к уже известной и применить традиционный метод ее решения. Только после того, как большая часть подзадач будет обработана таким образом, можно позволить себе увлекательное "свободное творчество". Итак, сосредоточимся на методах выражения идей в виде программ.

Непосредственное выражение идей в виде программ — это основная цель программирования. Это совершенно очевидно, но до сих пор мы еще не привели достаточно ярких примеров. Мы еще не раз будем возвращаться к этому. Если в нашей программе необходимо целое число, мы храним его в виде переменной типа `int`, предусматривающего основные операции с целыми числами. Если мы хотим работать со строками символов, то храним их в виде переменных типа `string`, обеспечивающего основные операции по манипуляции текстом. В идеале, если у нас есть идея, концепция, сущность или какая-то "вещь", которую можно изобразить на доске и сослаться на нее в ходе дискуссии, о которой написано в учебнике (по некомпьютерным наукам), то мы хотим, чтобы это нечто существовало в нашей программе в виде именованной сущности (типа), предусматривающей требуемые операции. Если мы собираемся проводить математические вычисления, то нам потребуются тип `complex` для комплексных чисел и тип `Matrix` для матриц. Если хотим рисовать, то потребуются типы `Shape` (Фигура), `Circle` (Круг), `Color` (Цвет) и `Dialog_box` (Диалоговое окно). Если хотим работать с потоками данных, скажем, поступающих от датчика температуры, то нам понадобится тип `istream` (буква "`i`" означает "ввод" (Input)). Очевидно, что каждый такой тип должен обеспечивать совершенно конкретный набор предусмотренных операций. Мы привели лишь несколько примеров из книги. Кроме них, мы опишем инструменты и методы, позволяющие создавать собственные типы, описывающие любые концепции, необходимые для вашей программы.

Программирование носит частично практический, частично теоретический характер. Если вы ограничитесь его практическими аспектами, то будете создавать немасштабируемые и трудные для сопровождения поделки. Если же захотите остаться теоретиком, то будете разрабатывать непрактичные (и не экономичные) игрушки.

Различные точки зрения на идеалы программирования и биографии людей, внесших значительный вклад в создание языков программирования, изложены в главе 22.

Контрольные вопросы

Контрольные вопросы предназначены для выделения основных идей, изложенных в главе. Их можно рассматривать как дополнение к упражнениям. В то время как упражнения подчеркивают практический аспект, контрольные вопросы посвящены идеям и концепциям.

1. Что такое программное обеспечение?
2. Чем объясняется важность программного обеспечения?
3. В чем проявляется важность программного обеспечения?
4. Что может произойти, если программное обеспечение будет работать неправильно? Приведите несколько примеров.
5. В каких областях программное обеспечение играет важную роль? Приведите несколько примеров.
6. Какие виды деятельности связаны с разработкой программного обеспечения? Приведите несколько примеров.
7. В чем разница между компьютерными науками и программированием?
8. Где в процессе проектирования, конструирования и использования кораблей используется программное обеспечение?
9. Что такое группа серверов?
10. Какие запросы вы посылаете по сети? Приведите примеры.
11. Как программное обеспечение используется в научных исследованиях? Приведите примеры.
12. Как программное обеспечение используется в медицине? Приведите примеры.
13. Как программное обеспечение используется в индустрии развлечений? Приведите примеры.
14. Какими свойствами должно обладать хорошее программное обеспечение?
15. Как выглядит разработчик программного обеспечения?
16. Перечислите этапы разработки программного обеспечения.
17. Чем могут объясняться трудности разработки программного обеспечения? Назовите несколько причин.
18. Как программное обеспечение может облегчить жизнь?
19. Как программное обеспечение может осложнить жизнь?

Термины

Приведенные термины входят в основной словарь по программированию и языку C++. Чтобы понимать, что люди говорят о программировании, и озвучивать собственные идеи, следует понимать их смысл.

CAD/CAM	корректность	проектирование
анализ	обратная связь	реализация
графический пользовательский интерфейс	пользователь	стереотип
заказчик	программирование	тестирование
идеалы	программист	школьная доска
коммуникации	программное обеспечение	экономичность

Упражнения

1. Перечислите виды деятельности, которыми вы занимаетесь большую часть времени (например, ходите в университет, едите или смотрите телевизор). Укажите среди них те виды деятельности, которые более или менее тесно связаны с компьютерами.
2. Укажите профессию, которой вы хотели бы овладеть или о которой вы что-нибудь знаете. Перечислите виды деятельности, связанные с этой профессией и компьютерами.
3. Отдайте список, заполненный при выполнении упр. 2, своему другу и возьмите у него аналогичный список, посвященный некоторой иной профессии. Уточните его список. Когда вы оба сделаете это, сравните результаты. Помните: упражнения, допускающие разные решения, не имеют однозначного ответа, поэтому они всегда могут уточняться.
4. Опишите виды деятельности, которые, исходя из вашего жизненного опыта, невозможны без компьютеров.
5. Перечислите программы (программное обеспечение), которые вы используете непосредственно. Укажите только программы, с которыми вы взаимодействуете прямо (например, выбирая новую композицию на MP3-плеере), а не перечисляйте программы, которые могут быть установлены на используемых вами компьютерах (например, при вращении руля в вашем автомобиле).
6. Укажите десять видов деятельности, занимаясь которыми, люди никак не используют компьютеры, даже косвенно. Это упражнение сложнее, чем кажется!
7. Укажите пять задач, для решения которых компьютеры в настоящее время не используются, но в будущем, по вашему мнению, будут использоваться. Обоснуйте свой ответ.

8. Объясните, чем вам нравится программирование (используя не менее 100, но не более 500 слов). Если же вы убеждены, что не станете программистом, то объясните, почему. В любом случае приведите продуманные и логичные аргументы.
9. Опишите роль, помимо профессии программиста (независимо от ответа, данного выше), которую вы хотели бы играть в компьютерной индустрии (используя не менее 100, но не более 500 слов).
10. Могут ли компьютеры когда-нибудь стать сознательными и мыслящими существами, конкурирующими с человеком? Обоснуйте свою точку зрения (используя не менее 100 слов).
11. Перечислите свойства, присущие наиболее успешным программистам. После этого укажите характеристики, которые общественное мнение приписывает программистам.
12. Назовите пять приложений компьютерных программ, упомянутых в главе, и укажите одно из них, которое считаете наиболее интересным и в разработке которого хотели бы принять участие. Обоснуйте свою точку зрения (используя не менее 100 слов).
13. Сколько памяти может понадобиться для хранения а) этой страницы текста, б) этой главы и в) всех произведений Шекспира? Будем считать, что для хранения одного символа требуется один байт, а допустимая точность ответа составляет 20%.
14. Какой объем памяти вашего компьютера? Какой объем оперативной памяти? Какой объем жесткого диска?

Послесловие

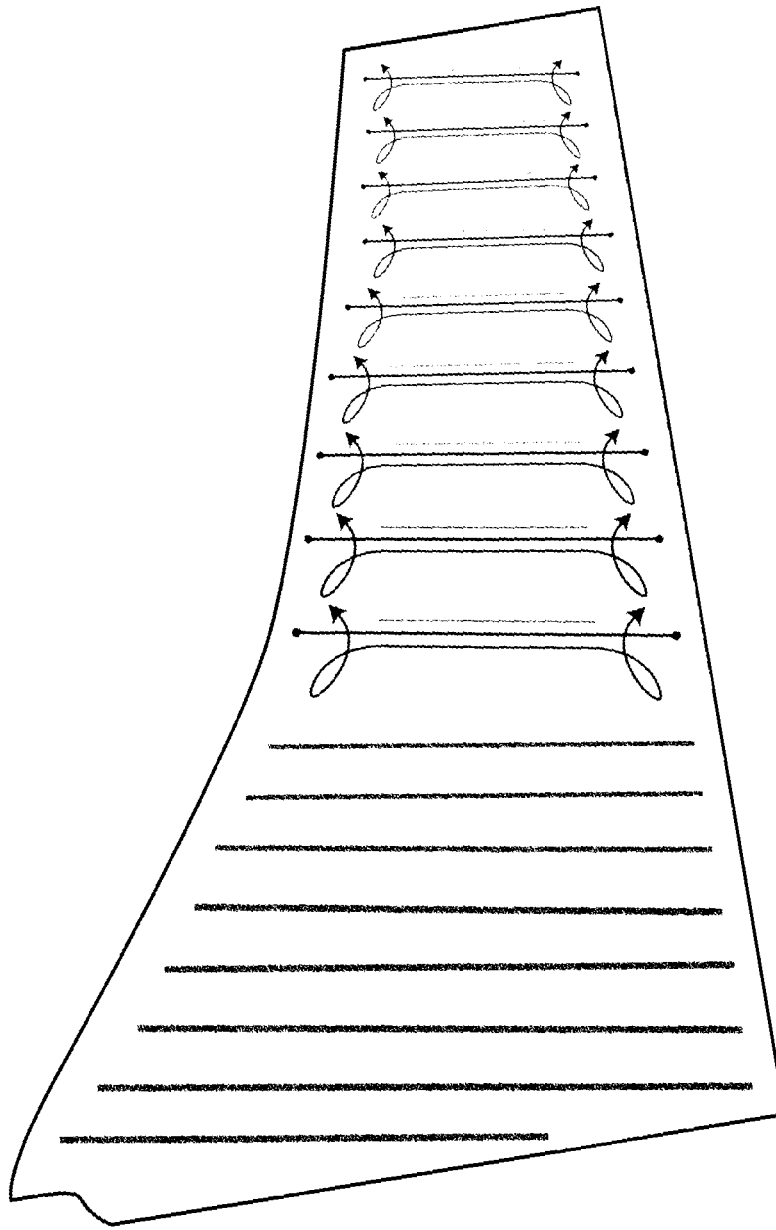
Жизнедеятельность нашей цивилизации зависит от программного обеспечения. Разработка программного обеспечения — это область невероятно разнообразных возможностей для интересной, социально полезной и прибыльной работы. Создавая программное обеспечение, необходимо быть принципиальным и серьезным: необходимо устранять проблемы, а не создавать их.

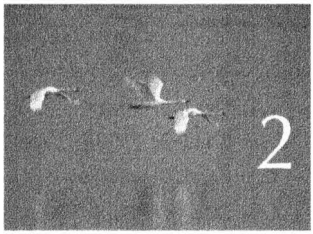
Разумеется, мы испытываем благоговение перед программным обеспечением, пронизывающим всю нашу техническую цивилизацию. Конечно, не все программы хороши, но это другая история. Здесь мы хотим подчеркнуть, насколько широко распространено программное обеспечение и как сильно зависит от них наша повседневная жизнь. Все эти программы написаны людьми, похожими на нас. Все эти ученые, математики, инженеры, программисты и другие специалисты начинали примерно так же, как и вы.

Теперь вернемся на землю и приступим к овладению техническими навыками, необходимыми для программирования. Если вы начнете

сомневаться, стоит ли заниматься этой трудной работой (большинство разумных людей время от времени думают об этом), вернитесь назад, перечитайте эту главу, предисловие и часть главы 0. Если начнете сомневаться, сможете ли справиться с этой работой, помните, что миллионы людей справляются с ней и становятся компетентными программистами, проектировщиками, разработчиками программного обеспечения и т.д. Мы уверены — вы не исключение и тоже сумеете с ней справиться.

Часть I
Основы





2

Hello, World!

Программирование изучается с помощью написания программ.
— Брайан Кернеган (Brian Kernighan)

В этой главе приводится простейшая программа на языке C++, которая что-то делает. Цели создания этой программы следующие.

- Дать вам возможность поработать с интегрированной средой разработки программ.
- Дать вам почувствовать, как можно заставить компьютер сделать что-то для вас.

Словом, в этой главе мы представим понятие программы, идею о преобразовании программ с помощью компилятора из текстовой формы, понятной для человека, в машинные команды для последующего выполнения компьютером.

2.1. Программы

Для того чтобы заставить компьютер сделать что-то, вы (или кто-то еще) должны точно рассказать ему — со всеми подробностями, — что именно от него требуется. Описание того, "что следует сделать", называется *программой*, а *программирование* — это вид деятельности, который заключается в создании и отладке таких программ.

В некотором смысле мы все уже программисты. В конце концов, мы множество раз получали описательные задания, которые должны были выполнить, например "как идти в школу на уроки" или "как поджарить мясо в микроволновой печи". Разница между такими описаниями и программами заключается в степени точности: люди стараются компенсировать неточность инструкций, руководствуясь здравым смыслом, а компьютеры этого сделать не могут. Например, "по коридору направо, вверх по лестнице, а потом налево" — вероятно, прекрасная инструкция, позволяющая найти нужный кабинет на верхнем этаже. Однако если вы внимательно посмотрите на эти простые инструкции, то увидите, что они являются грамматически неточными и неполными. Человек может легко восполнить этот недостаток. Представим, например, что вы вошли в здание и спрашиваете, как найти кабинет нужного вам человека. Отвечающий вам совершенно не обязан говорить, чтобы вы прошли через дверь, открыв ее поворотом ручки (а не выбив ее ногой), не толкнули в холле вазон с цветами и т.д. Вам также никто не скажет, чтобы вы были осторожны, поднимаясь по лестнице, и постучали, добравшись до нужной двери. Как открыть дверь в кабинет, прежде чем войти в него, вам, вероятно, также не будут рассказывать.

В противоположность этому компьютер *действительно* глуп. Ему все необходимо точно и подробно описать. Вернемся к инструкциям "по коридору направо, вверх по лестнице, а потом налево". Где находится коридор? Что такое коридор? Что значит "направо"? Что такое лестница? Как подняться по лестнице? По одной ступеньке? Через две ступеньки? Держась за перила? Что находится слева от меня? Когда это окажется слева от меня? Для того чтобы подробно описать инструкции для компьютера, необходим точно определенный язык, имеющий специфическую грамматику (естественный язык слишком слабо структурирован), а также хорошо

определенный словарь для всех видов действий, которые мы хотим выполнить. Такой язык называется *языком программирования*, и язык программирования C++ — один из таких языков, разработанных для решения широкого круга задач.

Более широкие философские взгляды на компьютеры, программы и программирование изложены в главе 1. Здесь же мы рассмотрим код, начиная с очень простой программы, а также несколько инструментов и методов, необходимых для ее выполнения.

2.2. Классическая первая программа

Приведем вариант классической первой программы. Она выводит на экран сообщение **Hello, World!**.

```
// Эта программа выводит на экран сообщение "Hello,World!"
#include "std_lib_facilities.h"
int main() // Программы на C++ начинаются с выполнения функции main
{
    cout << "Hello, World!\n"; // Вывод "Hello,World!"
    return 0;
}
```

Рассматривайте этот текст как набор команд, которые должен выполнить компьютер; это напоминает кулинарный рецепт или инструкции по сборке новой игрушки. Посмотрим, что делает каждая из строк программы, начиная с самого начала:

```
cout << "Hello, World!\n"; // Вывод "Hello,World!"
```

Именно эта строка выводит сообщение на экран. Она выводит символы **Hello, World!**, за которыми следует символ перехода на новую строку; иначе говоря, после вывода символов **Hello, World!** курсор будет установлен на начало новой строки. *Курсор* — это небольшой мерцающий символ или линия, показывающая, где будет выведен следующий символ.

В языке C++ строковые литералы выделяются двойными кавычками ("); т.е. **"Hello, Word!\n"** — это строка символов. **\n** — это "специальный символ", означающий переход на новую строку. Имя **cout** относится к стандартному потоку вывода. Символы, "помещенные в поток **cout**" с помощью оператора вывода **<<**, будут отображены на экране. Имя **cout** произносится как "see-out", но является аббревиатурой от "**c**haracter **out**put stream" ("поток вывода символов"). Аббревиатуры довольно широко распространены в программировании. Естественно, все эти сокращения на первых порах могут показаться неудобными для запоминания, но, привыкнув, вы уже не сможете от них отказаться, так как они позволяют создавать короткие и управляемые программы.

Конец строки

```
// Вывод "Hello,World!"
```

является комментарием. Все, что написано после символа // (т.е. после двойной косой черты (/), которая называется слэшем), считается комментарием. Он игнорируется компилятором и предназначен для программистов, которые будут читать программу. В данном случае мы использовали комментарий для того, чтобы сообщить вам, что именно означает первая часть этой строки.

Комментарии описывают предназначение программы и содержат полезную информацию для людей, которую невозможно выразить в коде. Скорее всего, человеком, который извлечет пользу из ваших комментариев, окажетесь вы сами, когда вернетесь к своей программе на следующей неделе или на следующий год, забыв, для чего вы ее писали. Итак, старайтесь хорошо документировать свои программы. В разделе 7.6.4 мы обсудим, как писать хорошие комментарии.

Программа пишется для двух аудиторий. Естественно, мы пишем программы для компьютеров, которые будут их выполнять. Однако мы долгие годы проводим за чтением и модификацией кода. Таким образом, второй аудиторией программ являются другие программисты. Поэтому создание программ можно считать формой общения между людьми. В действительности имеет смысл главными читателями своей программы считать людей: если они с трудом понимают, что вы написали, то вряд ли программа когда-нибудь станет правильной. А потому нельзя забывать, что код предназначен для чтения — необходимо делать все, чтобы программа легко читалась. В любом случае комментарии нужны только людям; компьютеры их полностью игнорируют.

Первая строка программы — это типичный комментарий, который сообщает читателям, что будет делать программа.

```
// Эта программа выводит на экран сообщение "Hello,World!"
```

Такие комментарии очень полезны, так как по исходному тексту программы можно понять, что она делает, но нельзя выяснить, что мы на самом деле хотели. Кроме того, в комментариях мы можем намного лаконичнее объяснить цель программы, чем в самом коде (как правило, более подробном). Часто такие комментарии размещаются в начальной части программы и напоминают, что мы пытаемся сделать в данной программе.

Строка

```
#include "std_lib_facilities.h"
```

представляет собой директиву **#include**. Она заставляет компьютер сделать доступными ("включить") функциональные возможности, описанные в файле **std_lib_facilities.h**. Этот файл упрощает использование

возможностей, предусмотренных во всех реализациях языках C++ (стандартной библиотеке языка C++).

По мере продвижения вперед мы объясним эти возможности более подробно. Они написаны на стандартном языке C++, но содержат детали, в которые сейчас не стоит углубляться, отложив их изучение до следующих глав. Важность файла **std_lib_facilities.h** для данной программы заключается в том, что с его помощью мы получаем доступ к стандартным средствам ввода-вывода языка C++. Здесь мы используем только стандартный поток вывода **cout** и оператор вывода **<<**. Файл, включаемый в программу с помощью директивы **#include**, обычно имеет суффикс **.h** и называется *заголовком* (header), или *заголовочным файлом* (header file). Заголовок содержит определения терминов, таких как **cout**, которые мы используем в нашей программе.

Как компьютер узнает, с чего начинать выполнение программы? Он ищет функцию с именем **main** и начинает выполнять инструкции, содержащиеся в ней. Вот как выглядит функция **main** нашей программы:

```
int main() // Программы на C++ начинаются с выполнения функции main
{
    cout << "Hello, World!\n"; // Вывод "Hello,World!"
    return 0;
}
```

Для того чтобы указать отправную точку выполнения, каждая программа на языке C++ должна содержать функцию с именем **main**. По сути, функция представляет собой именованную последовательность инструкций, которую компьютер выполняет в порядке их перечисления. Функция состоит из четырех частей.

- *Тип возвращаемого значения*; в этой функции — тип **int** (т.е. integer, целое число), определяет, какой результат возвращает функция в точку вызова (если она возвращает какое-нибудь значение). Слово **int** является зарезервированным в языке C++ (*ключевым словом*), поэтому его нельзя использовать в качестве имени чего-нибудь иного (см. раздел A.3.1).
- *Имя*; в данном случае **main**.
- *Список параметров*, заключенный в круглые скобки (см. разделы 8.2 и 8.6); в данном случае список параметров пуст и имеет вид **()**.
- *Тело функции*, заключенное в фигурные скобки **{}** и перечисляющее действия (*инструкции*), которые функция должна выполнить.

Отсюда следует, что минимальная программа на языке C++ выглядит так:

```
int main() { }
```

Пользы от такой программы мало, так как она ничего не делает. Тело функции **main** программы "Hello, World!" содержит две инструкции:

```
cout << "Hello, World!\n"; // Вывод "Hello,World!"
return 0;
```

Сначала она выводит на экран строку **Hello, World!**, а затем возвращает значение 0 (нуль) в точку вызова. Поскольку функция **main()** вызывается системой, мы возвращаемое значение использовать не будем. Однако в некоторых системах (в частности, Unix/Linux) это значение можно использовать для проверки успешности выполнения программы. Нуль (0), возвращаемый функцией **main()**, означает, что программа выполнена успешно.

Часть программы на языке C++, определяющая некоторое действие и не являющаяся директивой **#include** (или другой директивой препроцессора; см. разделы 4.4 и A.17), называется *инструкцией* (statement).

2.3. Компиляция

C++ — компилируемый язык. Это означает, что для запуска программы сначала необходимо транслировать ее из текстовой формы, понятной для человека, в форму, понятную для машины. Эту задачу выполняет особая программа, которая называется *компилятором*. То, что вы пишете и читаете, называется *исходным кодом* или *исходным текстом программы*, а то, что выполняет компьютер, называется *выполняемым*, *объектным* или *машинным кодом*. Обычно файлы с исходным кодом программы на языке C++ имеют суффикс **.cpp** (например, **hello_world.cpp**) или **.h** (например, **std_lib_facilities.h**), а файлы с объектным кодом имеют суффикс **.obj** (в Windows) или **.o** (в Unix). Следовательно, простое слово *код* является двусмысленным и может ввести в заблуждение; его следует употреблять с осторожностью и только в ситуациях, когда недоразумение возникнуть не может. Если не указано иное, под словом *код* подразумевается "исходный код" или даже "исходный код за исключением комментариев", поскольку комментарии предназначены для людей и компилятор не переводит их в объектный код.

Компилятор читает исходный код и пытается понять, что вы написали. Он проверяет, является ли программа грамматически корректной, определен ли смысл каждого слова. Обнаружив ошибку, компилятор сообщает о ней, не пытаясь выполнить программу. Компиляторы довольно придирчивы к синтаксису. Пропуск какой-нибудь детали, например директивы

#include, двоеточия или фигурной скобки, приводит к ошибке. Кроме того, компилятор точно так же абсолютно нетерпим к опечаткам. Продемонстрируем это рядом примеров, в каждом из которых сделана одна небольшая ошибка. Каждая из этих ошибок является довольно типичной.

```
// пропущен заголовочный файл
int main()
{
     cout << "Hello, World!\n";
     return 0;
}
```

Мы не сообщили компилятору о том, что представляет собой объект **cout**, поэтому он сообщает об ошибке. Для того чтобы исправить программу, следует добавить директиву **#include**.

```
#include "std_facilities.h"
int main()
{
     cout << "Hello, World!\n";
     return 0;
}
```

К сожалению, компилятор снова сообщает об ошибке, так как мы сделали опечатку в строке **std_lib_facilities.h**. Компилятор заметил это.

```
#include "std_lib_facilities.h"
int main()
{
     cout << "Hello, World!\n;
     return 0;
}
```

В этом примере мы пропустили закрывающую двойную кавычку ("). Компилятор указывает нам на это.

```
#include "std_lib_facilities.h"
integer main()
{
     cout << "Hello, World!\n";
     return 0;
}
```

Теперь мы вместо ключевого слова **int** использовали слово **integer**, которого в языке C++ нет. Компилятор таких ошибок не прощает.

```
#include "std_lib_facilities.h"
int main()
{
     cout < "Hello, World!\n";
     return 0;
}
```

Здесь вместо символов << (оператор вывода) использован символ < (оператор "меньше"). Компилятор это заметил.

```
#include "std_lib_facilities.h"
int main()
{
    cout << 'Hello, World!\n';
    return 0;
}
```

Здесь вместо двойных кавычек, ограничивающих строки, по ошибке использованы одинарные. Приведем заключительный пример.

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n"
    return 0;
}
```

В этой программе мы забыли завершить строку, содержащую оператор вывода, точкой с запятой. Обратите внимание на то, что в языке C++ каждая инструкция завершается точкой с запятой (;). Компилятор распознает точку с запятой как символ окончания инструкции и начала следующей. Трудно коротко, неформально и технически корректно описать все ситуации, в которых нужна точка с запятой. Пока просто запомните правило: точку с запятой следует ставить после каждого выражения, которое не завершается закрывающей фигурной скобкой }.

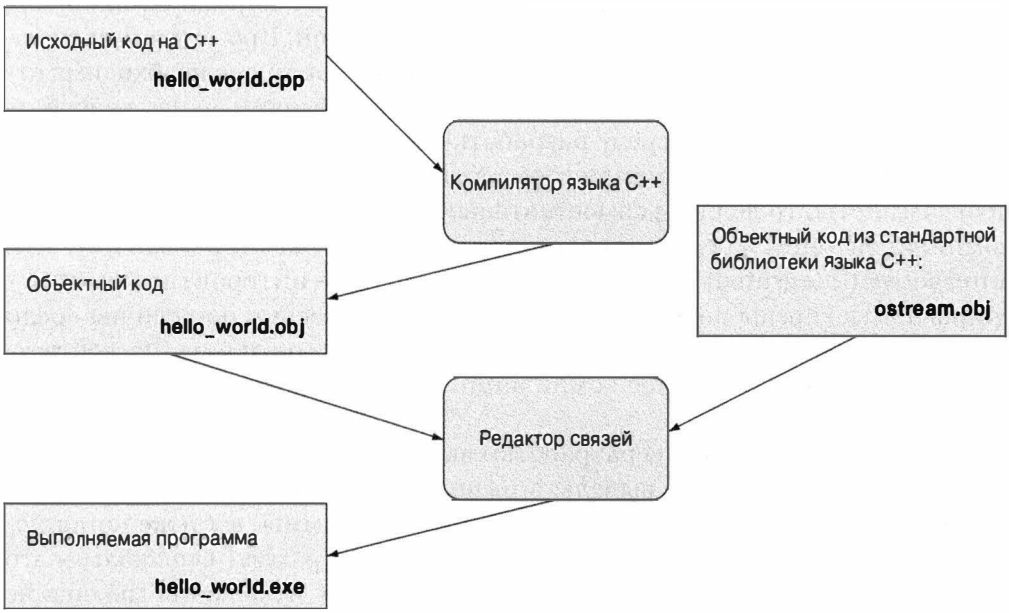
Для чего мы посвятили две страницы и несколько минут вашего драгоценного времени демонстрации тривиальных примеров, содержащих тривиальные ошибки? Для того, чтобы в будущем вы не тратили много времени на поиск ошибок в исходном тексте программы. Большую часть времени программисты ищут ошибки в своих программах. В конце концов, если вы убеждены, что некий код является правильным, то вы, скорее всего, обратитесь к анализу некоторого другого кода, чтобы не тратить время зря. На заре компьютерной эры первые программисты сильно удивлялись, насколько часто они делали ошибки и как долго их искали. И по сей день большинство начинающих программистов удивляются этому не меньше.

Компилятор нередко будет вас раздражать. Иногда будет казаться, что он придирается к несущественным деталям (например, к пропущенным точкам с запятыми) или к вещам, которые вы считаете абсолютно правильными. Однако компилятор, как правило, не ошибается: если уж он выводит сообщение об ошибке и отказывается создавать объектный код из вашего исходного кода, то это значит, что ваша программа не в порядке; иначе говоря, то, что вы написали, не соответствует стандарту языка C++.

Компилятор не руководствуется здравым смыслом (это не человек!) и очень придирчив к деталям. Поскольку здравый смысл ему чужд, он не пытается угадать, что на самом деле вы имели в виду, написав фрагмент программы, который выглядит абсолютно правильным, но не соответствует стандарту языка C++. Если бы он угадывал смысл программы, но при этом результат оказался бы для вас совершенно неожиданным, вы провели бы очень много времени, пытаясь понять, почему программа не делает то, что требуется. Компилятор — наш друг, позволяющий избежать множества проблем, которые мы сами же создаем. Никогда не забывайте об этом: компилятор — не просто ваш друг; пожалуй, когда вы занимаетесь программированием, — это наилучший ваш друг.

2.4. Редактирование связей

Программа обычно состоит из нескольких отдельных частей, которые часто разрабатываются разными людьми. Например, программа "Hello, World!" состоит из части, которую написали вы, и частей стандартной библиотеки языка C++. Эти отдельные части (иногда называемые *единицами трансляции*) должны быть скомпилированы, а файлы с результирующим объектным кодом должны быть скомпонованы в единое целое, образуя выполняемый файл. Программа, связывающая эти части в одно целое, называется (вполне ожидаемо) компоновщиком или *редактором связей*.



Заметьте, что объектные и выполняемые коды являются *не* переносимыми из одной системы в другую. Например, когда вы компилируете программу для Windows, то получите объектный код именно для Windows, и этот код не будет работать в Linux.

Библиотека — это просто некоторый код (обычно написанный другими), доступ к которому можно получить с помощью объявлений, содержащихся в файле, включенном директивой `#include`. *Объявление* — это инструкция программы, указывающая, как можно использовать некоторый фрагмент кода; объявления будут подробно описаны позднее (см., например, раздел 4.5.2).

Ошибки, обнаруженные компилятором, называются *ошибками времени компиляции*; ошибки, обнаруженные компоновщиком, называются *ошибками времени компоновки*, а ошибки, не обнаруженные на этих этапах и проявляющиеся при выполнении программы, называются *ошибками времени выполнения* или *логическими ошибками*. Как правило, ошибки времени компиляции легче понять и исправить, чем ошибки времени компоновки. В свою очередь, ошибки времени компоновки легче обнаружить и исправить, чем ошибки времени выполнения. Ошибки и способы их обработки более детально обсуждаются в главе 5.

2.5. Среды программирования

Для программирования необходим язык программирования. Кроме того, для преобразования исходного кода в объектный нужен компилятор, а для редактирования связей нужен редактор связей. Кроме того, для ввода и редактирования исходного текста в компьютере также необходима отдельная программа. Эти инструменты, крайне необходимые для разработки программы, образуют среду разработки программ.

Если вы работаете с командной строкой, как многие профессиональные программисты, то должны самостоятельно решать проблемы, связанные с компилированием и редактированием связей. Если же вы используете IDE (interactive (integrated) development environment — интерактивные (интегрированные) среды разработки), которые также весьма популярны среди профессиональных программистов, то достаточно щелкнуть на соответствующей кнопке. Описание компиляции и редактирования связей приведено в приложении B.

Интегрированные среды разработки включают в себя редактор текстов, позволяющий, например, выделять разными цветами комментарии, ключевые слова и другие части исходного кода программы, а также помогающий отладить, скомпилировать и выполнить программу. *Отладка* — это поиск и исправление ошибок в программе (по ходу изложения мы еще не раз вспомним о ней).

Работая с этой книгой, вы можете использовать любую систему, предоставляющую современную, соответствующую стандарту реализацию C++. Большинство из того, о чем мы говорим, с очень малыми модификациями справедливо для всех реализаций языка C++, и приводимый нами код будет работать везде. В нашей работе мы используем несколько разных реализаций.

Задание

До сих пор мы говорили о программировании, коде и инструментах (например, о компиляторах). Теперь нам необходимо выполнить программу. Это очень важный момент в изложении и в обучении программированию вообще. Именно с этого начинается усвоение практического опыта и овладение хорошим стилем программирования. Упражнения в этой главе предназначены для того, чтобы вы освоились с вашей интегрированной средой программирования. Запустив программу "Hello, World!" на выполнение, вы сделаете первый и главный шаг как программист.

Цель задания — закрепить ваши навыки программирования и помочь вам приобрести опыт работы со средами программирования. Как правило, задание представляет собой последовательность модификаций какой-нибудь простой программы, которая постепенно "вырастает" из совершенно тривиального кода в нечто полезное и реальное. Для выявления вашей инициативы и изобретательности предлагаем набор традиционных упражнений. В противоположность им задания не требуют особой изобретательности. Как правило, для их выполнения важна последовательность пошагового выполнения действий, каждое из которых должно быть простым (и даже тривиальным). Пожалуйста, не умничайте и не пропускайте описанные шаги, поскольку это лишь тормозит работу или сбивает с толку.

Вам может показаться, что вы уже все поняли, прочитав книгу или прослушав лекцию преподавателя, но для выработки навыков необходимы повторение и практика. Этим программирование напоминает спорт, музыку, танцы и любое другое занятие, требующее упорных тренировок и репетиций. Представьте себе музыканта, который репетирует от случая к случаю. Можно себе представить, как он играет. Постоянная практика — а для профессионала это означает непрерывную работу на протяжении всей жизни — это единственный способ развития и поддержания профессиональных навыков.

Итак, никогда не пропускайте заданий, как бы вам этого ни хотелось; они играют важную роль в процессе обучения. Просто начинайте с первого шага и продолжайте, постоянно перепроверяя себя.

Не беспокойтесь, если вы не понимаете все тонкости используемого синтаксиса, и не стесняйтесь просить помощи у преподавателей или друзей. Работайте, выполняйте все задания и большинство упражнений, и со временем все прояснится.

Итак, вот первое задание.

1. Откройте приложение B, и выполните все шаги, необходимые для настройки проекта. Создайте пустой консольный проект на C++ под названием **hello_world**.
2. Введите текст файла **hello_world.cpp** в точности таким, как показано ниже, сохраните его в рабочем каталоге и включите его в проект **hello_world**.

```
#include "std_lib_facilities.h"
int main() // Программы на C++ начинаются с выполнения функции main
{
     cout << "Hello, World!\n"; // Вывод строки "Hello, World!"
     keep_window_open(); // Ожидание ввода символа
     return 0;
}
```

   Вызов функции **keep_window_open()** нужен при работе под управлением некоторых версий операционной системы Windows для того, чтобы окно не закрылось прежде, чем вы прочитаете строку вывода. Это особенность вывода системы Windows, а не языка C++. Для того чтобы упростить разработку программ, мы поместили определение функции **keep_window_open()** в файл **std_lib_facilities.h**.

   Как найти файл **std_lib_facilities.h**? Если вы учитесь с преподавателем, спросите у него. Если работаете самостоятельно, загрузите его с сайта **www.stroustrup.com/Programming**. А если у вас нет ни преподавателя, ни доступа к вебу? В этом (и только в этом!) случае замените директиву **#include** строками

```
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>
#include<cmath>
using namespace std;
inline void keep_window_open() { char ch; cin>>ch; }
```

   В этих строках непосредственно используется стандартная библиотека. Подождите до главы 5 и еще более подробного изложения в разделе 8.7.

3. Скомпилируйте и выполните программу "Hello, World!". Вполне вероятно, что у вас это сразу не получится. Очень редко первая попытка использовать новый язык программирования или новую среду

разработки программ завершается успехом. Найдите источник проблем и устраните его! В этот момент целесообразно заручиться поддержкой более опытного специалиста, но при этом вы должны убедиться, что понимаете, что именно он сделал и почему, и сможете повторить эти действия в дальнейшем самостоятельно.

4. Возможно, вы столкнетесь с наличием ошибок в программе и будете должны их исправить. Тогда самое время познакомиться с тем, как ваш компилятор находит ошибки и сообщает о них программисту! Посмотрите, как отреагирует компилятор на шесть ошибок, описанных в разделе 2.3, внося их и пытаясь скомпилировать программу. Придумайте еще как минимум пять других ошибок в вашей программе (например, пропустите вызов функции `keep_window_open()`, наберите ее имя в верхнем регистре или поставьте запятую вместо точки с запятой) и посмотрите, что произойдет при попытке скомпилировать и выполнить эту программу.

Контрольные вопросы

Основная идея контрольных вопросов — дать вам возможность выяснить, насколько хорошо вы усвоили основные идеи, изложенные в главе. Вы можете найти ответы на эти вопросы в тексте главы; это нормально и вполне естественно, можете перечитать все разделы, и это тоже нормально и естественно. Но если даже после этого вы не можете ответить на контрольные вопросы, то вам следует задуматься о том, насколько правильный способ обучения вы используете? Возможно, вы слишком торопитесь. Может быть, имеет смысл остановиться и попытаться поэкспериментировать с программами? Может быть, вам нужна помощь друга, с которым вы могли бы обсуждать возникающие проблемы?

1. Для чего предназначена программа "Hello, World!"?
2. Назовите четыре части функции.
3. Назовите функцию, которая должна иметься в каждой программе на языке C++.
4. Для чего предназначена строка `return 0` в программе "Hello,World!"?
5. Для чего предназначен компилятор?
6. Для чего предназначена директива `#include`?
7. Что означает суффикс `.h` после имени файла в языке C++?
8. Что делает редактор связей?
9. В чем заключается различие между исходным и объектным файлами?
10. Что такое интегрированная среда разработки и для чего она предназначена?
11. Если в книге вам все понятно, то зачем нужна практическая работа?

Обычно контрольный вопрос имеет ясный ответ, явно сформулированный в главе. Однако иногда мы включаем в этот список вопросы, связанные с информацией, изложенной в других главах и даже в других книгах. Мы считаем это вполне допустимым; для того чтобы научиться писать хорошие программы и думать о последствиях их использования, мало прочитать одну главу или книгу.

Термины

Приведенные термины входят в основной словарь по программированию и языку C++. Чтобы понимать, что люди говорят о программировании, и озвучивать собственные идеи, следует понимать их смысл.

`#include`	библиотека	компилятор
`//`	вывод	объектный код
`<<`	выполняемый файл	ошибка времени компиляции
`C++`	заголовок	программа
`cout`	инструкция	редактор связей
IDE	исходный код	функция
`main()`	комментарий	

Можете пополнять этот словарь самостоятельно, выполняя приведенное ниже пятое упражнение для каждой прочитанной главы.

Упражнения

Мы приводим задания отдельно от упражнений; прежде чем приступать к упражнениям, необходимо выполнить все задания. Тем самым вы сэкономите время.

1. Измените программу так, чтобы она выводила две строки:

```
Hello, programming!
Here we go!
```

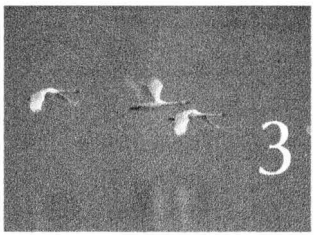
2. Используя приобретенные знания, напишите программу, содержащую инструкции, с помощью которых компьютер нашел бы кабинет на верхнем этаже, о котором шла речь в разделе 2.1. Можете ли вы указать большее количество шагов, которые подразумевают люди, а компьютер — нет? Добавьте эти команды в ваш список. Это хороший способ научиться думать, как компьютер. Предупреждаем: для большинства людей "иди в указанное место" — вполне понятная команда. Для людей, которые никогда не видели современного строения (например, для перемещенных во времени неандертальцев), этот список может оказаться *очень* длинным. Пожалуйста, не делайте его больше страницы. Для удобства читателей можете изобразить схему строения.

3. Напишите инструкции, как пройти от входной двери вашего дома до двери вашей аудитории (будем считать, что вы студент; если нет, выберите другую цель). Покажите их вашему другу и попросите уточнить их. Для того чтобы не потерять друзей, неплохо бы сначала испытать эти инструкции на себе.
4. Откройте хорошую поваренную книгу и прочитайте рецепт изготовления булочек с черникой (если в вашей стране это блюдо является экзотическим, замените его каким-нибудь более привычным). Обратите внимание на то, что, несмотря на небольшое количество информации и инструкций, большинство людей вполне способны выпекать эти булочки, следуя рецепту. При этом никто не считает этот рецепт сложным и доступным лишь профессиональным поварам или искусным кулинарам. Однако, по мнению автора, лишь некоторые упражнения из нашей книги можно сравнить по сложности с рецептом по выпечке булочек с черникой. Удивительно, как много можно сделать, имея лишь небольшой опыт!
   - Перепишите эти инструкции так, чтобы каждое отдельное действие было указано в отдельном абзаце и имело номер. Подробно перечислите все ингредиенты и всю кухонную утварь, используемую на каждом шаге. Не пропустите важные детали, например желательную температуру, предварительный нагрев духовки, подготовку теста, время выпекания и средства защиты рук при извлечении булочек из духовки.
   - Посмотрите на эти инструкции с точки зрения новичка (если вам это сложно, попросите об этом друга, ничего не понимающего в кулинарии). Дополните рецепт информацией, которую автор (разумеется, опытный кулинар) счел очевидной.
   - Составьте словарь использованных терминов. (Что такое противень? Что такое предварительный разогрев? Что подразумевается под духовкой?)
   - Теперь приготовьте несколько булочек и насладитесь результатом.
5. Напишите определение каждого из терминов, включенных в раздел "Термины". Сначала попытайтесь сделать это, не заглядывая в текст главы (что маловероятно), а затем перепроверьте себя, найдя точное определение в тексте. Возможно, вы обнаружите разницу между своим ответом и нашей версией. Можете также воспользоваться каким-нибудь доступным глоссарием, например, размещенным по адресу `www.stroustrup.com/glossary.html`. Формулируя свое описание, вы закрепите полученные знания. Если для этого вам пришлось перечитать главу, то это пойдет вам только на пользу. Можете пересказывать смысл термина своими словами и уточнять его по своему разумению.

Часто для этого полезно использовать примеры, размещенные после основного определения. Целесообразно записывать свои ответы в отдельный файл, постепенно добавляя в него новые термины.

Послесловие

Почему программа "Hello, World!" так важна? Ее цель — ознакомить вас с основными инструментами программирования. Мы стремились использовать для этого максимально простой пример. Так мы разделяем обучение на две части: сначала изучаем основы новых инструментов на примере тривиальных программ, а затем исследуем более сложные программы, уже не обращая внимания на инструменты, с помощью которых они написаны. Одновременное изучение инструментов программирования и языка программирования намного сложнее, чем овладение этими предметами по отдельности. Этот подход, предусматривающий разделение сложной задачи на ряд более простых задач, не ограничивается программированием и компьютерами. Он носит универсальный характер и используется во многих областях, особенно там, где важную роль играют практические навыки.



3

Объекты, типы и значения

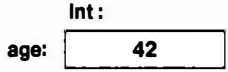
Фортуна благоволит подготовленному уму.
— Луи Пастер (Louis Pasteur)

В этой главе излагаются основы хранения и использования данных в программе. Сначала мы сосредоточим внимание на вводе данных с клавиатуры. После введения фундаментальных понятий объектов, типов, значений и переменных рассмотрим несколько операторов и приведем много примеров использования переменных типов **char**, **int**, **double** и **string**.

3.1. Ввод

Программа "Hello, World!" просто записывает текст на экран. Она осуществляет вывод. Она ничего не считывает, т.е. не получает ввода от пользователя. Это довольно скучно. Реальные программы, как правило, производят результаты на основе каких-то данных, которые мы им даем, а не делают одно и то же каждый раз, когда мы их запускаем.

Для того чтобы считать данные, необходимо место, куда можно ввести информацию; иначе говоря, нужно какое-то место в памяти компьютера, чтобы разместить в нем то, что мы прочтем. Мы называем такое "место" объектом. *Объект* — это место в памяти, имеющее *тип*, который определяет вид информации, которая может храниться в этом месте. Именованный объект называется *переменной*. Например, строки символов вводятся в переменные типа **string**, а целые числа — в переменные типа **int**. Объект можно интерпретировать как "коробку", в которую можно поместить значение, имеющее тип объекта.



Например, на рисунке изображен объект типа **int** с именем **age**, содержащий целочисленное значение **42**. Используя строковую переменную, мы можем считать строку с устройства ввода и вывести ее на экран, как показано ниже.

```
// Считать и записать имя
#include "std_lib_facilities.h"
int main()
{
    cout << "Введите ваше имя (и нажмите 'enter'):\n";
    string first_name; // first_name – переменная типа string
    cin >> first_name; // Считываем символы в first_name
    cout << "Привет, " << first_name << "!\n";
}
```

Директива **#include** и функция **main()** известны нам из главы 2. Поскольку директива **#include** необходима во всех наших программах (вплоть до главы 12), мы отложим ее изучение, чтобы не запутывать ситуацию. Аналогично иногда мы будем приводить код, который будет работать только будучи помещенным в тело функции **main()** или некоторой иной наподобие кода

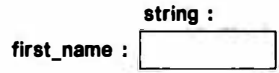
```
cout << "Введите ваше имя (и нажмите 'enter'):\n";
```

Мы считаем, что вы умеете добавить такой код в полную программу и выполнить ее тестирование.

Первая строка функции **main()** просто выводит на экран сообщение, предлагающее пользователю ввести свое имя. Такое сообщение называется *приглашением*, поскольку предлагает пользователю предпринять какое-то действие. Следующие строки определяют переменную типа **string** с именем **first_name**, считывают данные с клавиатуры в эту переменную и выводят на экран приветствие. Рассмотрим эти строки по очереди.

```
string first_name; // first_name – это переменная типа string
```

Эта строка выделяет участок памяти для хранения строки символов и присваивает ему имя **first_name**.

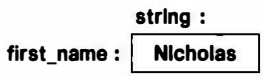


Инструкция, вводящая новое имя в программе и выделяющая память для переменной, называется *определением*.

Следующая строка считывает символы с устройства ввода (клавиатуры) в переменную:

```
cin >> first_name; // Считываем символы в first_name
```

Имя **cin** обращается к стандартному потоку ввода (читается как "си-ин" и является аббревиатурой от **c**haracter **in**put), определенному в стандартной библиотеке. Второй операнд оператора **>>** ("получить из") определяет участок памяти, в который производится ввод. Итак, если мы введем некое имя, например **Nicolas**, а затем нажмем клавишу <Enter> для перехода на новую строку, строка **"Nicolas"** станет значением переменной **first_name**.



Переход на новую строку необходим для того, чтобы привлечь внимание компьютера. Пока переход на новую строку не будет выполнен (не будет нажата клавиша <Enter>), компьютер просто накапливает символы. Эта "отсрочка" дает нам шанс передумать, стереть некоторые символы или заменить их другими перед тем, как нажать

клавишу <Enter>. Символ перехода на новую строку не является частью строки, хранящейся в памяти.

Введя входную строку в переменную `first_name`, ее можно использовать в дальнейшем.

```
cout << "Привет, " << first_name << "!\n";
```

Эта строка выводит на экран слово `Привет`, за которым следует имя `Nicolas` (значение переменной `first_name`) с восклицательным знаком (`!`) и символом перехода на новую строку экрана (`'\n'`).

```
Привет, Nicolas!
```

Если бы мы любили повторяться и набирать лишний текст, то разбили бы эту строку на несколько инструкций.

```
cout << "Привет, ";
cout << first_name;
cout << "!\n";
```

Однако мы не страдаем графоманией и, что еще важнее, очень не любим лишние повторы (поскольку любой повтор создает возможность для ошибки), а потому объединили три оператора вывода в одну инструкцию.

Обратите внимание на то, что мы заключили выражение `"Привет"` в двойные кавычки, но не сделали этого с именем `first_name`. Двойные кавычки используются для работы с литеральными строками. Если двойные кавычки не указаны, то мы ссылаемся на нечто, имеющее имя. Рассмотрим код

```
cout << "first_name" << " - " << first_name;
```

Здесь `"first_name"` представляет собой набор из десяти символов, а просто `first_name` без кавычек позволяет вывести на экран значение переменной `first_name`, в данном случае — `Nicolas`. Таким образом, мы получим

```
first_name - Nicolas
```

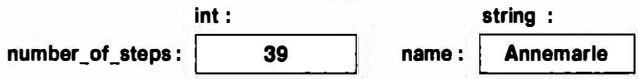
3.2. Переменные

В принципе, не имея возможности хранить данные в памяти так, как это было сделано в предыдущем примере, с помощью компьютера невозможно сделать ничего интересного. "Места", в которых хранятся данные, называют *объектами*. Для доступа к объекту необходимо знать его имя. Именованный объект называется *переменной* и имеет конкретный *тип* (такой, как `int` или `string`), определяющий, какую информацию можно записать в объект (например, в переменную типа `int` можно записать число `123`, а в объект типа `string` — строку символов

"Hello, World!\n"), а также какие операции к нему можно применять (например, переменные типа **int** можно перемножать с помощью оператора ***, а объекты типа **string** можно сравнивать с помощью оператора **<=**). Данные, записанные в переменные, называют *значениями*. Инструкция, определяющая переменную, называется (вполне естественно) *определением*, причем в определении можно (и обычно желательно) задавать начальное значение переменной. Рассмотрим следующий пример:

```
string name = "Annemarie";
int number_of_steps = 39;
```

Эти переменные можно изобразить следующим образом:



Мы не можем записывать в переменную значение неприемлемого типа.

```
string name2 = 39;                  // Ошибка: 39 — не строка
int number_of_steps = "Annemarie"; // Ошибка: "Annemarie" —
                                    // не целое число
```

Компилятор запоминает тип каждой переменной и позволяет использовать переменную лишь так, как предусмотрено ее типом, указанным в определении.

В языке C++ предусмотрен довольно широкий выбор типов (см. раздел A.8). Однако можно создавать прекрасные программы, обходясь лишь пятью из них.

```
int number_of_steps = 39;  // int — для целых чисел
double flying_time  = 3.5; // double — для чисел с плавающей точкой
char decimal_point  = '.'; // char — для символов
string name = "Annemarie"; // string — для строк
bool tap_on = true;        // bool — для логических переменных
```

Ключевое слово **double** используется по историческим причинам: оно является сокращением от выражения "число с плавающей точкой двойной точности" ("double precision floating point.") Число с плавающей точкой представляет собой компьютерное приближение математической концепции действительного числа.

Обратите внимание на то, что каждый из этих типов имеет свой характерный способ записи.

```
39          // int: целое число
3.5         // double: число с плавающей точкой
'.'         // char: отдельный символ в одинарных кавычках
"Annemarie" // string: последовательность символов,
            // ограниченная двойными кавычками
true        // bool: либо истина, либо ложь
```

Иначе говоря, последовательность цифр (например, `1234`, `2` или `976`) означает целое число, отдельный символ в одинарных кавычках (например, `'1'`, `'@'` или `'x'`) означает символ, последовательность цифр с десятичной точкой (например, `1.234`, `0.12` или `.98`) представляет собой число с плавающей точкой, а последовательность символов, заключенных в двойные кавычки (например, `"1234"`, `"Howdy!"` или `"Annemarie"`), обозначает строку. Подробное описание литералов приведено в разделе А.2.

3.3. Ввод и тип

Операция ввода `>>` ("получить из") чувствительна к типу данных, т.е. она считывает информацию в соответствии с типом переменной, в которую производится запись. Рассмотрим пример.

```
// Ввод имени и возраста
int main()
{
    cout << "Пожалуйста, введите свое имя и возраст\n";
    string first_name; // Строковая переменная
    int age;           // Целочисленная переменная
    cin >> first_name; // Считываем строковое значение
    cin >> age;        // Считываем целочисленное значение
    cout << "Привет, " << first_name << " (age " << age << ")\n";
}
```

Таким образом, если вы наберете на клавиатуре `Carlos 22`, то оператор `>>` считает значение `Carlos` в переменную `first_name`, число `22` — в переменную `age` и выведет на экран следующий результат:

```
Привет, Carlos (возраст 22)
```

Почему в переменную `first_name` не была введена вся строка `Carlos 22`? Потому что по умолчанию считывание строки прекращается, как только будет обнаружен так называемый *пробельный символ* (whitespace), т.е. пробел, символ перехода на новую строку или символ табуляции. В других ситуациях по умолчанию разделители оператором `>>` игнорируются. Например, перед считываемым числом можно поместить сколько угодно пробелов; оператор `>>` пропустит их и считает число.

Если вы введете с помощью клавиатуры строку `22 Carlos`, то увидите нечто, что может вас удивить, пока вы хорошенько не подумаете над происшедшим. Число `22` будет считано в переменную `first_name`, поскольку, в конце концов, `22` — это тоже последовательность символов. С другой стороны, строка `Carlos` не является целым числом, поэтому она считана не будет. В результате на экран будет выведено число `22`, за которым будет следовать строковый литерал `"( возраст "` и какое-то случайное число, например `-96739` или `0`. Почему? Потому что вы не указали начальное

значение переменной **age**, а впоследствии в нее ничего не записали. В итоге получили какое-то "мусорное значение", хранившееся в участке памяти в момент запуска программы. В разделе 10.6 мы покажем способ исправления ошибок, связанных с форматом ввода. А пока просто инициализируем переменную **age** так, чтобы она имела предсказуемое значение при ошибке ввода.

```
// Ввод имени и возраста (2-я версия)
int main()
{
    cout << "Пожалуйста, введите свое имя и возраст\n";
    string first_name = "???"; // Строковая переменная ("???"
                               // означает "имя неизвестно")
    int age = -1;              // Целочисленная переменная (-1
                               // означает "возраст неизвестен")
    cin >> first_name >> age;  // Считываем строку, а затем
                               // целое число
    cout << "Привет, " << first_name << " (возраст " << age << ")\n";
}
```

Теперь ввод строки **22 Carlos** приведет к следующему результату:

```
Привет, 22 (возраст -1)
```

Обратите внимание на то, что мы можем одним оператором ввода ввести одновременно несколько значений, как и одним оператором вывода вывести несколько значений на экран. Кроме того, оператор **<<**, как и оператор **>>**, чувствителен к типу, поэтому можно вывести переменную **age** типа **int** вместе со строковой переменной **first_name** и строковыми литералами **"Привет, "**, **" (возраст "** и **"\n"**.

Ввод объекта типа **string** с помощью оператора **>>** прекращается (по умолчанию), когда обнаруживается пробельный символ; иначе говоря, оператор **>>** считывает отдельное слово. Однако иногда нам необходимо прочитать несколько слов. Для этого существует много возможностей. Например, можно прочитать имя, состоящее из двух слов, следующим образом.

```
int main()
{
    cout << "Пожалуйста, введите свое имя и отчество\n";
    string first;
    string second;
    cin >> first >> second; // Считываем две строки
    cout << "Привет, " << first << ' ' << second << '\n';
}
```

Здесь мы просто использовали оператор **>>** дважды, применив его к каждому из слов. Когда мы выводим эти слова на экран, мы вставляем между ними пробел.

ПОПРОБУЙТЕ

Запустите программу "имя и возраст". Измените ее так, чтобы она выводила возраст в месяцах: считайте возраст, выраженный в годах, и умножьте это число на 12 (используя оператор `*`). Считывайте возраст в переменную типа `double`, чтобы дети могли гордиться тем, что им пять с половиной, а не просто пять лет.

3.4. Операции и операторы

Кроме значений, которые могут храниться в переменной, ее тип определяют также операции, которые можно к ней применять, и их смысл. Рассмотрим пример.

```
int count;
cin >> count;                 // >> считывает целое число в count
string name;
cin >> name;                  // >> считывает строку в name
int c2 = count+2;             // + складывает целые числа
string s2 = name + " Jr. ";   // + добавляет символы
int c3 = count-2;             // - вычитает целые числа
string s3 = name - "Jr. ";    // Ошибка: "-" для строк не определен
```

Под ошибкой мы подразумеваем то, что компилятор откажется компилировать программу, пытающуюся вычитать строки. Компилятор точно знает, какие операции можно применять к каждой из переменных, и, следовательно, может предотвратить множество ошибок. Однако компилятор не знает, какие операции имеют смысл для тех или иных значений, поэтому охотно допускает выполнение корректных операций, приводящих к абсурдным результатам, например:

```
int age = -100;
```

Очевидно, что человек не может иметь отрицательный возраст (хотя почему бы и нет?), но никто не сказал компилятору об этом, поэтому он успешно создает код для такого определения.

Вот таблица полезных операторов для наиболее распространенных типов.

	`bool`	`char`	`int`	`double`	`string`
Присваивание	`=`	`=`	`=`	`=`	`=`
Сложение			`+`	`+`	
Конкатенация					`+`
Вычитание			`-`	`-`	
Умножение			`*`	`*`	
Деление			`/`	`/`	
Остаток			`%`		

	bool	char	int	double	string
Увеличение на 1			++	++	
Уменьшение на 1			--	--	
Увеличение на n			+=n	+=n	
Добавить в конец					+=
Уменьшение на n			-=n	-=n	
Умножить и присвоить			*=	*=	
Поделить и присвоить			/=	/=	
Найти остаток и присвоить			%=		
Считать из s в x	s>>x	s>>x	s>>x	s>>x	s>>x
Записать x в s	s<<x	s<<x	s<<x	s<<x	s<<x
Равно	==	==	==	==	==
Не равно	!=	!=	!=	!=	!=
Больше	>	>	>	>	>
Больше или равно	>=	>=	>=	>=	>=
Меньше	<	<	<	<	<
Меньше или равно	<=	<=	<=	<=	<=

Пустые ячейки означают, что данная операция не может применяться к указанному типу непосредственно (хотя существует множество косвенных способов их использования; см. раздел 3.9.1). Со временем мы объясним все эти операции. Главное, что существует множество полезных операций, и их смысл для схожих типов почти одинаков.

Рассмотрим пример, в котором фигурируют числа с плавающей точкой.

```
// Простая программа, демонстрирующая работу операторов
int main()
{
    cout << "Пожалуйста, введите значение с плавающей точкой: ";
    double n;
    cin >> n;
    cout << "n == " << n
         << "\nn+1 == " << n+1
         << "\nтри раза по n == " << 3*n
         << "\nдва раза по n == " << n+n
         << "\nn в квадрате == " << n*n
         << "\nполовина n == " << n/2
         << "\nкорень квадратный из n == " << sqrt(n)
         << '\n';    // Переход на новую строку
}
```

Очевидно, что обычные арифметические операции имеют традиционные обозначения, а их смысл известен нам со школьной скамьи. Естественно также, что не все операции над числами с плавающей точкой реализованы в виде операторов; например, квадратный корень можно извлечь лишь с

помощью функции. Многие операции представлены именованными функциями. В данном случае для извлечения квадратного корня из числа `n` используется функция `sqrt()` из стандартной библиотеки. Система обозначений близка к математической. Более подробно функции рассматриваются в разделах 4.5 и 8.5.

ПОПРОБУЙТЕ

Запустите эту небольшую программу. Затем измените ее так, чтобы считывать значение типа `int`, а не `double`. Обратите внимание на то, что функция `sqrt()` для целых чисел не определена, поэтому присвойте значение `n` переменной типа `double` и лишь затем примените к ней функцию `sqrt()`. Кроме того, выполните несколько других операций. Обратите внимание на то, что операция `/` для целых чисел представляет собой целочисленное деление, а операция `%` — вычисление остатка от деления, так что `5/2` равно `2` (а не `2.5` или `3`), а `5%2` равно `1`. Определения целочисленных операций `*`, `/` и `%` гарантируют, что для двух положительных переменных `a` и `b` типа `int` выполняется равенство `a/b*b + a%b == a`.

Для строк предусмотрено меньше операций, но, как будет показано в главе 23, для них имеется множество специальных функций. Тем не менее к ним можно применять и некоторые обычные операторы, как показано в следующем примере:

```
// Ввод имени и отчества
int main()
{
    cout << "Пожалуйста, введите свое имя и отчество\n";
    string first;
    string second;
    cin >> first >> second;              // Считываем две строки
    string name = first + ' ' + second; // Конкатенация строк
    cout << "Привет, " << name << '\n';
}
```

Для строк оператор `+` означает конкатенацию; иначе говоря, если переменные `s1` и `s2` имеют тип `string`, то `s1+s2` является строкой, в которой вслед за символами строки `s1` следуют символы строки `s2`. Например, если строка `s1` имеет значение `"Hello"`, а строка `s2` — значение `"World"`, то `s1+s2` содержит значение `"HelloWorld"`. Особенно полезным является сравнение строк.

```
// Ввод и сравнение имен
int main()
{
    cout << "Пожалуйста, введите два имени\n";
    string first;
    string second;
```

```
    cin >> first >> second; // считываем две строки
    if (first == second) cout << "имена совпадают\n";
    if (first < second)
        cout << first << " предшествует " << second
             << "в алфавитном порядке" <<'\n';
    if (first > second)
        cout << first << " следует за " << second
             << "в алфавитном порядке"  <<'\n';
}
```

Здесь для выбора действия в зависимости от условия использована инструкция `if`, смысл которой будет подробно изложен в разделе 4.4.1.1.

3.5. Присваивание и инициализация

Одним из наиболее интересных операторов является присваивание, которое обозначается символом =. Этот оператор присваивает переменной новое значение. Рассмотрим пример.

```
int a = 3; // Начальное значение переменной a равно 3
```

a: 3

```
a = 4;      // Переменная a принимает значение 4
            //("становится четверкой")
```

a: 4

```
int b = a; // Начальное значение переменной b является копией
           // значения переменной a (т.е. 4)
```

a: 4

b: 4

```
b = a+5;   // Переменная b принимает значение a+5 (т.е. 9)
```

a: 4

b: 9

```
a = a+7;   // Переменная a принимает значение a+7 (т.е. 11)
```

a: 11

b: 9

Последнее присваивание заслуживает внимания. Во-первых, оно ясно показывает, что знак "равно" не означает равенства, поскольку очевидно, что **a** не равно **a+7**. Этот знак означает присваивание, т.е. помещение в переменную нового значения. Рассмотрим подробнее, что происходит при выполнении инструкции **a=a+7**.

1. Сначала получаем значение переменной **a**; оно равно целому числу 4.
2. Затем добавляем к четверке семерку, получаем целое число 11.
3. В заключение записываем значение 11 в переменную **a**.

Эту операцию можно продемонстрировать также на примере строк.

```
string a = "alpha"; // Начальное значение переменной a равно "alpha"
```

a: alpha

```
a = "beta";         // Переменная a принимает значение "beta"
                    // (становится равной "beta")
```

a: beta

```
string b = a;       // Начальное значение переменной b является
                    // копией значения переменной a (т.е. "beta")
```

a: beta

b: beta

```
b = a+"gamma";      // Переменная b принимает значение a+"gamma"
                    // (т.е. "betagamma")
```

a: beta

b: betagamma

```
a = a+"delta";      // Переменная a принимает значение a+"delta"
                    // (т.е. "betadelta")
```

a: betadelta

b: betagamma

В предыдущих примерах мы использовали выражения "начальное значение" и "принимает значение", для того чтобы отличить похожие, но логически разные операции:

- Инициализация (присваивание переменной ее начального значения).
- Присваивание (запись в переменную нового значения).

Эти операции настолько похожи, что в языке C++ для них используется одно и то же обозначение =.

```
int y = 8;              // Инициализация переменной y значением 8
x = 9;                  // Присваивание значения 9 переменной x

string t = "howdy!"; // Инициализация переменной t
                        // значением "howdy!"
s = "G'day";            // Присваивание переменной s
                       // значения "G'day"
```

Однако с логической точки зрения присваивание и инициализация различаются. Например, инициализация всегда происходит одновременно с указанием типа (например, `int` или `string`), а присваивание — нет. В принципе, инициализация всегда осуществляется с пустой переменной. С другой стороны, присваивание (в общем случае) сначала должно стереть старое значение из переменной и лишь затем записать в нее новое значение. Переменную можно представить в виде небольшого ящика, а значение — в виде конкретной вещи (например, монеты), лежащей в этом ящике. Перед инициализацией ящик пуст, но после нее он всегда содержит монету, поэтому, для того чтобы положить в него новую монету, вы (т.е. оператор присваивания) сначала должны вынуть из него старую ("стереть старое значение"), причем ящик нельзя оставлять пустым. Разумеется, в памяти компьютера эти операции происходят не так буквально, как мы описали, но ничего вредного в такой аллегории нет.

3.5.1. Пример: выявление повторяющихся слов

Присваивание необходимо, когда нам требуется записать в объект новое значение. Если подумать, то станет совершенно ясно, что присваивание является особенно полезным, когда приходится многократно повторять операции. Присваивание необходимо, когда требуется повторить операцию с новым значением. Рассмотрим небольшую программу, выявляющую повторяющиеся слова в предложении. Такие программы являются частью большинства инструментов для проверки грамматики.

```
int main()
{
    string previous = " ";          // Переменная previous;
                                    // инициализирована "не словом"
    string current;                 // Текущее слово
    while (cin>>current) {          // Считываем поток слов
        if (previous == current)    // Проверяем совпадение
                                    // с предыдущим словом
            cout << "Повторяющееся слово: " << current << '\n';
        previous = current;
    }
}
```

Эту программу нельзя назвать очень полезной, поскольку она не способна указать, в каком именно месте стоит повторяющееся слово, но этого для нас пока достаточно. Рассмотрим эту программу строка за строкой.

```
string current;     // Текущее слово
```

Это строковая переменная, в которую мы непосредственно считываем текущее (т.е. последнее прочитанное) слово с помощью кода

```
while (cin>>current)
```

Эта конструкция, называемая инструкцией **while**, интересна сама по себе, и мы еще вернемся к ней в разделе 4.4.2.1. Ключевое слово **while** означает, что инструкция, следующая за выражением (**cin>>current**), будет повторяться до тех пор, пока операция **cin>>current** выполняется успешно, а она будет выполняться успешно до тех пор, пока в стандартном потоке ввода будут считываемые символы. Напомним, что для типа **string** оператор **>>** считывает слова, отделенные друг от друга разделителями. Этот цикл завершается вводом символа конца ввода (как правило, называемым *концом файла*). В системе Windows этот символ вводится путем нажатия комбинации клавиш <Ctrl+Z>, а затем — клавиши <Enter>. В системе Unix или Linux для этого используется комбинация клавиш <Ctrl+D>.

Таким образом, мы считываем слово из потока ввода в переменную **current** и сравниваем его с предыдущим (уже хранящимся в памяти в переменной **previous**). Если они окажутся одинаковыми, мы сообщим об этом.

```
if (previous == current) // Проверяем совпадение с предыдущим словом
    cout << "повторяющееся слово: " << current << '\n';
```

Затем мы должны повторить описанные действия с очередным словом. Для этого скопируем значение переменной **current** в переменную **previous**:

```
previous = current;
```

Эта инструкция учитывает все возможные ситуации, кроме начальной. Что делать с первым словом, у которого нет предыдущего, с которым его следовало бы сравнивать? Эта проблема решается с помощью следующего определения переменной **previous**:

```
string previous = " ";        // Переменная previous;
                              // инициализирована "не словом"
```

Строка " " состоит из одного символа (пробела, который вводится путем нажатия клавиши пробела). Оператор ввода **>>** пропускает разделители, поэтому мы не смогли бы считать этот символ из потока ввода. Следовательно, в ходе первой проверки **while** сравнение

```
if (previous == current)
```

покажет, что значения переменных не совпадают (что и требуется).

Для того чтобы понять программу, надо на время стать "компьютером", т.е. умозрительно выполнять программу строка за строкой. Просто нарисуйте квадратики на бумаге, записывайте в них значения и изменяйте их так, как указано программой.

ПОПРОБУЙТЕ

Выполните рассмотренную выше программу самостоятельно, записывая промежуточные результаты на лист бумаги. Для проверки используйте фразу **The cat cat jumped**. Даже опытные программисты используют этот метод для визуализации относительно неочевидных действий в небольших фрагментах кода.

ПОПРОБУЙТЕ

Запустите разработанную выше программу для выявления повторяющихся слов. Протестируйте ее на предложении **She she laughed He He He because what he did did not look very very good good**. Сколько повторяющихся слов обнаружила ваша программа? Почему? Что означает *слово* в этой программе? А что означает *повторяющееся слово*? Например, **She she** — это повтор или нет?

3.6. Составные операторы присваивания

Операция инкремента переменной (т.е. прибавление к ее значению единицы) настолько часто встречается в программах на языке C++, что для нее предусмотрена отдельная синтаксическая конструкция. Например, выражение

```
++counter
```

означает

```
counter = counter + 1
```

Существует множество способов изменения текущего значения переменной. Например, мы можем захотеть прибавить к нему 7, вычесть 9 или умножить его на 2. Такие операции также непосредственно поддерживаются в языке C++, например:

```
a += 7; // Означает a = a+7
b -= 9; // Означает b = b-9
c *= 2; // Означает c = c*2
```

В целом для любого бинарного оператора **oper** выражение **a oper= b** означает **a = a oper b** (см. раздел A.5). Для начала это правило дает нам операторы **+=**, **-=**, ***=**, **/=** и **%=**. Такая компактная запись позволяет просто и ясно

выражать свои идеи. Например, во многих приложениях операторы ***=** и **/=** означают "масштабирование".

3.6.1. Пример: выявление повторяющихся слов

Вернемся к программе, выявляющей повторяющиеся соседние слова. Ее можно улучшить, если мы сумеем определять место повторяющегося слова. Например, можно просто подсчитывать и выводить на экран количество слов перед повторяющимся.

```
int main()
{
    int number_of_words = 0;
    string previous = " "; // Не слово
    string current;
    while (cin>>current) {
        ++number_of_words; // Увеличиваем счетчик слов
        if (previous == current)
            cout << "Повторение слова " << current
                 << " после " << number_of_words
                 << " слов." << '\n';
        previous = current;
    }
}
```

Счетчик слов инициализируется нулем. Каждый раз, когда мы обнаруживаем слово, мы увеличиваем значение счетчика:

```
++number_of_words;
```

Таким образом, первое слово имеет номер 1, второе — 2 и т.д. Эту операцию можно записать иначе:

```
number_of_words += 1;
```

Или даже так:

```
number_of_words = number_of_words+1;
```

Но выражение **++number_of_words** короче и выражает идею инкремента намного проще.

Обратите внимание, насколько эта программа похожа на пример из раздела 3.5.1. Очевидно, что мы просто взяли программу из раздела 3.5.1 и слегка переделали ее для своих целей. Этот способ очень распространен: если нам нужно решить какую-то задачу, мы ищем похожую и используем готовое решение, внося в него необходимые изменения. Не начинайте разработку программы "с нуля", если есть такая возможность. Использование предыдущего варианта программы в качестве основы для модификации часто экономит много времени и сил.

3.7. Имена

Мы даем своим переменным имена, чтобы запоминать их и обращаться к ним из других частей программы. Каким может быть имя в языке C++? В программе на языке C++ имя начинается с буквы и содержит только буквы, цифры и символ подчеркивания, например

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

Приведенные ниже слова именами не являются:

```
2x              // Имя должно начинаться с буквы
time$to$market  // Символ $ — не буква, не цифра и не подчерк
Start menu      // Пробел — не буква, не цифра и не подчерк
```

Когда мы говорим, что эти последовательности символов не являются именами, то имеем в виду, что компилятор языка C++ не считает их именами.

Если заглянуть в системные коды или коды, сгенерированные машиной, то можно увидеть имена, начинающиеся с символа подчеркивания, такие как `_foo`. Никогда не называйте так свои переменные; такие имена зарезервированы для реализаций языка и системных объектов. Не начиная имена своих переменных с символа подчеркивания, вы избежите конфликтов с системными именами.

Имена чувствительны к регистру; иначе говоря, буквы в нижнем и верхнем регистрах отличаются друг от друга, так что `x` и `X` — это разные имена. Приведем небольшую программу, в которой сделано по крайней мере четыре ошибки.

```
#include "std_lib_facilities.h"
int Main()
{
    STRING s = "Прощай, жестокий мир! ";
    cOut << S << '\n';
}
```

Как правило, использование имен, отличающихся лишь регистром, например `one` и `One`, — плохая идея; хотя это не сможет ввести в заблуждение компилятор, зато легко собьет с толку самого программиста.

ПОПРОБУЙТЕ

Попытайтесь скомпилировать программу "Прощай, жестокий мир!" и просмотрите выведенные компилятором сообщения об ошибках. Смог ли компилятор выявить все ошибки? Какие проблемы обнару-

жил компилятор? Не запутался ли компилятор и не выявил ли он больше четырех ошибок? Поочередно удалите ошибки, начиная с лексически первой, и посмотрите, как будут изменяться (и уточняться) сообщения об ошибках.

В языке C++ зарезервировано около семидесяти ключевых слов. Все они перечислены в разделе A.3.1. Их нельзя использовать в качестве имен переменных, типов, функций и т.п. Рассмотрим пример.

```
int if = 7; // Ошибка: "if" — это ключевое слово
```

В программах можно использовать имена, определенные в стандартных библиотеках (такие, как **string**), но делать этого не следует. Повторное использование распространенных имен может привести к проблемам, если вы воспользуетесь стандартной библиотекой.

```
int string = 7; // Это порождает проблемы
```

Выбирая имена для своих переменных, функций, типов и тому подобного, используйте осмысленные слова; иначе говоря, выбирайте имена, понятные для людей, которые будут читать вашу программу. Даже сам автор может запутаться в тексте своей программы, если станет использовать только простые и короткие имена, такие как **x1**, **x2**, **s3** или **p7**. Аббревиатуры и сокращения могут запутать людей, поэтому использовать их следует как можно реже. Такие сокращения могут быть понятными для вас, но впоследствии вы можете забыть, что значат следующие обозначения:

```
mtbf
TLA
myw
NBV
```

Через несколько месяцев вы забудете, что все это значило. Короткие имена, такие как **x** и **i**, целесообразно использовать в стандартных ситуациях, т.е. переменная **x** должна быть локальной переменной или параметром (см. разделы 4.5 и 8.4), а **i** — счетчиком цикла (см. раздел 4.4.2.3).

Не используйте слишком длинные имена; их трудно набирать, они занимают много места и плохо читаются. Приведем удачные, на наш взгляд, варианты:

```
partial_sum
element_count
stable_partition
```

А вот следующие имена нам кажутся слишком длинными:

```
the_number_of_elements
remaining_free_slots_in_symbol_table
```

Мы предпочитаем использовать в качестве разделителей слов в идентификаторе символы подчеркивания, например `element_count`, и не пользоваться альтернативным именованием `elementCount` или `ElementCount`. Мы никогда не используем имена, состоящие лишь из прописных букв, такие как `ALL_CAPITAL_LETTERS`, поскольку по обычным соглашениям об именовании они резервируются для макросов (см. разделы 27.8 и A.17.2), применения которых мы избегаем. Мы используем в качестве первых букв в именах типов прописные буквы, например `Square` или `Graph`. В языке C++ и его стандартной библиотеке прописные буквы не используются, поэтому типы называются `int` и `string`, а не `Int` и `String`. Таким образом, принятое соглашение позволяет минимизировать вероятность конфликтов имен между пользовательскими и стандартными типами.

Избегайте имен, в которых легко сделать опечатку или ошибиться, или спутать их при чтении, например:

```
Name    names    nameS
foo     f00      fl
f1      fI       fi
```

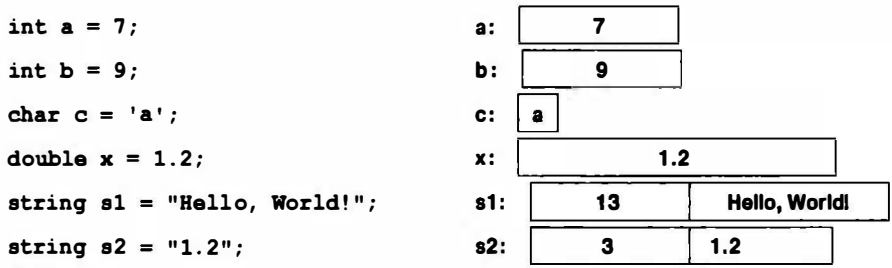
Особенно часто порождают ошибки символы `0` (нуль), `o` (малое O), `O` (большое o), `1` (единица), `l` (малая L), `I` (большая i).

3.8. Типы и объекты

Понятие типа является центральным в языке C++ и большинстве других языков программирования. Рассмотрим типы пристальнее и немного более строго. Особое внимание уделим типам объектов, в которых хранятся данные в процессе вычислений. Все это сэкономит нам время в ходе долгих вычислений и позволит избежать некоторых недоразумений.

- *Тип* определяет набор возможных значений и операций, выполняемых над объектом.
- *Объект* — участок памяти, в котором хранится значение определенного типа.
- *Значение* — набор битов в памяти, интерпретируемый в соответствии с типом.
- *Переменная* — именованный объект.
- *Объявление* — инструкция, приписывающая объекту определенное имя.
- *Определение* — объявление, выделяющее память для объекта.

Неформально объект можно представить в виде ящика, в который можно положить значения определенного типа. В ящике для объектов типа `int` можно хранить только целые числа, например `7`, `42` и `-399`. В ящике для объектов типа `string` можно хранить символьные строки, например `"Interoperability"`, `"tokens: !@#$%^&*"` и `"Old MacDonald had a farm"`. Графически это можно представить так:



Представление объекта типа `string` немного сложнее, чем объекта типа `int`, так как тип `string` хранит количество символов в строке. Обратите внимание на то, что объект типа `double` хранит число, а объект типа `string` — символы. Например, переменная `x` содержит число `1.2`, а переменная `s2` — три символа: `'1'`, `'.'` и `'2'`. Кавычки вокруг символа и строковых литералов в переменных не хранятся.

Все переменные типа `int` имеют один и тот же размер; иначе говоря, для каждой переменной типа `int` компилятор выделяет одинаковое фиксированное количество памяти. В типичном настольном компьютере этот объем равен 4 байтам (32 бита). Аналогично фиксированный размер имеют и объекты типов `bool`, `char` и `double`. В настольном компьютере переменные типа `bool` и `char`, как правило, занимают один байт (8 бит), а переменная типа `double` — 8 байт. Обратите внимание на то, что разные типы объектов занимают разное количество памяти в компьютере. В частности, переменная типа `char` занимает меньше памяти, чем переменная типа `int`, а переменная типа `string` отличается от переменных типов `double`, `int` и `char` тем, что разные строки могут занимать разное количество памяти.

Смысл битов, размещенных в памяти, полностью зависит от типа, используемого для доступа к этим битам. Это следует понимать следующим образом: память компьютера ничего не знает о типах; это просто память и больше ничего. Биты, расположенные в этой памяти, приобретают смысл, только когда мы решаем, как интерпретировать данный участок памяти. Такая ситуация вполне типична при повседневном использовании чисел. Что значит `12.5`? Мы не знаем. Это может быть `12.5 долл.`, `12.5 см` или `12.5 кг`. Только после того, как мы припишем числу `12.5` единицу измерения, оно приобретет конкретный смысл.

Например, один и тот же набор битов в памяти может представлять число `120`, если его интерпретировать как переменную типа `int`, и символ `'x'`, если трактовать его как объект типа `char`. Если взглянуть на него

как на объект типа **string**, то он вообще потеряет смысл, и попытка его использовать приведет к ошибке времени выполнения программы. Эту ситуацию можно проиллюстрировать следующим образом (здесь 1 и 0 означают значения битов в памяти).

00000000 00000000 00000000 01111000

Этот набор битов, записанных в участке памяти (слове), можно прочитать как переменную типа **int** (120) или **char** (**'x'**, если учитывать только младшие 8 битов). **Бит** — это единица памяти компьютера, которая может хранить только либо 0, либо 1. Смысл *двоичных* (*бинарных*) чисел описан в разделе A.2.1.1.

3.9. Безопасность типов

Каждый объект при определении получает тип. Программа — или часть программы — является безопасной с точки зрения использования типов (type-safe), если объекты используются только в соответствии с правилами, предусмотренными для их типов. К сожалению, существуют операции,

которые не являются безопасными с этой точки зрения. Например, использование переменной до ее инициализации является небезопасным.

```
int main()
{
    double x;          // Переменная x не инициализирована,
                       // ее значение не определено
    double y = x;      // Значение переменной y не определено
    double z = 2.0+x; // Смысл операции + и значение переменной z
                       // не определены
}
```

При попытке использовать неинициализированную переменную **x** реализация может даже выдать ошибку аппаратного обеспечения. Всегда инициализируйте свои переменные! У этого правила есть лишь несколько — очень мало — исключений (например, если переменная немедленно используется для ввода данных), но инициализация переменных в любом случае — это хорошая привычка, предотвращающая множество неприятностей.

Полная безопасность типов является идеалом и, следовательно, общим правилом для языка. К сожалению, компилятор языка C++ не может гарантировать полную безопасность типов, но мы можем избежать ее нарушений, используя хороший стиль программирования и проверки времени выполнения. Идеально было бы вообще никогда не использовать свойства языка, безопасность которых компилятор не в состоянии доказать.

Такая безопасность типов называется *статической*. К сожалению, это сильно ограничило бы наиболее интересные сферы применения программирования. Очевидно, если бы компилятор неявно генерировал код, проверяющий нарушения безопасность типов, и перехватывал их все, то это замедляло бы программу и выходило бы за рамки языка C++. Если мы принимаем решения использовать приемы, не являющиеся безопасными с точки зрения использования типов, то должны проверять себя сами и самостоятельно обнаруживать такие ситуации.

Идеал безопасности типов невероятно важен при написании кода. Вот почему мы поминаем о нем так рано. Пожалуйста, помните об этой опасности и старайтесь избегать ее в своих программах.

3.9.1. Безопасные преобразования

В разделе 3.4 мы видели, что нельзя непосредственно складывать объекты типа `char` или сравнивать объекты типов `double` и `int`. Однако в языке C++ это можно сделать косвенным образом. При необходимости объект типа `char` можно преобразовать в объект типа `int`, а объект типа `int` — в объект типа `double`. Рассмотрим пример.

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

Здесь значения переменных `i1` и `i2` равны `120`, т.е. 8-битовому ASCII-коду символа `'x'`. Это простой и безопасный способ получения числового представления символа. Мы называем такое преобразование типа `char` в тип `int` безопасным, поскольку при этом не происходит потери информации; иначе говоря, мы можем скопировать результат, хранящийся в переменной типа `int`, обратно в переменную типа `char` и получить исходное значение.

```
char c2 = i1;
cout << c << ' ' << i1 << ' ' << c2 << '\n';
```

Этот фрагмент программы выводит на экран следующий результат:

```
x 120 x
```

В этом смысле — что значение всегда преобразуется в эквивалентное значение или (для типа `double`) в наилучшее приближение эквивалентного значения — следующие преобразования являются безопасными.

- `bool` в `char`
- `bool` в `int`
- `bool` в `double`
- `char` в `int`
- `char` в `double`
- `int` в `double`

Наиболее полезным является преобразование переменной типа **int** в переменную типа **double**, поскольку позволяет использовать комбинацию этих типов в одном выражении.

```
double d1 = 2.3;
double d2 = d1+2; // Перед сложением 2 преобразуется в 2.0
if (d1 < 0)       // Перед сравнением 0 преобразуется в 0.0
    cout << "d1 — отрицательно";
```

Для действительно больших чисел типа **int** при их преобразовании в переменные типа **double** мы можем (в некоторых компьютерах) потерять точность. Однако эта проблема возникает редко.

3.9.2. Небезопасные преобразования

Безопасные преобразования обычно не беспокоят программистов и упрощают разработку программ. К сожалению, язык C++ допускает также (неявные) опасные преобразования. Под опасными преобразованиями мы подразумеваем то, что значение может неявно превратиться в значение иного типа, которое не равно исходному. Рассмотрим пример.

```
int main()
{
    int a = 20000;
    char c = a; // Попытка втиснуть большое значение типа int
                // в маленькую переменную типа char
    int b = c;
    if (a != b) // != означает "не равно"
       cout << "Ой!: " << a << "!=" << b << '\n';
    else
       cout << "Ого! Однако у нас очень большой char\n";
}
```

Такие преобразования называют "сужающими", поскольку они помещают значение в объект, размер которого слишком мал ("узок") для его хранения. К сожалению, лишь некоторые компиляторы предупреждают об опасной инициализации переменной типа **char** значением переменной типа **int**. Проблема заключается в том, что тип **int**, как правило, намного больше типа **char**, так что он может (в нашем случае так и происходит) хранить значение типа **int**, которое невозможно представить значением типа **char**. Попробуйте поэкспериментировать с приведенной далее программой.

```
int main()
{
    double d = 0;
    while (cin>>d) { // Повторяем последующие инструкции,
                     // пока вводятся числа
    int i = d;       // Попытка втиснуть double в int
```

```
    char c = i;       // Попытка втиснуть int в char
    int i2 = c;       // Получаем целое значение типа char
    cout << " d==" << d        // Исходное значение типа double
         << " i==" << i        // Преобразованное в значение int
         << " i2==" << i2      // Целочисленное значение char
         << " char(" << c << ")\n"; // Символ
    }
}
```

Использованная в этой программе инструкция **while**, позволяющая ввести много значений, рассматривается в разделе 4.4.2.1.

ПОПРОБУЙТЕ

Выполните эту программу, вводя разные значения. Попробуйте ввести небольшие значения (например, **2** и **3**); большие значения (больше, чем **127**; больше, чем **1000**); отрицательные значения; испытайте числа **56**, **89**, **128**; нецелые числа (например, **56.9** и **56.2**). Кроме результатов преобразования типа **double** в тип **int** и типа **int** в тип **char** на вашем компьютере, эта программа показывает, какой символ (если таковой имеется) выводится для заданного целого числа.

Вы обнаружите, что многие числа приводят к "бессмысленным" результатам. Образно говоря, это происходит, когда вы пытаетесь перелить жидкость из трехлитровой банки в пол-литровую. Компилятор разрешает все перечисленные ниже преобразования, несмотря на их опасность.

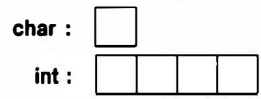
double в **int**
double в **char**
double в **bool**
int в **char**
int в **bool**
char в **bool**

Эти преобразования являются опасными в том смысле, что значение, хранящееся в переменной, может отличаться от присвоенного. Почему эта ситуация считается проблемой? Потому что зачастую вы и не подозреваете о том, что такое преобразование имело место. Рассмотрим пример.

```
double x = 2.7;
// Какой-то код
int y = x; // Значение переменной y становится равным 2
```

К моменту определения переменной **y** вы уже могли забыть, что переменная **x** имеет тип **double**, или упустить из виду, что преобразование **double** в **int** приводит к усечению (округлению по направлению к нулю). Результат вполне предсказуем: семь десятых потеряны.

Преобразование **int** в **char** не порождает проблем с усечением — ни тип **int**, ни тип **char** не могут представить дробную часть целого числа. Однако переменная типа **char** может хранить только очень небольшие целочисленные значения. В персональных компьютерах переменная типа **char** занимает 1 байт, в то время как переменная типа **int** — 4 байта.



Таким образом, мы не можем записать большое число, например 1000, в переменную типа **char** без потери информации: значение "сужается". Рассмотрим пример.

```
int a = 1000;
char b = a; // Значение b становится равным -24
```

Не все значения типа **int** имеют эквиваленты типа **char**, а точный диапазон значений типа **char** зависит от конкретной реализации. На персональных компьютерах значения типа **char** колеблются в диапазоне [-128,127], но мобильность программ можно обеспечить только в диапазоне [0,127], поскольку не каждый компьютер представляет собой ПК, и у разных компьютеров разные диапазоны значений **char**, например такие как [0,255].

Почему люди смирились с проблемой сужающих преобразований? Основная причина носит исторический характер: язык C++ унаследовал сужающие преобразования от предшественника, языка C. К первому дню существования языка C++ уже было множество программ, написанных на языке C и содержащих сужающие преобразования. Кроме того, многие такие преобразования на самом деле не создают никаких проблем, поскольку используемые значения не выходят за пределы допустимых диапазонов, и многие программисты жалуются, что "компиляторы указывают им, что надо делать". В частности, опытные программисты легко справляются с проблемой опасных преобразований в небольших программах, хотя в более крупных программах и в неопытных руках такие преобразования могут стать источником ошибок. Однако компиляторы могут предупреждать программистов о сужающих преобразованиях — и многие из них делают это.

В C++11 вводится запись инициализации, которая запрещает сужающие преобразования. Например, мы можем (и должны) переписать приведенные выше проблемные примеры с использованием записи со списками в фигурных скобках вместо записи со знаком присваивания:

```
double x {2.7}; // OK
int y {x};      // Ошибка: double -> int может быть сужающим
int a {1000};   // OK
char b {a};     // Ошибка: int -> char может быть сужающим
```

Когда инициализатор представляет собой целочисленный литерал, компилятор в состоянии проверить его фактическое значение и принять значения, не вызывающие сужения:

```
char b1 {1000}; // Ошибка: сужение (в предположении
                // 8-битового типа char)
char b2 {48};   // OK
```

Так что же делать, если вы думаете, что преобразование может привести к неверным результатам? Воспользуйтесь инициализатором `{}` во избежание инцидентов, а когда хотите выполнить преобразование, перед присваиванием выполните проверку, подобную сделанной в первом примере этого раздела. Упрощенный вариант проверки описан в разделах 5.6.4 и 7.5. Запись с использованием списка в фигурных скобках известна как *универсальная унифицированная инициализация* (universal and uniform initialization), и позже мы детально познакомимся с ней.

Задание

На каждом этапе выполнения задания запускайте программу и убеждайтесь, что она делает именно то, что вы ожидали. Составьте список сделанных ошибок, чтобы вы могли избегать их в будущем.

1. Напишите программу, формирующую простую форму для письма на основе входной информации. Для начала наберите программу из раздела 3.1, предложив пользователю ввести свое имя и предусмотрев вывод строки "Привет, **first_name**", где **first_name** — это имя, введенное пользователем. Затем модифицируйте программу следующим образом: измените приглашение на строку "Введите имя адресата" и измените вывод на строку "Дорогой **first_name**," (не забудьте о запятой!).
2. Введите одну или две вступительные фразы, например "Как дела? У меня все хорошо. Я скучаю по тебе". Убедитесь, что первая строка отделена от других. Добавьте еще несколько строк по своему усмотрению — это же ваше письмо.
3. Предложите пользователю ввести имя другого приятеля и сохраните его в переменной **friend_name**. Добавьте в ваше письмо следующую строку: "Давно ли ты встречал **friend_name**?".
4. Объявите переменную типа **char** с именем **friend_sex** и инициализируйте ее нулем. Предложите пользователю ввести значение **m**, если ваш друг — мужчина, и **f** — если женщина. Присвойте переменной **friend_sex** введенное значение. Затем с помощью двух инструкций **if** запишите следующее.

Если друг — мужчина, то напишите строку "Если ты увидишь `friend_name`, пожалуйста, попроси его позвонить мне".

Если друг — женщина, то напишите строку "Если ты увидишь `friend_name`, пожалуйста, попроси ее позвонить мне".

5. Предложите пользователю ввести возраст адресата и присвойте его переменной `age`, имеющей тип `int`. Ваша программа должна вывести на экран строку "Я слышал, ты только что отметил день рождения и тебе исполнилось `age` лет". Если значение переменной `age` меньше или равно 0 или больше или равно 110, выполните вызов `simple_error("ты шутишь!")`, используя функцию `simple_error()` из заголовочного файла `std_lib_facilities.h`.
6. Добавьте в ваше письмо следующее.

   Если вашему другу меньше 12 лет, напишите "На следующий год тебе исполнится `age+1` лет".

   Если вашему другу 17 лет, напишите "В следующем году ты сможешь голосовать".

   Если вашему другу больше 70 лет, напишите "Я надеюсь, что ты не скучаешь на пенсии".

   Убедитесь, что ваша программа правильно обрабатывает каждое из этих значений.
7. Добавьте строку "Искренне твой", затем введите две пустые строки для подписи и укажите свое имя.

Контрольные вопросы

1. Что подразумевается под *приглашением*?
2. Какой оператор используется для чтения в переменную?
3. Если вы хотите, чтобы пользователь ввел целое значение в переменную с именем `number`, какие две строки следует добавить в программу, чтобы предложить пользователю ввести значение в вашу программу и считать введенное им значение?
4. Как называется символ `\n` и для чего он предназначен?
5. Что прекращает ввод значения в строку?
6. Что прекращает ввод значения в целочисленную переменную?
7. Как записать одной строкой следующий код?

```
cout << "Привет, ";
cout << first_name;
cout << "!\n";
```

8. Что такое объект?

9. Что такое литерал?
10. Какие виды литералов существуют?
11. Что такое переменная?
12. Каковы типичные размеры переменных типов `char`, `int` и `double`?
13. В каких единицах измеряется объем памяти, занимаемой небольшими переменными, например объектами типов `int` и `string`?
14. В чем заключается разница между операторами = и ==?
15. Что такое определение?
16. Что такое инициализация и чем она отличается от присваивания?
17. Что такое конкатенация строк и как она выполняется в языке C++?
18. Какие из приведенных далее имен являются допустимыми в языке C++? Если имя является недопустимым, укажите, по какой причине.

```
This_1ittle_pig      This_1_is fine      2_For_1_special
latest thing         the_$12_method      _this_is_ok
MiniMineMine         number              correct?
```

19. Приведите пять примеров корректных имен, которые вы не стали бы использовать, чтобы не создавать недоразумений.
20. Сформулируйте разумные правила для выбора имен.
21. Что такое безопасность типов и почему она так важна?
22. Почему преобразование типа `double` в тип `int` может привести к неприятностям?
23. Сформулируйте правило, помогающее решить, безопасно ли то или иное преобразование типов.

Термины

`cin`
безопасность типов
декремент
значение
имя
инициализация
инкремент
конкатенация
объект
объявление
оператор
операция
определение
переменная
преобразование
присваивание
сужение
тип

Упражнения

1. Выполните задание из раздела **Попробуйте**, если вы не сделали его раньше.
2. Напишите на языке C++ программу, которая преобразует мили в километры. Ваша программа должна содержать понятное приглашение пользователю ввести количество миль. *Указание*: в одной миле 1.609 км.

3. Напишите программу, которая ничего не делает, а просто объявляет ряд переменных с допустимыми и недопустимыми именами (например, `int double = 0;`), и посмотрите на реакцию на них компилятора.
4. Напишите программу, предлагающую пользователю ввести два целочисленных значения. Запишите эти значения в переменные типа `int` с именами `val1` и `val2`. Напишите программу, определяющую наименьшее и наибольшее значения, а также сумму, разность, произведение и частное этих значений.
5. Измените программу так, чтобы она просила пользователя ввести два числа с плавающей точкой и сохраняла их в переменных типа `double`. Сравните результаты работы этих двух программ для некоторых входных данных на ваш выбор. Совпадают ли эти результаты? Должны ли они совпадать? Чем они различаются?
6. Напишите программу, предлагающую пользователю ввести три целых числа, а затем выводит их в порядке возрастания, разделяя запятыми. Например, если пользователь вводит числа **10 4 6**, то программа должна вывести на экран числа **4, 6, 10**. Если два числа совпадают, то они должны идти одно за другим. Например, если пользователь вводит числа **4 5 4**, то программа должна вывести на экран **4, 4, 5**.
7. Выполните упр. 6 для трех строковых значений. Так, если пользователь вводит значения **Steinbeck**, **Hemingway**, **Fitzgerald**, то вывод программы должен имеет вид **Fitzgerald, Hemingway, Steinbeck**.
8. Напишите программу, проверяющую четность или нечетность целых чисел. Как всегда, убедитесь, что вывод программы ясен и полон. Иначе говоря, не следует ограничиваться простой констатацией вроде "**да**" или "**нет**". Вывод должен быть информативным, например "**Число 4 является четным**". Подсказка: см. оператор вычисления остатка в разделе 3.4.
9. Напишите программу, преобразующую слова "нуль", "два" и так далее в цифры 0, 2 и т.д. Когда пользователь вводит число в виде слова, программа должна вывести на экран соответствующую цифру. Выполните эту программу для цифр 0, 1, 2, 3 и 4. Если пользователь введет что-нибудь другое, например фразу "**глупый компьютер!**", программа должна ответить "**Я не знаю такого числа!**"
10. Напишите программу, принимающую на входе символ операции с двумя операндами и выводящую на экран результат вычисления. Например:

```
+ 100 3.14
* 4 5
```

Считайте символ операции в строку с именем `operation` и, используя инструкцию `if`, выясните, какую операцию хочет выполнить пользователь, например `if (operation=="+")`. Считывайте операнды в переменные типа `double`. Выполните операции `+`, `-`, `*`, `/`, `plus`, `minus`, `mul` и `div`, имеющие очевидный смысл.

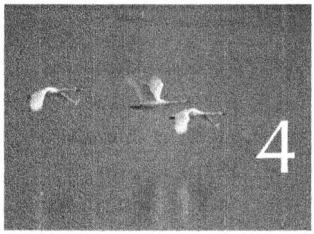
11. Напишите программу, предлагающую пользователю ввести определенное количество 1-, 5-, 10-, 25-, 50-центовых и долларовых монет. Пользователь должен по отдельности ввести количество монет каждого достоинства, например **"Сколько у вас одноцентовых монет?"** Затем программа должна вывести результат примерно такого вида.

```
У вас 23 одноцентовых монет.
У вас 17 пятицентовых монет.
У вас 14 десятицентовых монет.
У вас 7 двадцатипятицентовых монет.
У вас 3 пятидесятицентовых монет.
Общая стоимость ваших монет равна 573 центам.
```

Усовершенствуйте программу: если у пользователя только одна монета, выведите ответ в грамматически правильной форме. Например, **"14 десятицентовых монет"** и **"1 одноцентовая монета"** (а не **"1 одноцентовых монет"**). Кроме того, выведите результат в долларах и центах, т.е. **5 долларов 73 цента**, а не **573 цента.**

Послесловие

Не следует недооценивать важность безопасности типов. Типы — центральное понятие при создании правильных программ, и некоторые из наиболее эффективных методов разработки программ основаны на разработке и использовании типов (см. главы 6 и 9 и части II–IV).



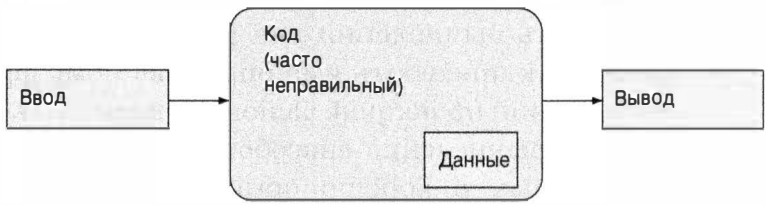
4 Вычисления

Если результат не обязан быть точным, я могу вычислить его сколь угодно быстро.
— Джеральд Вайнберг (Gerald M. Weinberg)

В главе излагаются основы вычислений. В частности, объясняется, как вычислять значения с помощью набора операндов (*выражения*), как выбирать действия среди возможных альтернатив (*выбор*), как повторять вычисления для ряда значений (*итерации*) и как присвоить имя определенному фрагменту вычислений (*функции*). Основная цель главы — представить вычисления способом, ведущим к созданию правильных и хорошо организованных программ. Чтобы помочь вам выполнять более реалистичные вычисления, мы вводим тип **vector**, предназначенный для хранения последовательностей значений.

4.1. Вычисления

Все программы что-нибудь вычисляют; иначе говоря, они получают на вход какие-то данные и выводят какие-то результаты. В конце концов, само устройство, на котором выполняются программы, называется компьютером (от английского слова *compute* — "вычислять". — *Примеч. ред.*) Эта точка зрения является правильной и обоснованной, пока мы придерживаемся широкой трактовки ввода и вывода.



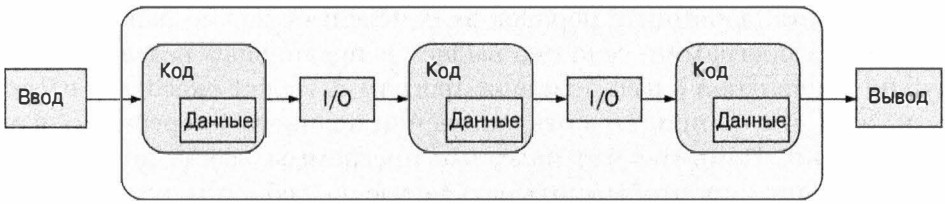
Входная информация может поступать с клавиатуры, от мыши, с сенсорного экрана, из файлов, от других устройств ввода и других частей программы. К категории "другие устройства ввода" относятся большинство интересных источников данных: музыкальные клавишные пульты, устройства видеозаписи, датчики температуры, сенсоры цифровых видеокамер и т.п. Разнообразие этих устройств бесконечно.

Для обработки входной информации программы обычно используют специальные данные, которые называют *структурами данных* (data structures) или *состояниями* (states). Например, программа, имитирующая календарь, может содержать списки праздничных дней в разных странах и список ваших деловых встреч. Одни из этих данных являются частью программы изначально, а другие появляются, когда программа считывает входные данные и извлекает из них полезную информацию. Например, программа-календарь может создавать список ваших деловых встреч по мере их ввода в нее. В этом случае основной входной информацией являются месяцы и дни встреч (возможно, вводимые с помощью щелчка мышью) и данные о самих встречах (вероятно, вводимые с помощью клавиатуры).

Устройством вывода этой программы является экран, на котором высвечиваются календарь и данные о назначенных встречах, а также кнопки и приглашения для ввода, которые программа может выводить на экран.

Входная информация может поступать от самых разных источников. Аналогично результаты могут выводиться на самые разные устройства: на экран, в файл, в сетевое подключение, в иные устройства вывода, в другие программы или части программы. К примерам устройств вывода относятся также сетевые интерфейсы, музыкальные синтезаторы, электрические моторы, лампочки и светодиоды, обогреватели и т.п.

С программистской точки зрения наиболее важными и интересными категориями ввода-вывода являются "в другую программу" и "в другие части программы". Большая часть настоящей книги посвящена последней категории: как представить программу в виде взаимодействующих частей и как обеспечить совместный доступ к данным и обмен информацией. Это ключевые вопросы программирования. Проиллюстрируем их графически.



Аббревиатура "I/O" означает "ввод-вывод". В данном случае вывод из одной части программы является вводом в следующую часть. Эти части программы имеют доступ к данным, хранящимся в основной памяти, на постоянном устройстве хранения данных (например, на диске) или передающимся через сетевые соединения. Под частями программы мы подразумеваем сущности, такие как функция, вычисляющая результат на основе полученных аргументов (например, извлекающая корень квадратный из числа с плавающей точкой), функция, выполняющая действия над физическими объектами (например, рисующая линию на экране), или функция, модифицирующая некую таблицу в программе (например, добавляющая имя в таблицу клиентов).

Когда мы говорим "ввод" и "вывод", то обычно подразумеваем, что в компьютер вводится или из компьютера выводится некая информация, но, как вы вскоре увидите, мы можем использовать эти термины и для информации, переданной другой части программы или полученной от нее. Информацию, которая является вводом в часть программы, часто называют *аргументом*, а данные, поступающие от части программы, — *результатом*.

Вычислением мы называем некое действие, создающее определенные результаты и основанное на определенных входных данных, например порождение результата (вывода), равного `49`, на основе аргумента (ввода),

равного 7, с помощью вычисления (функции) возведения в квадрат `square` (см. раздел 4.5). Как курьезный факт напомним, что до 1950-х годов компьютером (вычислителем) в США назывался человек, выполнявший вычисления, например бухгалтер, навигатор или физик. В настоящее время мы просто перепоручили большинство вычислений компьютерам (машинам), среди которых простейшими являются карманные калькуляторы.

4.2. Цели и средства

Цель программиста — выразить вычисления, причем это должно быть сделано

- правильно;
- просто;
- эффективно.

Пожалуйста, запомните порядок этих целей: не важно, как быстро работает ваша программа, если она выдает неправильные результаты. Аналогично правильная и эффективная программа может оказаться настолько сложной, что ее придется отклонить или полностью переписать в виде новой версии. Помните, что полезные программы всегда должны допускать модификации, чтобы учитывать новые потребности, новые аппаратные устройства и т.д. Для этого программа — и любая ее часть — должна быть как можно более простой. Например, предположим, что вы написали идеальную программу для обучения основам арифметики детей в вашей местной школе, но ее внутренняя структура является слишком запутанной. На каком языке вы собираетесь общаться с детьми? На английском? Английском и испанском? А не хотели бы вы, чтобы вашу программу использовали в Финляндии? А в Кувейте? Как изменить естественный язык, используемый для общения с детьми? Если программа имеет слишком запутанную структуру, то логически простая (но на практике почти всегда очень сложная) операция изменения естественного языка для общения с пользователями становится непреодолимой.

Забота о правильности, простоте и эффективности программ возлагается на нас с той минуты, когда мы начинаем писать программы для других и осознаем ответственность за качество своей работы; иначе говоря, решив стать профессионалами, мы обязаны создавать хорошие программы. С практической точки зрения это значит, что мы не можем просто нагромождать инструкции, пока программа не заработает; мы должны разработать определенную структуру программы. Парадоксально, но забота о структуре и качестве кода часто является самым быстрым способом разработки работоспособных программ. Если программирование выполнено качественно, то хорошая структура программы

позволяет сэкономить время на самой неприятной части работы: отладке. Иначе говоря, хорошая структура программы, продуманная на этапе разработки, может минимизировать количество сделанных ошибок и уменьшить объем времени, затрачиваемого на поиск таких ошибок и их исправление.

Наша главная цель при организации программы — и организации наших мыслей, возникающих в ходе работы над программой, — разбить большой объем вычислений на множество небольших фрагментов. Существуют два варианта этого метода.

- *Абстракция*. Этот способ предполагает сокрытие деталей, которые не являются необходимыми для работы с программой (детали реализации) за удобным и универсальным интерфейсом. Например, вместо изучения деталей сортировки телефонной книги (о методах сортировки написано множество толстых книг) мы просто вызываем алгоритм `sort` из стандартной библиотеки языка C++. Все, что нам нужно для сортировки, — знать, как вызывается этот алгоритм, так что мы можем написать инструкцию `sort(b)`, где `b` — телефонная книга; `sort()` представляет собой вариант алгоритма `sort` (разделы 21.8 и Б.5.4), определенного в `std_library.h`. Другой пример связан с использованием памяти компьютера. Непосредственное использование памяти может быть довольно сложным, поэтому чаще к участкам памяти обращаются через переменные, имеющие тип и имя (раздел 3.2), объекты класса `vector` из стандартной библиотеки (раздел 4.6, главы 17–19), объекты класса `map` (глава 21) и т.п.
- "*Разделяй и властвуй*". Этот способ подразумевает разделение большой задачи на несколько меньших задач. Например, если требуется создать словарь, то работу можно разделить на три части: чтение, сортировка и вывод данных. Каждая из новых задач намного меньше исходной.

Чем это может помочь? Ведь в конечном итоге программа, созданная из частей, обычно немного больше, чем программа, в которой все фрагменты оптимально соединены друг с другом. Причина заключается в том, что мы плохо справляемся в большими задачами. Как правило, как в программировании, так и в жизни, мы разбиваем их на меньшие части, полученные части разделяем на еще более мелкие, пока не получим достаточно простую задачу, которую легко понять и решить. Возвращаясь к программированию, легко понять, что количество ошибок в программе, состоящей из

1000 строк, гораздо более чем в 10 раз превосходит количество ошибок в программе, состоящей из 100 строк, поэтому имеет смысл разделить большую программу на части, размер которых меньше 100 строк. Для более крупных программ, скажем, длиной более 10 миллионов строк, применение абстракции и метода "разделяй и властвуй"

является даже не пожеланием, а настоятельным требованием. Мы просто не в состоянии писать и поддерживать работу крупных монолитных программ. Оставшуюся часть книги можно рассматривать как длинный ряд примеров задач, которые необходимо разбить на более мелкие части, а также методов и способов, используемых для этого.

Рассматривая процесс разбиения программ, мы всегда учитываем, какие инструменты помогают выделить эти части и обеспечить взаимодействие между ними. Хорошая библиотека, содержащая полезные средства для выражения идей, может существенно повлиять на распределение функциональных свойств между разными частями программы. Мы не можем просто сидеть и фантазировать, как лучше разбить программу на части; мы должны учитывать, какие библиотеки находятся в нашем распоряжении и как их можно использовать. Пока вы находитесь в начале пути, но вскоре увидите, что использование существующих библиотек, таких как стандартная библиотека языка C++, позволяет сэкономить много сил не только на этапе программирования, но и на этапах тестирования и документации. Например, потоки ввода-вывода позволяют нам не вникать в детали устройства аппаратных портов ввода-вывода. Это первый пример разделения программы на части с помощью абстракции. В следующих главах мы приведем новые примеры.

Обратите внимание на то, какое значение мы придаем структуре и организации программы: вы не сможете написать хорошую программу, просто перечислив множество инструкций. Почему мы упоминаем об этом сейчас? На текущем этапе вы (или, по крайней мере, многие читатели) слабо представляете себе, что такое программа, и лишь через несколько месяцев будете готовы написать программу, от которой может зависеть жизнь или благосостояние других людей. Мы упоминаем об этом, чтобы помочь вам правильно спланировать свое обучение. Существует большой соблазн набросать примерный план курса по программированию — похожего на изложенный в оставшейся части книги, — выделив темы, которые имеют очевидное полезное применение и проигнорировав "тонкие", более концептуальные вопросы разработки программного обеспечения. Однако хорошие программисты и проектировщики систем знают (и это знание часто приобретается тяжелой ценой), что вопросы структуры лежат в основе хорошего программного обеспечения и пренебрежение ими порождает массу проблем. Не обеспечив хорошей структуры программы, вы, образно говоря, лепите ее из глины. Это вполне возможно, но таким образом никогда нельзя построить пятиэтажный дом (глина просто не выдержит). Если хотите построить не времянку, а солидное здание, то следует уделить внимание структуре и правильной организации кода, а не возвращаться к этим вопросам, совершив множество ошибок.

4.3. Выражения

Основными строительными конструкциями программ являются выражения. Выражение вычисляет определенное значение на основе некоторого количества операндов. Простейшее выражение представляет собой обычную литеральную константу, например `'a'`, `3.14` или `"Norah"`.

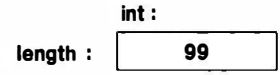
Имена переменных также являются выражениями. Переменная — это объект, имеющий имя. Рассмотрим пример.

```
// Вычисление площади:
int length = 20;          // Литеральное целое значение (исполь-
                          // зуется для инициализации переменной)
int width = 40;
int area = length*width; // Умножение
```

Здесь литералы `20` и `40` используются для инициализации переменных `length` и `width`, соответствующих длине и ширине. После этого длина и ширина перемножаются; иначе говоря, мы перемножаем значения `length` и `width`. Здесь выражение "значение `length`" представляет собой сокращение выражения "значение, хранящееся в объекте с именем `length`". Рассмотрим еще один пример.

```
length = 99;              // Присваиваем length значение 99
```

Здесь слово `length`, обозначающее левый операнд оператора присваивания, означает "объект с именем `length`", поэтому это выражение читается так: "записать число `99` в объект с именем `length`". Следует различать имя `length`, стоящее в левой части оператора присваивания или инициализации (оно называется "lvalue переменной `length`" или "объект с именем `length`") и в правой части этих операторов (в этом случае оно называется "rvalue переменной `length`", "значение объекта с именем `length`" или просто "значение `length`"). В этом контексте полезно представить переменную в виде ящика, помеченного именем.



Иначе говоря, `length` — это имя объекта типа `int`, содержащего значение `99`. Иногда (в качестве lvalue) имя `length` относится к ящику (объекту), а иногда (в качестве rvalue) — к самому значению, хранящемуся в этом ящике.

Комбинируя выражения с помощью операторов, таких как `+` и `*`, мы можем создавать более сложные выражения, так, как показано ниже. При необходимости для группировки выражения можно использовать скобки.

```
int perimeter = (length+width)*2; // Сложить и умножить
```

Без скобок это выражение пришлось бы записать следующим образом:

```
int perimeter = length*2+width*2;
```

что слишком громоздко и провоцирует ошибки наподобие следующей:

```
int perimeter = length+width*2;   // Сложить width*2 с length
```

Последняя ошибка является логической, и компилятор не может ее обнаружить. Компилятор просто видит переменную с именем **perimeter**, инициализированную корректным выражением. Если результат выражения не имеет смысла, то это ваши проблемы. Вы знаете математическое определение периметра, а компилятор — нет.

В программах применяются обычные математические правила, регламентирующие порядок выполнения операторов, поэтому **length+width*2** означает **length+(width*2)**. Аналогично выражение **a*b+c/d** означает **(a*b)+(c/d)**, а не **a*(b+c)/d**. Таблица приоритетов операторов приведена в разделе А.5.

Первое правило использования скобок гласит: "Если сомневаешься, используй скобки". И все же программист должен научиться правильно формировать выражения, чтобы не сомневаться в значении формулы **a*b+c/d**. Слишком частое использование скобок, например **(a*b)+(c/d)**, снижает удобочитаемость программы.

Почему мы заботимся об удобочитаемости? Потому что ваш код будете читать не только вы, но и, возможно, другие программисты, а запутанный код замедляет чтение и препятствует его анализу. Неуклюжий код не просто сложно читать, но и трудно исправлять. Плохо написанный код часто скрывает логические ошибки. Чем больше усилий требуется при его чтении, тем сложнее убедить себя и других, что он является правильным. Не пишите слишком сложных выражений вроде

```
a*b+c/d*(e-f/g)/h+7 // Слишком сложно
```

и всегда старайтесь выбирать осмысленные имена.

4.3.1. Константные выражения

В программах, как правило, используется множество констант. Например, в программе для геометрических вычислений может использоваться число "пи", а в программе для пересчета дюймов в сантиметры — множитель **2.54**. Очевидно, что этим константам следует назначать осмысленные имена (например, **pi**, а не **3.14159**). Аналогично константы не должны изменяться случайным образом. По этой причине в языке C++ предусмотрено понятие символической константы, т.е. именованного объекта, которому после его инициализации невозможно присвоить новое значение. Рассмотрим пример.

```
constexpr double pi = 3.14159;
pi = 7;          // Ошибка: присваивание значения константе
int v = 2*pi/r; // ОК: мы читаем, но не изменяем значение pi
```

Такие константы полезны для повышения удобочитаемости программ. Увидев фрагмент кода, вы, конечно, сможете догадаться о том, что константа `3.14159` является приближением числа "пи", но что вы скажете о числе `299792458`? Кроме того, если вас попросят изменить программу так, чтобы число "пи" было записано с точностью до 12 десятичных знаков, то, возможно, вы станете искать в программе число `3.14`, но если кто-нибудь неожиданно решил аппроксимировать число "пи" дробью `22/7`, то, скорее всего, вы ее не найдете. Намного лучше изменить определение константы `pi`, указав требуемое количество знаков.

```
constexpr double pi = 3.14159265359;
```

Следовательно, в программах предпочтительнее использовать не литералы (за исключением самых очевидных, таких как `0` и `1`). Вместо них следует применять константы с информативными именами. Неочевидные литералы в программе (за рамками определения констант) насмешливо называют *магическими*.

В некоторых местах, например в метках оператора `case` (см. раздел 4.4.1.3), язык C++ требует использовать целочисленные *константные выражения*, т.е. выражения, имеющие целочисленные значения и состоящие исключительно из констант. Рассмотрим пример.

```
constexpr int max = 17; // Литерал является
int val = 19;           // константным выражением
max+2                   // Константное выражение
                        // (константа плюс литерал)
val+2                   // Неконстантное выражение:
                        // используется переменная.
```

Кстати, число `299792458` — одна из фундаментальных констант вселенной, означающая скорость света в вакууме, измеренную в метрах в секунду. Если вы ее сразу не узнали, то, вполне возможно, будете испытывать трудности при распознавании других литералов в программе. Избегайте "магических" констант!

Символическая константа `constexpr` должна иметь значение, известное во время компиляции. Например:

```
constexpr int max = 100;
void use(int n)
{
    constexpr int c1 = max+7; // ОК: c1 равно 107
    constexpr int c2 = n+7;   // Ошибка: значение c2 неизвестно
    // ...
}
```

Для ситуаций, когда значение "переменной" инициализируется значением, которое неизвестно во время компиляции, но после инициализации остается неизменным, C++ предлагает вторую разновидность констант (`const`):

```
constexpr int max = 100;
void use(int n)
{
    constexpr int c1 = max+7; // ОК: c1 равно 107
    const int c2 = n+7;       // ОК, но не пытайтесь
                              // изменять значение c2
    // ...
    c2 = 7; // Ошибка: c2 является const
}
```

Такие "константные переменные" очень распространены по двум причинам.

- В C++98 не было `constexpr`, так что программисты использовали ключевое слово `const`.
- "Переменные", не являющиеся константными выражениями (их значения неизвестны во время компиляции), но значения которых не изменяются после инициализации, широко используются сами по себе.

4.3.2. Операторы

До сих пор мы использовали лишь простейшие операторы. Однако вскоре для выражения более сложных операций нам потребуются намного более широкие возможности. Большинство операторов являются привычными, поэтому мы отложим их подробный анализ на будущее. Перечислим наиболее распространенные операторы.

	Имя	Комментарий
`f(a)`	Вызов функции	Передача `a` в качестве аргумента в функцию `f`
`++lval`	Префиксный инкремент	Увеличить на единицу и использовать увеличенное значение
`--lval`	Префиксный декремент	Уменьшить на единицу и использовать уменьшенное значение
`!a`	Не	Результат имеет тип `bool`
`-a`	Унарный минус	
`a*b`	Умножение	
`a/b`	Деление	
`a%b`	Остаток от деления	Только для целочисленных типов
`a+b`	Сложение	

	Имя	Комментарий
`a-b`	Вычитание	
`out<<b`	Запись `b` в поток `out`	Здесь `out` — поток `ostream`
`in>>b`	Считать `b` из потока `in`	Здесь `in` — поток `istream`
`a<b`	Меньше	Результат имеет тип `bool`
`a<=b`	Меньше или равно	Результат имеет тип `bool`
`a>b`	Больше	Результат имеет тип `bool`
`a>=b`	Больше или равно	Результат имеет тип `bool`
`a==b`	Равно	Не путать с оператором `=`
`a!=b`	Не равно	Результат имеет тип `bool`
`a&&b`	Логическое И	Результат имеет тип `bool`
`a||b`	Логические ИЛИ	Результат имеет тип `bool`
`lval=a`	Присваивание	Не путать с оператором `==`
`lval*=a`	Составное присваивание	`lval=lval*a`; используется также с операторами `/`, `%`, `+` и `-`

В выражениях, в которых оператор изменяет операнд, мы использовали имя `lval` (сокращение фразы "значение, стоящее в левой части оператора присваивания"). Полный список операторов приведен в разделе А.5.

Примеры использования логических операторов `&&` (И), `||` (ИЛИ) и `!` (НЕ) приведены в разделах 5.5.1, 7.7, 7.8.2 и 10.4.

Обратите внимание на то, что выражение `a<b<c` означает `(a<b)<c`, а значение выражения `a<b` имеет тип `bool`, т.е. оно может быть либо `true`, либо `false`. Итак, выражение `a<b<c` эквивалентно тому, что выполняется либо неравенство `true<c`, либо неравенство `false<c`. В частности, выражение `a<b<c` не означает "Лежит ли значение `b` между значениями `a` и `c`?", как многие наивно (и совершенно неправильно) думают. Таким образом, выражение `a<b<c` в принципе является бесполезным. Не используйте такие выражения с двумя операциями сравнения и настораживайтесь, когда видите их в чужой программе, — скорее всего, это ошибка.

Инкремент можно выразить по крайней мере тремя способами:

```
++a
a+=1
a=a+1
```

Какой из способов следует предпочесть? Почему? Мы полагаем, что лучшим среди них является первый, `++a`, поскольку он точнее остальных отражает идею инкремента. Он показывает, что мы хотим сделать (добавить к значению переменной `a` единицу и записать результат в переменную `a`). В целом всегда следует выбирать тот способ записи, который точнее выражает вашу идею. Благодаря этому ваша

программа станет точнее, а ее читатель быстрее в ней разберется. Если мы запишем `a=a+1`, то читатель может засомневаться, действительно ли мы хотели увеличить значение переменной `a` на единицу. Может быть, мы просто сделали опечатку вместо `a=b+1`, `a=a+2` или даже `a=a-1`; если же в программе будет использован оператор `++a`, то простора для сомнений останется намного меньше. Пожалуйста, обратите внимание на то, что этот аргумент относится к области удобочитаемости и корректности программы, но не к ее эффективности. Вопреки распространенному мнению, если переменная `a` имеет встроенный тип, то современные компиляторы для выражений `a=a+1` и `++a`, как правило, генерируют совершенно одинаковые коды. Аналогично мы предпочитаем использовать выражение `a *= scale`, а не `a = a*scale`.

4.3.3. Преобразования

Типы в выражениях можно "смешивать". Например, выражение `2.5/2` означает деление значения типа `double` на значение типа `int`. Что это значит? Какое деление выполняется: целых чисел или с плавающей точкой? Целочисленное деление отбрасывает остаток; например, `5/2` равно `2`. Деление чисел с плавающей точкой отличается тем, что остаток в его результате не отбрасывается; например, `5.0/2.0` равно `2.5`. Следовательно, ответ на вопрос "Какие числа делятся в выражении `2.5/2`: целые или с плавающей точкой?" совершенно очевиден: "Разумеется, с плавающей точкой; в противном случае мы потеряли бы информацию". Мы хотели бы получить ответ `1.25`, а не `1`, и именно `1.25` мы и получим. Правило (для рассмотренных нами типов) гласит: если оператор имеет операнд типа `double`, то используется арифметика чисел с плавающей точкой и результат имеет тип `double`; в противном случае используется целочисленная арифметика, и результат имеет тип `int`. Например:

`5/2` равно `2` (а не `2.5`)
`2.5/2` равно `2.5/double(2)`, т.е. `1.25`
`'a'+1` означает `int{'a'}+1`

Записи `type(value)` и `type{value}` означают "преобразовать `value` в тип `type`, как если бы вы инициализировали переменную типа `type` значением `value`". Другими словами, при необходимости компилятор преобразовывает ("повышает") операнд типа `int` в операнд типа `double`, а операнд типа `char` — в операнд типа `int`. Вычислив результат, компилятор может преобразовать его снова для использования в качестве инициализатора или в правой части оператора присваивания, например:

```
double d = 2.5;
int i = 2;
double d2 = d/i; // d2 == 1.25
```

```
int i2 = d/i;     // i2 == 1
int i3 {d/i};     // Ошибка: преобразование double -> int может
                  // быть сужающим (раздел 3.9.2)
d2 = d/i;         // d2 == 1.25
i2 = d/i;         // i2 == 1
```

Будьте осторожны: если выражение содержит числа с плавающей точкой, можно легко забыть о правилах целочисленного деления. Рассмотрим обычную формулу для преобразования температуры по Цельсию в температуру по Фаренгейту: $f=9/5 \cdot c+32$. Ее можно записать так:

```
double dc;
cin >> dc;
double df = 9/5*dc+32; // Осторожно!
```

К сожалению, несмотря на вполне логичную запись, это выражение не дает точного преобразования температуры: значение `9/5` равно `1`, а не `1.8`, как мы надеялись. Для того чтобы формула стала правильной, либо `9`, либо `5` (либо оба числа) следует преобразовать в значение типа `double`.

```
double dc;
cin >> dc;
double df = 9.0/5*dc+32; // Лучше
```

4.4. Инструкции

Выражение вычисляет значение по набору операндов, используя операторы наподобие упомянутых в разделе 4.3. А что делать, если требуется вычислить несколько значений? А что если что-то необходимо сделать многократно? А как поступить, если надо сделать выбор из нескольких альтернатив? А если нам нужно считать входную информацию или вывести результат? В языке C++, как и во многих языках программирования, для создания таких выражений существуют специальные конструкции, именуемые *инструкциями* (statement).

До сих пор мы сталкивались с двумя видами инструкций: выражениями и объявлениями. Инструкции первого типа представляют собой выражения, которые завершаются точкой с запятой, например:

```
a = b;
++b;
```

Перед вами две инструкции, представляющие собой выражения. Например, присваивание `=` — это оператор, поэтому `a=b` — это выражение, и для его завершения необходимо поставить точку с запятой `a=b;`; в итоге возникает инструкция. Зачем нужна точка с запятой? Причина носит скорее технический характер. Рассмотрим пример.

```
a = b ++ b; // Синтаксическая ошибка: пропущена точка с запятой
```

Без точки с запятой компилятор не знает, что означает это выражение: `a=b++; b;` или `a=b; ++b;`. Проблемы такого рода не ограничиваются языками программирования. Например, рассмотрим выражение "Казнить нельзя помиловать!" Казнить или помиловать? Для того чтобы устранить неоднозначность, используются знаки пунктуации. Так, поставив запятую, мы полностью решаем проблему: "Казнить нельзя, помиловать!" Когда инструкции следуют одна за другой, компьютер выполняет их в порядке записи. Рассмотрим пример.

```
int a = 7;
cout << a << '\n';
```

Здесь объявление с инициализацией выполняется до оператора вывода. В целом мы хотим, чтобы инструкция давала какой-то результат (не обязательно имеющий вид значения). Без этого инструкции, как правило, бесполезны. Рассмотрим пример.

```
1+2; // Выполняется сложение, но сумму использовать невозможно
a*b; // Выполняется умножение, но произведение не используется
```

Такие безрезультатные инструкции обычно являются логическими ошибками, и компиляторы часто предупреждают программистов об этом. Таким образом, инструкции, представляющие собой выражения, обычно являются инструкциями присваивания, ввода-вывода или вызова функции.

Упомянем еще об одной разновидности: пустой инструкции. Рассмотрим следующий код:

```
if (x == 5);
{ y = 3; }
```

Это выглядит, как ошибка, и это почти правда. Точка с запятой в первой строке, вообще-то, не должна стоять на этом месте. Но, к сожалению, эта конструкция в языке C++ считается вполне допустимой. Она называется пустой инструкцией, т.е. инструкцией, которая ничего не делает. Пустая инструкция, стоящая перед точкой с запятой, редко бывает полезной. В нашем случае компилятор не выдаст никакого предупреждения об ошибке, и вам будет трудно понять причину неправильной работы программы.

Что произойдет, когда эта программа начнет выполняться? Компилятор проверит, равно ли значение переменной `x` числу `5`. Если это условие истинно, то будет выполнена следующая инструкция (пустая). Затем программа перейдет к выполнению следующей инструкции, присвоив переменной `y` значение `3`. Если же значение переменной `x` не равно `5`, то компилятор не будет выполнять пустую инструкцию (что также не порождает никакого эффекта) и присвоит переменной `y` значение `3` (вряд ли это то, чего вы добивались, если значение `x` не равно `5`). Другими словами,

инструкция `if` не играет никакой роли; переменной `y` будет присвоено значение `3` независимо от значения переменной `x`. Эта ситуация типична для программ, написанных новичками, причем такие ошибки бывает очень трудно обнаружить.

Следующий раздел посвящен инструкциям, позволяющим изменить порядок вычислений и выразить более сложные вычисления, чем те, которые сводятся к последовательному выполнению ряда инструкций.

4.4.1. Инструкции выбора

В программах, как и в жизни, мы часто делаем выбор из нескольких альтернатив. В языке C++ для этого используются инструкции `if` и `switch`.

4.4.1.1. Инструкция `if`

Простейшая форма выбора в языке C++ реализуется с помощью инструкции `if`, позволяющей выбрать одну из двух альтернатив. Рассмотрим пример.

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Пожалуйста, введите два целых числа\n";
    cin >> a >> b;
    if (a<b) // Условие
             // 1-я альтернатива
             // (выбирается, если условие истинно):
         cout << "max(" << a << "," << b
              << ") равно " << b <<"\n";
    else
             // 2-я альтернатива
             // (выбирается, когда условие ложно):
         cout << "max(" << a << "," << b
              << ") равно " << a << "\n";
}
```

Инструкция `if` осуществляет выбор из двух альтернатив. Если его условие является истинным, то выполняется первая инструкция; в противном случае выполняется вторая. Это простая конструкция. Она существует в

большинстве языков программирования. Фактически большинство базовых конструкций в языках программирования представляют собой просто новую запись понятий, известных всем еще со школьной скамьи или даже из детского сада. Например, вам, вероятно, говорили в детском саду, что для того, чтобы перейти улицу, вы должны дождаться, пока на светофоре загорится зеленый свет: "если горит зеленый свет, то можно переходить, а если горит красный свет, то необходимо подождать". В языке C++ это можно записать как-то так:

```
if (traffic_light==green) go();
if (traffic_light==red)   wait();
```

Итак, базовые понятия просты, но и простое понятие можно использовать слишком упрощенно. Рассмотрим неправильную программу (как обычно, не обращаем внимание на отсутствие директив **#include**).

```
// Преобразование дюймов в сантиметры и наоборот
// Суффикс 'i' или 'c' означает единицу измерения на входе
int main()
{
    constexpr double cm_per_inch = 2.54; // Сантиметров в дюйме
    double length = 1;                   // Длина (дюймы или см)
    char unit = 0;
    cout << "Пожалуйста, введите длину"
         << " и единицу измерения (c или i):\n";
    cin >> length >> unit;
    if (unit == 'i')
        cout << length << "in == "
             << cm_per_inch*length << "cm\n";
    else
        cout << length << "cm == "
             << length/cm_per_inch << "in\n";
}
```

На самом деле эта программа работает примерно так, как и предусмотрено: введите **1i**, и вы получите сообщение **1in == 2.54cm**; введите **2.54c**, и вы получите сообщение **2.54cm == 1in**. Поэкспериментируйте — это полезно.

Проблема заключается в том, что вы не можете предотвратить ввод неверной информации. Программа предполагает, что пользователь всегда вводит правильные данные. Условие **unit=='i'** отличает единицу измерения **'i'** от любых других вариантов. Она никогда не проверяет его для единицы измерения **'c'**.

Что произойдет, если пользователь введет **15f** (футов) "просто, чтобы посмотреть, что будет"? Условие **(unit == 'i')** станет ложным, и программа выполнит часть инструкции **else** (вторую альтернативу), преобразовывая сантиметры в дюймы. Вероятно, это не то, чего вы хотели, вводя символ **'f'**.

Мы должны всегда проверять входные данные программы, поскольку — вольно или невольно — кто-нибудь когда-нибудь да введет неверные данные. Программа должна работать разумно, даже если пользователь так не поступает.

Приведем улучшенную версию программы.

```
// Преобразование дюймов в сантиметры и наоборот
// Суффикс 'i' или 'c' означает единицу измерения на входе,
// любой другой суффикс считается ошибкой
int main()
```

```
{
    constexpr double cm_per_inch = 2.54; // количество см в дюйме
    double length = 1;                   // длина (дюймы или см)
    char unit = ' ';          // пробел - не единица измерения
    cout << "Пожалуйста, введите длину"
         << " и единицу измерения (c или i):\n";
    cin >> length >> unit;
    if (unit == 'i')
        cout << length << "in == "
             << cm_per_inch*length << "cm\n";
    else if (unit == 'c')
        cout << length << "cm == "
             << length/cm_per_inch << "in\n";
    else
        cout << "Извините, я не знаю, что такое '"
             << unit << "'\n";
}
```

Сначала мы проверяем условие `unit=='i'`, а затем условие `unit=='c'`. Если ни одно из этих условий не выполняется, выводится сообщение `"Извините, ..."`. Это выглядит так, будто вы использовали инструкцию `"else-if"`, но такой инструкции в языке C++ нет. Вместо этого мы использовали комбинацию двух инструкций `if`. Общий вид инструкции `if` выглядит так:

```
if ( выражение ) инструкция else инструкция
```

Иначе говоря, за ключевым словом `if` следует *выражение* в скобках, а за ним — *инструкция*, ключевое слово `else` и следующая *инструкция*. Вот как можно использовать инструкцию `if` в части `else` инструкции `if`:

```
if ( выражение ) инструкция else
        if ( выражение ) инструкция else инструкция
```

В нашей программе этот прием использован так:

```
if (unit == 'i')
. . . // 1-я альтернатива
else if (unit == 'c')
. . . // 2-я альтернатива
else
. . . // 3-я альтернатива
```

Таким образом можно записать сколь угодно сложную проверку и с каждой альтернативой связать свою инструкцию. Однако следует помнить, что программа должна быть простой, а не сложной. Не стоит демонстрировать свою изобретательность, создавая слишком сложные программы. Лучше докажите свою компетентность, написав самую простую программу, решающую поставленную задачу.

ПОПРОБУЙТЕ

Используя приведенный выше пример, напишите программу для перевода гривен, рублей и юаней в доллары. Если вы любите реализм, уточните обменные курсы в Интернете.

4.4.1.2. Инструкция switch

Сравнение значения `unit` с символами `'i'` и `'c'` представляет собой наиболее распространенную форму выбора: выбор, основанный на сравнении значения с несколькими константами. Такой выбор настолько часто встречается на практике, что в языке C++ для него предусмотрена отдельная инструкция: `switch`. Перепишем наш пример в ином виде

```
int main()
{
    constexpr double cm_per_inch = 2.54; // Количество см в дюйме
    double length = 1;                   // длина (дюймы или см)
    char unit = 'a';
    cout << "Пожалуйста, введите длину"
         << " и единицу измерения (c или i):\n";
    cin >> length >> unit;
    switch (unit) {
    case 'i':
        cout << length << "in == "
             << cm_per_inch*length << "cm\n";
        break;
    case 'c':
        cout << length << "cm == "
             << length/cm_per_inch << "in\n";
        break;
    default:
        cout << "Извините, я не знаю, что такое '"
             << unit << "'\n";
        break;
    }
}
```

Синтаксис оператора `switch` архаичен, но он намного яснее вложенных инструкций `if`, особенно если необходимо сравнить значение со многими константами. Значение, указанное в скобках после ключевого слова `switch`, сравнивается с набором констант. Каждая константа представлена как часть метки `case`. Если значение равно константе в метке `case`, то выбирается инструкция из данного раздела `case`. Каждый раздел `case` завершается ключевым словом `break`. Если значение не соответствует ни одной метке `case`, то выбирается оператор, указанный в разделе `default`. Этот раздел не обязателен, но желателен, чтобы гарантировать перебор всех альтернатив. Если вы еще этого не знали, то учтите,

что программирование приучает человека сомневаться практически во всем.

4.4.1.3. Технические подробности инструкции `switch`

Здесь под техническими подробностями подразумеваются следующие детали, касающиеся инструкции **switch**.

1. Значение, которое определяет выбор варианта, должно иметь целочисленный тип, тип **char** или быть перечислением (раздел 9.5). В частности, выбор по значению **string** произвести невозможно.
2. Значения меток разделов **case** должны быть константными выражениями (раздел 4.3.1). В частности, в метках раздела **case** нельзя использовать переменные.
3. Метки двух разделов **case** не могут иметь одинаковые значения.
4. Один выбор может описываться несколькими метками **case**.
5. Не забывайте завершать каждый раздел **case** ключевым словом **break**. К сожалению, компилятор не предупредит вас, если вы забудете об этом.

Рассмотрим пример.

```
int main() // Выбор можно производить только
{          // по целым числам и т.п. типам данных
    cout << "Вы любите рыбу?\n";
    string s;
    cin >> s;
    switch (s) {     // Ошибка: значение должно быть целым,
    case "нет":      // символом или перечислением
        // . . .
        break;
    case "да":
        // . . .
        break;
    }
}
```

Для выбора альтернатив по значению строки следует использовать инструкцию **if** или ассоциативный массив **map** (подробнее об этом речь пойдет в главе 21).

Инструкция **switch** генерирует оптимизированный код для сравнения значения с набором констант. Для крупных множеств констант он обычно создает более эффективный код по сравнению с набором инструкций **if**. Однако это значит, что значения меток разделов **case** должны быть константами и отличаться одна от другой. Рассмотрим пример.

```
int main() // Метки разделов case должны быть константами
{          // Определяем альтернативы:
    int y = 'y';       // Это может привести к проблемам
    constexpr char n = 'n';
    constexpr char m = '?';
    cout << "Вы любите рыбу?\n";
    char a;
    cin >> a;
    switch (a) {
    case n:
        // . . .
        break;
    case y:            // Ошибка: переменная в метке case
         // . . .
         break;
    case m:
         // . . .
         break;
    case 'n':          // Ошибка: дубликат метки case
                       // (значение метки n равно 'n')
         // . . .
         break;
    default:
         // . . .
        break;
    }
}
```

Часто для разных значений инструкции **switch** целесообразно выполнить одно и то же действие. Было бы утомительно повторять это действие для каждой метки из такого набора. Рассмотрим пример.

```
int main() // Одна инструкция может иметь несколько меток
{
    cout << "Пожалуйста, введите цифру\n";
    char a;
    cin >> a;

    switch (a) {
    case '0': case '2': case '4': case '6': case '8':
          cout << "четная\n";
          break;
    case '1': case '3': case '5': case '7': case '9':
          cout << "нечетная\n";
          break;
    default:
          cout << "не цифра\n";
          break;
    }
}
```

Чаще всего, используя инструкцию **switch**, программисты забывают завершить раздел **case** ключевым словом **break**. Рассмотрим пример.

```
int main() // Пример плохой программы (отсутствует break)
{
    constexpr double cm_per_inch = 2.54; // Количество см в дюйме
    double length = 1;                   // Длина в дюймах или см
    char unit = 'a';
    cout << "Пожалуйста, введите длину и "
         << "единицу измерения (c или i):\n";
    cin >> length >> unit;

    switch (unit) {
    case 'i':
        cout << length << "in == "
             << cm_per_inch*length << "cm\n";
    case 'c':
        cout << length << "cm == "
             << length/cm_per_inch << "in\n";
    }
}
```

К сожалению, компилятор примет этот текст, и когда вы закончите выполнение раздела **case** с меткой **'i'**, просто "провалитесь" в раздел **case** с меткой **'c'**, так что при вводе строки **2i** программа выведет на экран следующие результаты:

```
2in == 5.08cm
2cm == 0.787402in
```

Мы вас предупредили!

ПОПРОБУЙТЕ

Перепишите программу преобразования валют из предыдущего раздела, используя инструкцию **switch**. Добавьте конвертацию евро и фунта стерлингов. Какую из версий программы легче писать, понимать и модифицировать? Почему?

4.4.2. Итерация

Мы редко делаем что-либо только один раз. По этой причине в языках программирования предусмотрены удобные средства для многократного повторения действий. Эта процедура называется *повторением* или (особенно, когда действия выполняются над последовательностью элементов в структуре данных) *итерацией*.

4.4.2.1. Инструкция `while`

В качестве примера итерации рассмотрим первую программу, выполненную на машине с хранимой программой (EDSAC). Она была написана Дэвидом Уилером (David Wheeler) в компьютерной лаборатории Кэмбриджского университета (Cambridge University, England) 6 мая 1949 года. Эта программа вычисляет и распечатывает простой список квадратов.

```
0 0
1 1
2 4
3 9
4 16
. . .
98 9604
99 9801
```

Здесь в каждой строке содержится число, за которым следуют знак табуляции (`'\t'`) и квадрат этого числа. Версия этой программы на языке C++ выглядит так:

```
// Вычисляем и распечатываем таблицу квадратов чисел 0-99
int main()
{
    int i = 0; // Начинаем с нуля
    while (i<100) {
        cout << i << '\t' << square(i) << '\n';
        ++i;    // Инкремент i (i становится равным i+1)
    }
}
```

Обозначение `square(i)` означает квадрат числа `i`. Позднее, в разделе 4.5, мы объясним, как это работает.

Нет, на самом деле первая современная программа не была написана на языке C++, но ее логика была такой же.

- Вычисления начинаются с нуля.
- Проверяем, не достигли ли мы числа 100, и если достигли, то завершаем вычисления.
- В противном случае выводим число и его квадрат, разделенные символом табуляции (`'\t'`), увеличиваем число и повторяем вычисления.

Очевидно, что для этого нам необходимо следующее.

- Способ повторного выполнения инструкции (цикл).
- Переменная, с помощью которой можно было бы отслеживать количество повторений инструкции в цикле (*счетчик цикла*, или *управляющая переменная*). В данной программе она имеет тип `int` и называется `i`.

- Начальное значение счетчика цикла (в данном случае — `0`).
- Критерий прекращения вычислений (в данном случае мы хотим выполнить возведение в квадрат 100 раз).
- Нечто, выполняемое в цикле (*тело* цикла).

В данной программе мы использовали инструкцию **`while`**. Сразу за ключевым словом **`while`** следует условие, а затем тело цикла.

```
while (i<100) // Условие цикла, проверяющее значение счетчика i
{
    cout << i << '\t' << square(i) << '\n';
    ++i;      // Инкремент i (i становится равным i+1)
}
```

Тело цикла — это блок (заключенный в фигурные скобки), который распечатывает таблицу и увеличивает счетчик цикла `i` на единицу. Каждое повторение цикла начинается с проверки условия `i<100`. Если это условие истинно, то мы не заканчиваем вычисления и продолжаем выполнять тело цикла. Если же мы достигли конца, т.е. переменная `i` равна `100`, то выходим из инструкции **`while`** и выполняем инструкцию, следующую за ней. В этой программе после выхода из цикла программа заканчивает работу, поэтому мы из нее выходим.

Счетчик цикла для инструкции **`while`** должен быть определен и проинициализирован заранее, вне инструкции **`while`**. Если мы забудем это сделать, то компилятор выдаст сообщение об ошибке. Если мы определим счетчик цикла, но забудем его проинициализировать, то большинство компиляторов предупредят об этом (сообщив что-то наподобие "локальная переменная `i` не установлена"), но не станут препятствовать выполнению программы. Не пропускайте такие предупреждения! Компиляторы практически никогда не ошибаются, если дело касается неинициализированных переменных. Такие переменные часто становятся источником ошибок. В этом случае следует написать

```
int i = 0; // Начинаем вычисления с нуля
```

и все будет хорошо.

Как правило, написание циклов не вызывает затруднений. Тем не менее при решении реальных задач эта задача может оказаться сложной. В частности, иногда сложно правильно выразить условие и проинициализировать все переменные так, чтобы цикл был корректным.

ПОПРОБУЙТЕ

Символ **`'b'`** равен **`char('a'+1)`**, **`'c'`** равен **`char('a'+2)`** и т.д. Используя цикл, выведите на экран таблицу символов и соответствующих им целочисленных значений.

```
a 97
b 98
...
z 122
```

4.4.2.2. Блоки

Обратите внимание на то, как мы сгруппировали две инструкции, подлежащие выполнению.

```
while (i<100) {
    cout << i << '\t' << square(i) << '\n';
    ++i ; // Инкремент i (i становится равным i+1)
}
```

Последовательность инструкций, заключенных в фигурные скобки ({ и }), называется *блоком* или *составной инструкцией*. Блок — это разновидность инструкции. Пустой блок {} иногда оказывается полезным для выражения того, что в данном месте программы не следует ничего делать. Рассмотрим пример.

```
if (a<=b) { // Ничего не делаем
    }
else {      // Меняем местами a и b
    int t = a;
    a = b;
    b = t;
}
```

4.4.2.3. Инструкция `for`

Итерация над последовательностями чисел настолько часто используется в языке C++, как и в других языках программирования, что для этой операции предусмотрена специальная синтаксическая конструкция. Инструкция **for** похожа на инструкцию **while**, за исключением того, что управление счетчиком цикла сосредоточено в его начале, где за ним легко следить и распознавать. "Первопрограмму" можно переписать так:

```
// Вычисляем и распечатываем таблицу квадратов чисел 0-99
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

Это значит "Выполнить тело цикла, начиная с переменной **i**, равной нулю, и увеличивать ее на единицу при каждом выполнении тела цикла, пока переменная **i** не достигнет значения **100**". Инструкция **for** всегда эквивалентна некоей инструкции **while**. В данном случае конструкция

```
for (int i = 0; i<100; ++i)
    cout << i << '\t' << square(i) << '\n';
```

эквивалентна коду

```
{
    int = 0;          // Инициализатор инструкции for
    while (i<100) { // Условие инструкции for
                      // Тело инструкции for
        cout << i << '\t' << square(i) << '\n';
        ++i;          // Инкремент инструкции for
    }
}
```

Некоторые новички предпочитают использовать инструкцию **while**, а не инструкцию **for**. Однако с помощью инструкции **for** можно создать намного более ясный код, поскольку цикл **for** содержит простые операции инициализации, проверки условия и инкремента счетчика. Используйте инструкцию **while** только тогда, когда нет другого выхода.

Никогда не изменяйте счетчик цикла в теле инструкции **for**. Это нарушит все разумные предположения читателя программы о содержании цикла. Рассмотрим пример.

```
int main()
{
    for (int i = 0; i<100; ++i) { // Для i из диапазона [0,100)
        cout << i << '\t' << square(i) << '\n';
        ++i;                      // Что это? Похоже на ошибку!
    }
}
```

Любой читатель, увидевший этот цикл, разумно предположит, что его тело будет выполнено 100 раз. Однако это не так. Инструкция **++i** в его теле обеспечивает двойной инкремент счетчика **i** на каждой итерации, так что вывод будет осуществлен только для 50 четных чисел. Увидев такой код, вы можете предположить, что это ошибка, вызванная некорректным преобразованием инструкции **for** из инструкции **while**. Если вы действительно хотите, чтобы счетчик увеличивался на 2, напишите следующее:

```
// Вычисляем и выводим на печать таблицу квадратов
// четных чисел из диапазона [0,100)
int main()
{
    for (int i = 0; i<100; i+=2)
    cout << i << '\t' << square(i) << '\n';
}
```

Пожалуйста, не забывайте, что ясная и простая программа короче запутанной. Это общее правило.

ПОПРОБУЙТЕ

Перепишите программу из предыдущей врезки "**Попробуйте**", выводящую символы и соответствующие им целочисленные значения, с помощью инструкции **for**. Затем модифицируйте программу так, чтобы таблица содержала значения для прописных символов и цифр.

Имеется более простая инструкция "**for** для диапазона", предназначенная для обхода коллекций данных, таких как **vector**; см. раздел 4.6.

4.5. Функции

В приведенной выше программе осталась невыясненной роль выражения **square(i)**. Это вызов функции. Конкретнее, это вызов функции, вычисляющей квадрат аргумента **i**. *Функция* — это именованная последовательность инструкций. Она может возвращать результат (который также называется *возвращаемым значением*).

В стандартной библиотеке предусмотрено множество полезных функций, таких как функция для вычисления корня квадратного из числа **sqrt()**, использованная в разделе 3.4. Однако многие функции мы пишем самостоятельно. Рассмотрим возможное определение функции **square**.

```
int square(int x) // Возвращает квадрат числа x
{
    return x*x;
}
```

Первая строка этого определения утверждает, что это функция (об этом говорят скобки), которая называется **square**, принимающая аргумент типа **int** (с именем **x**) и возвращающая значение типа **int** (тип результата всегда предшествует объявлению функции); иначе говоря, ее можно использовать примерно так:

```
int main()
{
    cout << square(2) << '\n';  // Выводим 4
    cout << square(10) << '\n'; // Выводим 100
}
```

Мы не обязаны использовать значение, возвращаемое функцией, но обязаны передать функции именно столько аргументов, сколько требуется. Рассмотрим пример.

```
square(2);              // Возвращаемое значение не используется
                        // Ошибки:
int v1 = square();      //   пропущен аргумент
```

```
int v2 = square;          //   пропущены скобки
int v3 = square(1,2);     //   слишком много аргументов
int v4 = square("two"); //   неверный тип аргумента – нужен int
```

Многие компиляторы предупреждают о неиспользуемых возвращаемых значениях, и все сообщают об ошибках, показанных выше. Кое-кто может решить, будто компилятор достаточно умен, чтобы понять, что, написав строку `"two"`, вы на самом деле имели в виду число 2. Однако компилятор языка C++ сознательно сделан не таким умным. Он просто проверяет, соответствуют ли ваши инструкции синтаксическим правилам языка C++, и точно их выполняет. Если компилятор станет угадывать, что вы имели в виду, то он может ошибиться и вы — или пользователи вашей программы — будете огорчены. Достаточно сложно предсказать, что будет делать ваша программа, если компилятор будет пытаться угадывать ваши намерения.

Тело функции является блоком (см. раздел 4.4.2.2), который выполняет реальную работу.

```
{
    return x*x; // Возвращает квадрат числа x
}
```

Для функции `square` эта работа тривиальна: мы вычисляем квадрат аргумента и возвращаем его в качестве результата. Выразить это на языке C++ проще, чем на естественном языке. Это типично для простых идей. Помимо всего прочего, язык программирования предназначен именно для простого и точного выражения таких идей.

Синтаксис *определения функции* можно описать так:

тип идентификатор (*список_параметров*) *тело_функции*

За *типом* (тип возвращаемого значения) следует *идентификатор* (имя функции), за ним — список параметров в скобках, а затем — тело функции (инструкции, выполняемые при вызове функции). Список аргументов, ожидаемых функцией, называют *списком параметров*, а элементы этого списка — *параметрами* (или *формальными аргументами*). Список параметров может быть пустым. Если вы не хотите возвращать результат, то перед именем функции в качестве типа возвращаемого значения следует поставить ключевое слово `void` (означающее "ничего"). Рассмотрим пример.

```
void write_sorry() // Не принимает никаких аргументов;
{                  // ничего не возвращает
    cout << "Извините\n";
}
```

Технические аспекты функций, связанные с языком программирования, будут описаны в главе 8.

4.5.1. Зачем нужны функции

Функции нужны в ситуациях, когда требуется выделить некие вычисления и присвоить им конкретное имя, руководствуясь следующими соображениями.

- Эти вычисления логически отделены от других.
- Отделение вычислений делает программу яснее (с помощью присваивания имен функциям).
- Функцию можно использовать более чем в одном месте программы.
- Использование функций упрощает отладку программы.

В дальнейшем мы увидим много примеров, в которых следует руководствоваться этими соображениями. Обратите внимание на то, что в реальных программах используются тысячи, а временами даже сотни тысяч функций. Очевидно, что мы никогда не сможем понять такие программы, если их части (т.е. фрагменты вычислений) не будут отделены друг от друга и не получат имен. Кроме того, как мы вскоре убедимся, многие функции часто оказываются полезными в разных ситуациях, и повторять один и тот же код каждый раз довольно утомительно. Например, вы, конечно, можете писать выражения вида `x*x` или `7*7`, или `(x+7)*(x+7)`, а не `square(x)`, `square(7)` или `square(x+7)`, но это только потому, что функция `square` очень проста. Рассмотрим теперь извлечение квадратного корня (в языке C++ эта функция называется `sqrt`): вы явно предпочтете написать выражение `sqrt(x)` или `sqrt(7)`, или `sqrt(x+7)`, а не повторять запутывающий программу (к тому же достаточно сложный и состоящий из большого количества строк) код, вычисляющий квадратный корень, . Более того: вам даже не надо знать, как именно вычисляется квадратный корень числа, так как вам достаточно знать, что функция `sqrt(x)` дает квадратный корень своего аргумента `x`.

В разделе 8.5 мы рассмотрим множество технических деталей, связанных с функциями, а пока рассмотрим еще один пример.

Если бы мы хотели действительно упростить цикл в функции `main()`, то можно было бы написать такой код:

```
void print_square(int v)
{
    cout << v << '\t' << v*v << '\n';
}
int main()
{
    for (int i = 0; i<100; ++i) print_square(i);
}
```

Почему же мы не использовали версию программы на основе функции `print_square()`? Дело в том, что эта программа ненамного проще, чем версия, основанная на функции `square()`, и, кроме того,

- функция `print_square()` является слишком специализированной и вряд ли будет использована в другой программе, в то время как функция `square()`, скорее всего, будет полезной для других программистов;
- функция `square()` не требует подробной документации, а функция `print_square()`, очевидно, требует пояснений.

Функция `print_square()` выполняет два логически отдельных действия:

- выводит числа;
- вычисляет квадраты.

Программы легче писать и понимать, если каждая функция выполняет отдельное логическое действие. По этой причине функция `square()` является более предпочтительной.

В заключение попробуем ответить, почему мы использовали функцию `square(i)`, а не выражение `i*i`, использованное в первой версии программы. Одной из целей функций является упрощение кода путем распределения сложных вычислений по именованным функциям, а для программы 1949 года еще не было аппаратного обеспечения, которое могло бы непосредственно выполнить операцию "умножить". По этой причине в первоначальной версии этой программы выражение `i*i` представляло собой действительно сложное вычисление, как если бы вы выполняли его на бумаге. Кроме того, автор исходной версии программы Дэвид Уилер был изобретателем функций (в то время именуемых подпрограммами (subroutine)) в современном программировании. Так что вполне естественно использовать функции в ремейке программы их изобретателя!

ПОПРОБУЙТЕ

Реализуйте функцию `square()`, не используя оператор умножения; иначе говоря, выполните умножение `x*x` с помощью повторяющегося сложения (начните с переменной, равной нулю, и `x` раз добавьте к ней число `x`). Затем выполните версию "первопрограммы", используя функцию `square()`.

4.5.2. Объявления функций

Вы заметили, что вся информация, необходимая для вызова функции, содержится в первой строке ее объявления? Рассмотрим пример.

```
int square(int x)
```

Этой строки уже достаточно, чтобы написать инструкцию

```
int x = square(44);
```

На самом деле нам не обязательно заглядывать в тело функции. В реальных программах мы часто не хотим углубляться в детали реализации тела функции. Зачем нам знать, что написано в теле стандартной библиотечной функции **sqrt()**? Мы знаем, что она извлекает квадратный корень из своего аргумента. А зачем нам знать, как устроено тело функции **square()**? Разумеется, в нас может взыграть любопытство. Но в подавляющем большинстве ситуаций достаточно знать, как вызвать функцию — а для этого достаточно взглянуть на ее определение. К счастью, в языке C++ существует способ, позволяющий получить эту информацию отдельно, без полного определения функции. Он называется *объявлением функции*.

```
int square(int);      // Объявление функции square
double sqrt(double); // Объявление функции sqrt
```

Обратите внимание на завершающие точку с запятой, которая используется в объявлении функции вместо ее тела в соответствующем определении функции.

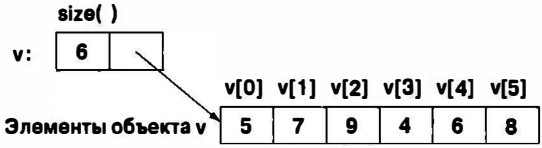
```
int square(int x)     // Определение функции square
{
    return x*x;
}
```

Итак, если мы хотим просто использовать функцию, то достаточно написать ее объявление, но еще чаще — включить его в программу с помощью директивы **#include**. Определение функции может быть в любом другом месте. Это "любое другое место" мы укажем в разделах 8.3 и 8.7. В более крупных программах разница между объявлениями и определениями становится существеннее. В этих программах определения позволяют сосредоточиться одновременно на одном фрагменте программы (раздел 4.2).

4.6. Вектор

Для того чтобы программа делала что-то полезное, необходимо хранить коллекцию данных, с которыми она работает. Например, нам может понадобиться список телефонных номеров, список игроков футбольной команды, список книг, прочитанных в прошлом году, список курсов, график платежей за автомобиль, список прогнозов погоды на следующую неделю, список цен на фотокамеру в интернет-магазине и т.д. Этот перечень можно продолжать до бесконечности, а потому и в программах эти списки встречаются очень часто. В дальнейшем мы рассмотрим множество способов хранения коллекций данных (контейнерные классы, описанные в главах 20 и 21). Пока начнем с простейшего и, пожалуй, наиболее полезного способа хранения данных: с помощью вектора **vector**.

Вектор — это просто последовательность элементов, к которым можно обращаться по индексу. Например, рассмотрим объект типа **vector** с именем **v**.



Первый элемент вектора имеет индекс, равный 0; второй элемент — индекс 1 и т.д. Мы обращаемся к элементу, указывая имя вектора и индекс элемента в квадратных скобках, так что значение **v[0]** равно **5**, значение **v[1]** равно 7 и т.д. Индексы вектора всегда начинаются с нуля и увеличиваются на единицу. Это вам должно быть знакомо: вектор из стандартной библиотеки C++ — это просто новый вариант старой и хорошо известной идеи. Я нарисовал вектор так, как показано на рисунке, чтобы подчеркнуть, что вектор "знает свой размер", т.е. всегда хранит его в одной из ячеек.

Такой вектор можно создать, например, так:

```
vector<int> v = {5, 7, 9, 4, 6, 8}; // Вектор из 6 целых чисел
```

Как видим, для того чтобы создать вектор, необходимо указать тип его элементов и их начальные значения. Тип элементов вектора указывается после слова **vector** в угловых скобках (**<>**), в данном случае это тип **int**. Вот еще один пример.

```
vector<string> philosopher // Вектор из четырех строк
    = { "Kant", "Plato", "Hume", "Kierkegaard" };
```

Естественно, в векторе можно хранить элементы только одного, объявленного типа.

```
philosopher[2] = 99; // Ошибка: присваивание целого числа строке
v[2] = "Hume";       // Ошибка: присваивание строки целому числу
```

Мы можем также определить **vector** заданного размера, не указывая значения его элементов. В этом случае используется запись **(n)**, где **n** — количество элементов, а элементы получают значение по умолчанию для данного типа. Например:

```
vector<int> vi(6);    // vector из 6 int, инициализированных 0
vector<string> vs(4); // vector из 4 строк, инициализированных ""
```

Строка без символов ("") называется пустой строкой.

Обратите внимание, что мы не можем просто обратиться к несуществующему элементу вектора:

```
vd[20000] = 4.7;      // Ошибка времени выполнения программы
```

Ошибки времени выполнения программы и работа с индексами будут описаны в следующей главе.

4.6.1. Обход вектора

`vector` "знает" свой размер, так что мы можем вывести элементы вектора следующим образом:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int i=0; i < v.size(); ++i)
    cout << v[i] << '\n';
```

Вызов `v.size()` дает количество элементов в векторе `v`. В общем случае `v.size()` дает нам возможность обращения к элементам `vector` без риска случайного обращения к элементу за пределами вектора. Диапазон индексов элементов вектора `v` представляет собой `[0,v.size())`. Это математическая запись полуоткрытой последовательности элементов. Первым элементом `v` является `v[0]`, а последним — `v[v.size()-1]`. Если `v.size()==0`, `v` не имеет элементов, т.е. представляет собой пустой вектор. Указанное понятие полуоткрытой последовательности широко используется в C++ и стандартной библиотеке C++ (разделы 17.3, 20.3).

Язык программирования использует преимущество полуоткрытой последовательности для простого цикла по всем элементам последовательности, такой, как элементы вектора, например:

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v)      // Для всех x из v
    cout << x << '\n';
```

Такой цикл называется циклом `for` по диапазону, так как слово *диапазон* (range) часто используется для обозначения последовательности элементов. Мы читаем `for(int x : v)` как "для каждого `int x` в `v`", и этот цикл в точности представляет собой цикл по всем индексам из диапазона `[0,v.size())`. Такой цикл по диапазону используется для простого прохода по всем элементам последовательности по одному элементу за раз. Более сложные задачи наподобие прохода по каждому третьему элементу вектора или только по второй половине диапазона или сравнение элементов двух векторов решаются с помощью более сложного и более общего традиционного цикла `for` (раздел 4.4.2.3).

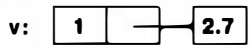
4.6.2. Увеличение вектора

Часто мы начинаем работу с пустым вектором и увеличиваем его размер по мере считывания или вычисления данных. Ключевой операцией является `push_back()`, добавляющая в вектор новый

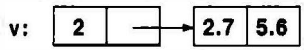
элемент, который становится последним элементом вектора. Рассмотрим пример.

```
vector<double> v; // Начинаем с пустого вектора,
                  // v не содержит ни одного элемента
```

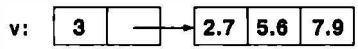

```
v.push_back(2.7); // Добавляем в конец v элемент 2.7
                  // Теперь v содержит один элемент v[0]==2.7
```




```
v.push_back(5.6); // Добавляем в конец v элемент 5.6
                  // Теперь v содержит два элемента и v[1]==5.6
```




```
v.push_back(7.9); // Добавляем в конец v элемент 7.9
                  // Теперь v содержит три элемента и v[2]==7.9
```



Обратите внимание на синтаксис вызова **push_back()**. Он называется *вызовом функции-члена*; **push_back()** является функцией-членом объекта типа **vector**, и поэтому для ее вызова используется форма вызова с точкой.

вызов функции-члена:
имя_объекта.*имя_функции_члена* (*список_аргументов*)

Размер вектора можно определить, вызвав другую функцию-член объекта типа **vector** — **size()**. Изначально значение, возвращаемое **v.size()**, равно 0, а после третьего вызова функции **push_back()** это значение равно **3**.

Если вы имеете опыт программирования, то можете заметить, что тип **vector** похож на массив в языке C и других языках программирования. Однако вам нет необходимости заранее указывать размер (длину) вектора, и вы можете добавлять в него элементы по мере необходимости. В дальнейшем мы убедимся, что тип **vector** из стандартной библиотеки C++ обладает и другими полезными свойствами.

4.6.3. Числовой пример

Рассмотрим более реалистичный пример. Часто у нас имеется ряд значений, которые мы считываем в программу, чтобы затем что-то с ними сделать. Это "что-то" может означать построение графика, вычисление среднего и медианы, поиск наибольшего значения, сортировку, смешивание с другими данными, поиск интересующих нас значений, сравнение с другими данными и т.п. Перечислять операции с данными можно бесконечно, но сначала данные необходимо считать в память компьютера.

Рассмотрим основной способ ввода неизвестного — возможно, большого — объема данных. В качестве конкретного примера попробуем считать числа с плавающей точкой, представляющие собой значения температуры.

```
// Считываем значения температуры в вектор
int main()
{
    vector<double> temps;           // Температуры
    for (double temp; cin>>temp; ) // Чтение в temp
        temps.push_back(temp);      // Вносим temp в vector
    // . . . Какие-то действия . . .
}
```

Итак, что же происходит в этом фрагменте программы? Сначала мы объявляем вектор для хранения данных:

```
vector<double> temps;           // Температуры
```

Здесь указан тип входных данных. Мы считываем и храним значения типа **double**.

Теперь выполняется сам цикл считывания.

```
for (double temp; cin>>temp; ) // Чтение в temp
    temps.push_back(temp);      // Вносим temp в vector
```

Мы определили переменную **temp** типа **double** для считывания значений. Инструкция **cin>>temp** считывает значение типа **double**, а затем это значение вносится в вектор (записывается в его конец). Эти операции уже были продемонстрированы выше. Новизна заключается в том, что в качестве условия выхода из цикла **for** мы используем операцию ввода **cin>>temp**. В основном условие **cin>>temp** является истинным, если значение считано корректно, в противном случае оно является ложным, так что в цикле **for** считываются все числа типа **double**, пока на вход не поступит нечто иное. Например, если мы подадим на вход данные

```
1.2 3.4 5.6 7.8 9.0 |
```

то в вектор **temps** будут занесены пять элементов: **1.2**, **3.4**, **5.6**, **7.8**, **9.0** (именно в указанном порядке, например **temps[0]==1.2**). Для прекращения ввода используется символ **'|'**, т.е. значение, не являющееся **double**. В разделе 10.6 мы обсудим способы прекращения ввода и способы обработки ошибок ввода.

Чтобы ограничить область видимости входной переменной **temp** циклом, мы используем цикл **for**, а не цикл **while**:

```
double temp;
while (cin>>temp)           // Чтение
    temps.push_back(temp); // Внесение в вектор
// Переменная temp может быть использована после цикла
```

Как обычно, в цикле **for** все, что должно происходить, показано в заголовке, так что такой код легче понять, и труднее допустить в нем случайную ошибку.

Записав данные в вектор, мы можем легко с ними работать. В качестве примера вычислим среднее и медиану значений температур.

```
// Вычисляем среднее и медиану значений температур
int main()
{
    vector<double> temps;           // Температуры
    for (double temp; cin>>temp; ) // Чтение в temp
        temps.push_back(temp);      // Внесение temp в вектор

    // Вычисление средней температуры:
    double sum = 0;
    for (int x : temps) sum += x;
    cout << "Средняя температура: "
         << sum/temps.size() << '\n';

    // Вычисление медианы температуры:
    sort(temps);              // Сортировка вектора температур
    cout << "Медианная температура: "
         << temps[temps.size()/2] << '\n';
}
```

Мы вычисляем среднее значение, просто суммируя все элементы и деля сумму на количество элементов (т.е. на значение **temps.size()**).

```
// Вычисление средней температуры:
double sum = 0;
for (int x : temps) sum += x;
cout << "Средняя температура: "
     << sum/temps.size() << '\n';
```

Обратите внимание, насколько удобным оказался оператор **+=**.

Для вычисления медианы (значения, относительно которого половина всех значений оказывается меньше, в другая половина — больше) элементы следует упорядочить. Для этой цели используется вариант алгоритма сортировки из стандартной библиотеки **sort()**.

```
// Вычисление медианы температуры:
sort(temps);              // Сортировка вектора температур
cout << "Медианная температура: "
     << temps[temps.size()/2] << '\n';
```

Этот алгоритм сортировки будет рассмотрен гораздо позже (в главе 20). После сортировки значений температуры найти медиану очень легко: мы просто находим средний элемент, т.е. элемент с индексом **temps.size()/2**. Если проявить определенную придирчивость (характерную для программистов), то можно заметить, что найденное нами значение может не

оказаться медианой в соответствии со строгим определением. Решение этой маленькой проблемы описано в упр. 2 в конце главы.

4.6.4. Текстовый пример

Приведенный выше пример интересен нам с общей точки зрения. Разумеется, среднее значение и медиана температуры интересуют многих людей — метеорологов, аграриев и океанографов, — но нам, программистам, важна обобщенность примера — то, что вектор и простые операции могут применяться в огромном количестве разнообразнейших приложений. Можно сказать, что какие бы данные нас не интересовали, при необходимости их анализа нам необходим вектор (или подобная структура данных; см. главу 21). В качестве примера создадим простой словарь.

```
// Простой словарь: список упорядоченных слов
int main()
{
    vector<string> words;
    for(string temp; cin>>temp; ) // Чтение слов, разделенных
                                  // пробельными символами
        words.push_back(temp);    // Внесение их в вектор
    cout << "Количество слов: " << words.size() << '\n';

    sort(words);                  // Сортировка слов
    for (int i = 0; i<words.size(); ++i)
        if (i==0 || words[i-1]!=words[i]) // Это новое слово?
            cout << words[i] << "\n";
}
```

Если в эту программу ввести несколько слов, то она выведет их в алфавитном порядке без повторов. Например, допустим, что в программу вводятся слова

```
a man a plan a canal panama
```

В ответ программа выведет на экран следующие слова:

```
a
canal
man
panama
plan
```

Как остановить чтение строки? Иначе говоря, как прекратить цикл ввода?

```
for(string temp; cin>>temp; ) // Чтение слов
    words.push_back(temp);    // Внесение их в вектор
```

Когда мы считывали числа (раздел 4.6.2), для прекращения ввода просто вводили какой-то символ, который не был числом. Однако для строк этот прием не работает, так как в строку может быть считан любой

(обычный) символ. К счастью, существуют символы, которые не являются "обычными". Как указывалось в разделе 3.5.1, в операционной системе Windows поток ввода завершается нажатием клавиш <Ctrl+Z>, а в операционной системе Unix — <Ctrl+D>.

Большая часть рассматриваемой программы удивительно похожа на программу для работы с температурой. Фактически мы написали "словарную программу" методом вырезания и вставки (копирования фрагментов кода) из "температурной программы". Единственной новой инструкцией является проверка

```
if (i==0 || words[i-1]!=words[i]) // Это новое слово?
```

Если удалить эту проверку из программы, то вывод станет иным:

```
a
a
a
canal
man
panama
plan
```

Мы не любим повторений, поэтому удаляем их с помощью данной проверки. Что она делает? Она выясняет, отличается ли предыдущее слово от вновь веденного (`words[i-1]!=words[i]`), и если отличается, то слово выводится на экран, а если нет, то не выводится. Очевидно, что у первого слова предшественника нет (`i==0`), поэтому сначала следует проверить номер слова и объединить эти проверки с помощью оператора `||` (или).

```
if (i==0 || words[i-1]!=words[i]) // Это новое слово?
```

Обратите внимание, что мы можем сравнивать строки. Для этого мы используем операторы `!=` (не равно), `==` (равно), `<` (меньше), `<=` (меньше или равно), `>` (больше) и `>=` (больше или равно), которые можно применять и к строкам. Операторы `<`, `>` и прочие используют обычный лексикографический порядок, так что строка `"Ape"` предшествует строкам `"Apple"` и `"Chimpanzee"`.

ПОПРОБУЙТЕ

Напишите программу, "заглушающую" нежелательные слова; иначе говоря, считывайте слова из потока `cin` и выводите их в поток `cout`, заменяя нежелательные слова словом `BLEEP`. Начните с одного нежелательного слова, например

```
string disliked = "Broccoli";
```

Когда отладите программу, добавьте еще несколько нежелательных слов.

4.7. Языковые возможности

В программах для работы с температурами и словарем используются фундаментальные возможности языка, описанные в данной главе: итерация (инструкции `for` и `while`), выбор (инструкция `if`), простая арифметика (операторы `++` и `+=`), операторы сравнения и логические операторы (`==`, `!=` и `||`), переменные и функции (например, `main()`, `sort()` и `size()`). Кроме того, мы использовали возможности стандартной библиотеки, такие как `vector` (контейнер элементов), `cout` (поток вывода) и `sort()` (алгоритм).

Если подсчитать, то окажется, что мы рассмотрели довольно много возможностей языка. Каждая языковая возможность существует для выражения некоторой фундаментальной идеи, и их можно комбинировать бессчетное количество раз, создавая все новые и новые полезные программы. Это принципиальный момент: компьютер — не устройство с фиксированными функциями. Напротив, компьютер можно запрограммировать для любых вычислений и при наличии устройств, обеспечивающих его контакт с внешним миром, в принципе его можно научить делать все что угодно.

Задание

Выполните задание шаг за шагом. Не следует торопиться и пропускать этапы. На каждом этапе проверьте программу, введя по крайней мере три пары значений — чем больше, тем лучше.

1. Напишите программу, содержащую цикл `while`, в котором считываются и выводятся на экран два числа типа `int`. Для выхода из программы используйте символ `'|'`.
2. Измените программу так, чтобы она выводила на экран строку `"Наименьшее значение равно:"` с последующим наименьшим значением, а затем — строку `"Наибольшее значение равно:"` с последующим наибольшим значением.
3. Исправьте программу так, чтобы она выводила строку `"Числа равны"`, но только при равенстве введенных чисел.
4. Измените программу так, чтобы она работала с числами типа `double`, а не `int`.
5. Измените программу так, чтобы она выводила строку `"Числа почти равны"`, если числа отличаются одно от другого меньше чем на `1.0/100`.
6. Теперь измените тело цикла так, чтобы он считывал только по одному числу типа `double` в каждой итерации. Определите две переменные, чтобы отслеживать наименьшее и наибольшее среди всех ранее введенных значений. В каждой итерации цикла выводите только что

введенное число. Если оно окажется наименьшим среди всех введенных, выводите на экран строку **"Наименьшее среди введенных"**. Если же оно окажется наибольшим среди введенных, выводите на экран строку **"Наибольшее среди введенных"**.

7. Добавьте к каждому введенному числу типа **double** единицу измерения; иначе говоря, вводите такие значения, как **10cm**, **2.5in**, **5ft** или **3.33m**. Допустимыми являются четыре единицы измерения: **cm**, **m**, **in**, **ft**. Примите следующие коэффициенты преобразования: **1m == 100cm**, **1in == 2.54cm**, **1ft == 12in**. Индикатор единицы измерения вводите в строку. Можно считать ввод **12 m** (с пробелом между числом и единицей измерения) эквивалентным вводу **12m** (без пробела).
8. Если введена неправильная единица измерения, например **yard**, **meter**, **km** или **gallons**, то такое значение следует отклонить.
9. Отслеживайте сумму введенных значений (помимо наименьшего и наибольшего) и их количество. Когда цикл завершится, выведите на экран наименьшее введенное значение, наибольшее введенное значение, количество введенных значений и их сумму. Обратите внимание на то, что, накапливая сумму, вы должны выбрать для нее единицу измерения (используйте метры).
10. Сохраняйте все введенные значения (преобразованные в метры) в векторе, а по завершении работы цикла выведите их на экран.
11. Перед тем как вывести значения из вектора, отсортируйте их в порядке возрастания.

Контрольные вопросы

1. Что такое вычисления?
2. Что подразумевается под входными данными и результатами вычислений? Приведите примеры.
3. Какие три требования должен иметь в виду программист при описании вычислений?
4. Для чего предназначены выражения?
5. В чем разница между инструкцией и выражением, описанная в тексте главы?
6. Что такое lvalue? Перечислите операторы, требующие наличия lvalue. Почему именно эти, а не другие операторы требуют наличия lvalue?
7. Что такое константное выражение?
8. Что такое литерал?
9. Что такое символьная константа и зачем она нужна?

10. Что такое "магическая" константа? Приведите примеры.
11. Назовите операторы, которые можно применять как к целым числам, так и к числам с плавающей точкой.
12. Какие операторы можно применять только к целым числам, но не к числам с плавающей точкой?
13. Какие операторы можно применять к строкам типа `string`?
14. Когда инструкция `switch` предпочтительнее инструкции `if`?
15. Какие проблемы порождает использование инструкции `switch`?
16. Объясните, каково предназначение каждой части заголовка цикла `for` и в каком порядке они выполняются?
17. Когда следует использовать цикл `for`, а когда — цикл `while`?
18. Как вывести числовой код символа (тип `char`)?
19. Опишите смысл записи `char foo(int x)` в определении функции.
20. Когда часть программы следует оформить в виде функции? Назовите причины.
21. Какие операции можно выполнить над объектом типа `int`, но нельзя над объектом типа `string`?
22. Какие операции можно выполнить над объектом типа `string`, но нельзя над объектом типа `int`?
23. Чему равен индекс третьего элемента вектора?
24. Как записать цикл `for`, в котором выводятся все элементы вектора?
25. Что делает выражение `vector<char>alphabet(26);`?
26. Опишите действие функции-члена вектора `push_back()`.
27. Опишите действие функций-членов вектора `begin()`, `end()` и `size()`.
28. Чем объясняются полезность и популярность типа `vector`?
29. Как упорядочить элементы вектора?

Термины

`begin()`	выбор	определение
`else`	вывод	повторение
`end()`	выражение	разделяй и властвуй
`lvalue`	вычисление	условная инструкция
`push_back()`	декремент	функция
`rvalue`	инкремент	функция-член
`size()`	инструкция	цикл
`sort()`	инструкция `if`	цикл `for`

`vector`	инструкция `switch`	цикл `for` для диапазона
абстракция	итерация	цикл `while`
ввод	объявление	

Упражнения

1. Выполните задание **Попробуйте**, если еще не сделали этого раньше.
2. Допустим, мы определяем медиану последовательности как "число, относительно которого ровно половина элементов меньше, а другая половина — больше". Исправьте программу из раздела 4.6.2 так, чтобы она всегда выводила медиану. Подсказка: медиана не обязана быть элементом последовательности.
3. Прочтите последовательности чисел типа `double` в вектор. Будем считать, что каждое значение представляет собой расстояние между двумя городами, расположенными на определенном маршруте. Вычислите и выведите на печать общее расстояние (сумму всех расстояний). Найдите и выведите на печать наименьшее и наибольшее расстояния между двумя соседними городами. Найдите и выведите на печать среднее расстояние между двумя соседними городами.
4. Напишите программу, угадывающую число. Пользователь должен задумать число от 1 до 100, а программа должна задавать вопросы, чтобы выяснить, какое число он задумал (например, "Задуманное число меньше 50"). Ваша программа должна уметь идентифицировать с помощью не более чем семи попыток. Подсказка: используйте операторы `<` и `<=`, а также конструкцию `if-else`.
5. Напишите программу, выполняющую самые простые функции калькулятора. Ваш калькулятор должен выполнять четыре основные арифметические операции — сложение, вычитание, умножение и деление. Программа должна предлагать пользователю ввести три аргумента: два значения типа `double` и символ операции. Если входные аргументы равны `35.6`, `24.1` и `'+'`, то программа должна вывести на экран строку `"Сумма 35.6 и 24.1 равна 59.7"`. В главе 6 мы рассмотрим более сложный калькулятор.
6. Создайте вектор, хранящий десять строковых значений `"zero"`, `"one"`, ..., `"nine"`. Используйте его в программе, преобразующей цифру в соответствующее строковое представление; например, при вводе цифры `7` на экран должна быть выведена строка `seven`. С помощью этой же программы, используя тот же цикл ввода, преобразуйте строковое представление цифры в числовое; например, при вводе строки `seven` на экран должна быть выведена цифра `7`.

7. Модифицируйте мини-калькулятор, описанный в упр. 5, так, чтобы он принимал на вход цифры, записанные как в числовом, так и в строковом формате.
8. Легенда гласит, что некий царь захотел поблагодарить изобретателя шахмат и предложил ему попросить любую награду. Изобретатель попросил положить на первую клетку одно зерно риса, на вторую — два, на третью — четыре и так далее, удваивая количество зерен на каждой из 64 клеток. На первый взгляд, это желание выглядит вполне скромным, но на самом деле в царстве не было такого количества риса! Напишите программу, вычисляющую, сколько клеток надо заполнить, чтобы изобретатель получил не менее 1000 зерен риса, не менее 1 000 000 зерен риса и не менее 1 000 000 000 зерен риса. Вам, разумеется, понадобится цикл и, вероятно, несколько переменных типа `int`, для того чтобы отслеживать текущий номер клетки, количество зерен в текущей клетке и количество зерен во всех предыдущих клетках. Мы предлагаем на каждой итерации цикла выводить на экран значения всех этих переменных, чтобы видеть промежуточные результаты.
9. Попробуйте вычислить количество зерен риса, запрошенных изобретателем шахмат в упр. 8. Оказывается, что это число настолько велико, что для его точного представления не подходит ни тип `int`, ни тип `double`. Чему равно наибольшее количество клеток, для которых еще можно вычислить точное количество зерен риса (с использованием переменной типа `int`). Определите наибольшее количество клеток, для которых еще можно вычислить приближенное количество зерен (с использованием переменной типа `double`)?
10. Напишите программу для игры "Камень, бумага, ножницы". Если вы не знаете правил этой игры, попробуйте выяснить их у друзей или с помощью Google (поиск информации — обычное занятие программистов). Для решения поставленной задачи используйте инструкцию `switch`. Кроме того, машина должна давать случайные ответы (т.е. выбирать камень, бумагу или ножницы на следующем ходу случайным образом). Написать настоящий генератор случайных чисел прямо сейчас вам будет довольно тяжело, поэтому заранее заполните вектор последовательностью значений. Если встроить этот вектор в программу, то она всегда будет играть одну и ту же игру, поэтому целесообразно позволить пользователю самому вводить некоторые значения. Попробуйте помешать пользователю легко угадывать следующий ход машины.
11. Напишите программу, находящую все простые числа между 1 и 100. Для этого можно написать функцию, проверяющую, является ли число простым (т.е. делится ли оно на простое число, не превосходящее

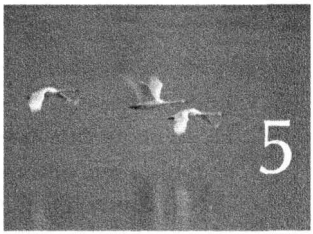
данное), используя вектор простых чисел, записанный в порядке возрастания (например, если вектор называется `primes`, то `primes[0]==2`, `primes[1]==3`, `primes[2]==5` и т.д.). Напишите цикл перебора чисел от 1 до 100, проверьте каждое из них на простоту и сохраните найденные простые числа в векторе. Напишите другой цикл, в котором все найденные простые числа выводятся на экран. Сравните полученные результаты с вектором `primes`. Первым простым числом считается число `2`.

12. Измените программу из предыдущего упражнения так, чтобы в нее вводилось число `max`, а затем найдите все простые числа от `1` до `max`.
13. Напишите программу, находящую все простые числа между 1 и 100. Для решения этой задачи существует классический метод "Решето Эратосфена". Если этот метод вам неизвестен, поищите его описание в Интернете. Напишите программу, использующую этот метод.
14. Измените программу, описанную в предыдущем упражнении, так, чтобы в нее вводилось число `max`, а затем найдите все простые числа от `1` до `max`.
15. Напишите программу, принимающую на вход число `n` и находящую первые `n` простых чисел.
16. В заданиях вам было предложено написать программу, которая для определенного ряда чисел определяла бы наибольшее и наименьшее числа в нем. Число, которое повторяется в последовательности наибольшее количество раз, называется *модой*. Напишите программу, определяющую моду множества положительных чисел.
17. Напишите программу, определяющую наименьшее и наибольшее значения, а также моду последовательности строк типа `string`.
18. Напишите программу для решения квадратных уравнений. Квадратное уравнение имеет вид $ax^2+b+c=0$. Если вы не знаете формул для решения такого уравнения, проведите дополнительные исследования. Напомним, что программисты часто выполняют такие исследования, прежде чем приступить к решению задачи. Для ввода чисел `a`, `b` и `c` используйте переменные типа `double`. Поскольку квадратное уравнение имеет два решения, выведите оба значения, `x1` и `x2`.
19. Напишите программу, в которую сначала вводится набор пар, состоящих из имени и значения, например `Joe 17` и `Barbara 22`. Для каждой пары занесите имя в вектор `names`, а число — в вектор `scores` (в соответствующие позиции, так что если `names[7]=="Joe"`, то `scores[7]==17`). Прекратите ввод, встретив строку ввода `NoName 0`. Убедитесь, что каждое имя единственное, и выведите сообщение об ошибке, если имеется имя, введенное дважды. Выведите на печать все пары (*имя*, *баллы*) по одной в строке.

20. Измените программу из упр. 19 так, чтобы при вводе имени она выводила соответствующее количество баллов или сообщение **"Имя не найдено"**.
21. Измените программу из упр. 19 так, чтобы при вводе целого числа она выводила имена всех студентов, получивших указанное количество баллов, или сообщение **"Баллы не найдены"**.

Послесловие

С философской точки зрения вы уже можете делать с помощью компьютера все, что захотите. Остальное — детали! Разумеется, важность деталей и практических навыков несомненна, поскольку вы только начинаете программировать. Но мы говорим серьезно. Инструменты, представленные в главе, позволяют описывать любые вычисления: у вас может быть столько переменных, сколько вам нужно (включая векторы и строки), вы можете выполнять арифметические операции, сравнения, а также выбор и итерации. С помощью этих примитивов можно выразить любые вычисления. Вы можете вводить и выводить числа и строки в виде текста (и даже графиков). Можете даже организовать вычисления в виде набора функций. Осталось только научиться писать хорошие программы, т.е. правильные, понятные и эффективные. Не менее важно, чтобы вы смогли научиться этому, затратив разумное количество сил.



5 Ошибки

Я понял, что с этого момента большую часть моей жизни буду искать и исправлять свои же ошибки.
— Морис Уилкс (Maurice Wilkes, 1949)

В этой главе обсуждаются вопросы, связанные с корректностью программ, а также с ошибками и методами их исправления. Если вы новичок, то обсуждение покажется вам иногда слишком абстрактным, а иногда — слишком подробным. Неужели обработка ошибок настолько важна? Да! И так или иначе вы должны научиться этому, прежде чем приступать к разработке программ, предназначенных для других людей. Мы попытаемся показать вам, что значит "думать, как программист" и как сочетать самые абстрактные стратегии с тщательным анализом деталей и альтернатив.

5.1. Введение

В предыдущих главах мы часто упоминали об ошибках, и, выполняя задания и упражнения, вы уже отчасти поняли, почему. При разработке программ ошибки просто неизбежны, хотя окончательный вариант программы должен быть безошибочным или по крайней мере не должен содержать неприемлемых ошибок.

Существует множество способов классификации ошибок. Рассмотрим один из них.

- *Ошибки времени компиляции*. Это ошибки, обнаруженные компилятором. Их можно подразделить на категории в зависимости от того, какие правила языка он нарушают:
  - синтаксические ошибки;
  - ошибки, связанные с типами.
- *Ошибки времени редактирования связей*. Это ошибки, обнаруженные редактором связей при попытке объединить объектные файлы в выполнимый модуль.
- *Ошибки времени выполнения*. Это ошибки, обнаруженные проверками в работающей программе. Их можно подразделить на следующие категории:
  - ошибки, обнаруженные компьютером (аппаратным обеспечением и/или операционной системой);
  - ошибки, обнаруженные библиотекой (например, стандартной библиотекой C++);

- ошибки, обнаруженные кодом пользователя.

- *Логические ошибки.* Это ошибки, найденные программистом в поисках причины неправильных результатов.

Соблазнительно сказать, что задача программиста — устранить все ошибки. Разумеется, это было бы прекрасно, но часто этот идеал оказывается недостижимым. На самом деле для реальных программ трудно сказать, что подразумевается под выражением "все ошибки". Например, если во время выполнения своей программы мы выдернем электрический шнур из розетки, то следует ли это рассматривать как ошибку и предусмотреть ее обработку? В большинстве случаев совершенно очевидно, что ответ будет отрицательным, но в программе медицинского мониторинга или в программе, управляющей телефонным узлом, это уже не так. В этих ситуациях пользователь вполне обоснованно может потребовать, чтобы система, частью которой является ваша программа, продолжала выполнять осмысленные действия, даже если исчезло энергопитание компьютера или космические лучи повредили его память. Основной вопрос заключается в следующем: должна ли программа сама обнаруживать ошибки? Если иное не указано явно, будем предполагать, что ваша программа удовлетворяет следующим условиям.

1. Должна вычислять желаемые результаты при всех допустимых входных данных.
2. Должна выдавать осмысленные сообщения обо всех неправильных входных данных.
3. Не обязана беспокоиться о некорректном поведении аппаратного обеспечения.
4. Не обязана беспокоиться о некорректном поведении системного программного обеспечения.
5. Она может завершать работу после обнаружения ошибки.

Программы, для которых предположения 3–5 не выполняются, выходят за рамки рассмотрения нашей книги. В то же время предположения 1 и 2 являются частью базового определения профессионализма, а профессионализм — это именно то, к чему мы стремимся. Даже если мы не всегда соответствуем идеалу на 100%, он должен быть идеалом.

При создании программы ошибки естественны и неизбежны. Вопрос лишь в том, что с ними делать. По нашему мнению, при разработке серьезного программного обеспечения на попытки обойти, найти и исправить ошибки направлено не менее 90% усилий программистов. Для программ, безопасность работы которых является критичной, эти усилия займут еще больше времени. В маленьких программах легко избежать ошибок, но если вы неряшливы, значит, проблемы будут и в этом случае.

Мы предлагаем три подхода к разработке приемлемого программного обеспечения.

- Организовать программное обеспечение так, чтобы минимизировать количество ошибок.
- Исключить большинство сделанных ошибок в ходе отладки и тестирования.
- Убедиться, что оставшиеся ошибки не серьезны.

Ни один из этих подходов сам по себе не позволяет полностью исключить ошибки; мы обязаны использовать их все.

При разработке надежных программ, т.е. программ, которые делают то, для чего предназначены при допустимом уровне ошибок, большую роль играет опыт. Пожалуйста, не забывайте, что в идеале программы всегда должны работать правильно. Разумеется, на практике мы можем лишь приблизиться к идеалу, но отказ от трудоемких попыток приблизиться к идеалу заслуживает безусловного осуждения.

5.2. Источники ошибок

Перечислим несколько источников ошибок.

- *Плохая спецификация.* Если мы слабо представляем себе, что должна делать программа, то вряд ли сможем адекватно проверить все ее "темные углы" и убедиться, что все варианты обрабатываются правильно (т.е. что при любом входном наборе данных мы получим либо правильный ответ, либо осмысленное сообщение об ошибке).
- *Неполные программы.* В ходе разработки неизбежно возникают варианты, которые мы не предусмотрели. Наша цель — убедиться, что все варианты обработаны правильно.
- *Непредусмотренные аргументы.* Функции принимают аргументы. Если функция принимает аргумент, который не был предусмотрен, то возникнет проблема, как, например, при вызове стандартной библиотечной функции извлечения корня из –1.2: `sqrt(-1.2)`. Поскольку функция `sqrt()` вычисляет квадратный корень от значения типа `double` и возвращает результат типа `double`, в этом случае она не сможет вернуть правильный результат. Такие проблемы обсуждаются в разделе 5.5.3.
- *Непредусмотренные входные данные.* Обычно программы считывают данные (с клавиатуры, из файлов, средствами графического пользовательского интерфейса, из сетевых соединений и т.д.). Как правило, программы предъявляют к входным данным много требований,

например, чтобы было введено целочисленное значение. Но что если пользователь введет не ожидаемое целочисленное значение, а строку "Отвали!"? Этот вид проблем обсуждается в разделах 5.6.3 и 10.6.

- *Непредусмотренное состояние.* Большинство программ хранит большое количество данных ("состояний"), предназначенных для использования разными частями системы. К их числу относятся списки адресов, каталоги телефонов или записанные в **vector** данные о температуре. Что если эти данные окажутся неполными или неправильными? В этом случае разные части программы должны сохранять управляемость. Эти проблемы обсуждаются в разделе 26.3.5.
- *Логические ошибки.* Это ошибки, когда программа просто делает не то, что от нее ожидается; мы должны найти и исправить эти ошибки. Примеры поиска таких ошибок приводятся в разделе 6.6 и 6.9.

Данный список имеет практическое применение. Мы можем использовать его для контроля качества программы. Ни одну программу нельзя считать законченной, пока не исследованы все потенциально возможные источники ошибок. Этот список целесообразно иметь в виду уже в самом начале проекта, поскольку очень маловероятно, что поиск и устранение ошибок в программе, запущенной на выполнение без предварительного анализа, не потребует серьезной переработки программы.

5.3. Ошибки времени компиляции

Когда вы пишете программы, на передовой линии защиты от ошибок находится компилятор. Перед тем как приступить к генерации кода, компилятор анализирует исходный текст в поисках синтаксических ошибок и ошибок типов. Только когда компилятор убедится, что программа полностью соответствует спецификациям языка, он разрешит ее дальнейшую обработку. Многие ошибки, которые обнаруживает компилятор, относятся к категории "грубых ошибок", вызванных опечатками или в результате неполного редактирования исходного текста. Другие ошибки являются результатом плохого понимания взаимодействия частей нашей программы. Новичкам компилятор часто кажется очень ограниченным, но по мере изучения свойств языка — и особенно его системы типов — вы по достоинству оцените способности компилятора выявлять проблемы, которые в противном случае заставили бы вас часами искать ошибки.

В качестве примера рассмотрим вызовы следующей простой функции:

```
int area(int length, int width); // Вычисление площади треугольника
```

5.3.1. Синтаксические ошибки

Что произойдет, если мы вызовем функцию **area()** следующим образом:

```
int s1 = area(7;   // Ошибка: пропущена скобка )
int s2 = area(7)   // Ошибка: пропущена точка с запятой ;
Int s3 = area(7); // Ошибка: Int – это не тип
int s4 = area('7); // Ошибка: пропущена кавычка '
```

Каждая из этих строк содержит синтаксическую ошибку; иначе говоря, они не соответствуют грамматике языка C++, поэтому компилятор их отклоняет. К сожалению, синтаксические ошибки не всегда можно описать так, чтобы программист легко понял, в чем дело. Это объясняется тем, что компилятор должен проанализировать несколько более крупный фрагмент текста, чтобы понять, действительно ли он обнаружил ошибку. В результате даже самые простые синтаксические ошибки (обнаружив которые, бывает трудно поверить, что это набранный вами текст!) часто описываются довольно запутанно, и при этом компилятор может указать на строку, которая расположена в программе немного дальше, чем сама ошибка.

Так что, если вы не видите ничего неправильного в строке, на которую указал компилятор, проверьте предшествующие строки программы.

Обратите внимание на то, что компилятор не знает, что именно вы пытаетесь сделать, поэтому формулирует сообщения об ошибках с учетом того, что вы на самом деле сделали, а не ваших намерений. Например, обнаружив ошибочное объявление переменной **s3**, компилятор вряд ли напишет что-то вроде следующей фразы:

> "Вы неправильно написали слово **int**; не следует употреблять прописную букву **i**."

Скорее, он выразится так:

> "Синтаксическая ошибка: пропущена '**;**' перед идентификатором '**s3**'"
>
> "У переменной '**s3**' пропущен идентификатор класса или типа"
>
> "Неправильный идентификатор класса или типа '**Int**'"

Такие сообщения выглядят туманно, пока вы не научитесь их понимать и использовать. Разные компиляторы могут выдавать разные сообщения, анализируя один и тот же код. К счастью, вы достаточно быстро научитесь понимать эти сообщения без каких-либо проблем. В общем, все эти зашифрованные сообщения можно перевести так:

> "Перед переменной **s3** имеется синтаксическая ошибка, и надо что-то сделать либо с типом **Int**, либо с переменной **s3**."

После этого уже нетрудно разобраться в проблеме.

ПОПРОБУЙТЕ

Попробуйте скомпилировать приведенные примеры и проанализируйте сообщения компилятора.

5.3.2. Ошибки, связанные с типами

После того как вы устраните синтаксические ошибки, компилятор начнет выдавать сообщения об ошибках, связанных с типами; иначе говоря, он сообщит о несоответствиях между объявленными типами (или о типах, которые вы забыли объявить) ваших переменных, функций и так далее и типами значений и выражений, которые вы им присваиваете, передаете в качестве аргументов и т.д.

```
int x0 = arena(7);          // Ошибка: необъявленная функция
int x1 = area(7);           // Ошибка: неверное количество
                            // аргументов
int x2 = area("seven",2); // Ошибка: первый аргумент
                            // имеет неверный тип
```

Рассмотрим эти ошибки.

1. При вызове функции **arena(7)** мы сделали опечатку: вместо **area** набрали **arena**, поэтому компилятор думает, что мы хотим вызвать функцию с именем **arena**. (А что еще он может "подумать"? Только то, что мы сказали.) Если в программе нет функции с именем **arena()**, то вы получите сообщение об ошибке, связанной с необъявленной функцией. Если же в программе есть функция с именем **arena** и эта функция может принимать значение 7 в качестве аргумента, то вы столкнетесь с гораздо худшей проблемой: программа будет скомпилирована, но работать будет неправильно (такие ошибки называют логическими; см. раздел 5.7).
2. Анализируя выражение **area(7)**, компилятор обнаруживает неверное количество аргументов. В языке C++ вызов каждой функции должен содержать ожидаемое количество аргументов, с верными типами и в правильном порядке. При правильном использовании система типов может стать мощным инструментом, позволяющим избежать ошибок времени выполнения программы (раздел 14.1).
3. Записывая выражение **area("seven",2)**, вы надеялись, что компилятор увидит строку **"seven"** и поймет, что вы имели в виду целочисленное значение 7? Напрасно. Если функция ожидает целочисленное значение, ей нельзя передавать строку. Язык C++ поддерживает некоторые неявные преобразования типов (см. раздел 3.9), но не позволяет конвертировать тип **string** в тип **int**. Компилятор не станет гадать,

что вы имели в виду. В самом деле, а если бы вы сделали вызовы `area("Hovel lane",2)`, `area("7,2")` и `area("sieben","zwei")`, какую реакцию компилятора вы бы хотели получить в этих случаях?

Мы перечислили лишь несколько примеров. Существует намного больше ошибок, которые компилятор может найти в вашей программе.

ПОПРОБУЙТЕ

Попробуйте скомпилировать приведенные примеры и проанализируйте сообщения компилятора. Придумайте еще несколько ошибок и проверьте реакцию компилятора на них.

5.3.3. Не ошибки

Работая с компилятором, хочется, чтобы он был достаточно умным и угадывал ваши намерения; иначе говоря, хочется, чтобы некоторые ошибки компилятор не считал таковыми. Это естественно. Но поверьте, каким удивительным это ни казалось бы вам сейчас, по мере накопления опыта вы захотите, чтобы компилятор был более придирчивым и отбраковывал побольше, а не поменьше. Рассмотрим пример.

```
int x4 = area(10,-7);        // OK, но что это за прямоугольник,
                             // с отрицательной шириной?
int x5 = area(10.7,9.3);     // OK, но будет вызвана area(10,9)
char x6 = area(100, 9999); // OK, но результат будет усечен
```

Строка с переменной `x4` не вызовет никаких сообщений компилятора. С его точки зрения вызов `area(10,-7)` является корректным: функция `area()` ожидает два целочисленных значения, их она и получает; никто не говорил, что они обязаны быть положительными.

В строке с переменной `x5` хороший компилятор должен был бы предупредить, что значения типа `double`, равные `10.7` и `9.3`, будут преобразованы в значения типа `int`, равные `10` и `9` (см. 3.9.2). Однако (устаревшие) правила языка утверждают, что вы можете неявно преобразовать переменную типа `double` в переменную типа `int`, поэтому у компилятора нет никаких оснований отвергать вызов `area(10.7,9.3)`.

Инициализация переменной `x6` представляет собой вариант той же проблемы, что и вызов `area(10.7,9.3)`. Значение типа `int`, возвращенное после вызова `area(100,9999)`, вероятно, равное `999900`, будет присвоено переменной типа `char`. В итоге, скорее всего, в переменную `x6` будет записано "усеченное" значение `-36`. И вновь хороший компилятор должен выдать предупреждение, даже если устаревшие правила языка позволяют ему не делать этого.

По мере приобретения опыта вы научитесь использовать как сильные, так и слабые стороны компилятора. Однако не будьте слишком самоуверенными: выражение "программа скомпилирована" вовсе не означает, что она будет выполнена. Даже если она будет запущена на выполнение, то, как правило, сначала будет выдавать неправильные результаты, пока вы не устраните все логические недостатки.

5.4. Ошибки времени редактирования связей

Любая программа состоит из нескольких отдельно компилируемых частей, которые называют *единицами трансляции* (translation units). Каждая функция в программе должна быть объявлена с одним и тем же типом во всех единицах трансляции, в которых она используется. Для этого используются заголовочные файлы (подробно о них речь пойдет в разделе 8.3). Кроме того, каждая функция должна быть определена в программе ровно один раз. Если хотя бы одно из этих правил нарушено, редактор связей сообщит об ошибке. Способы избежать ошибок времени редактирования связей (времени компоновки) рассматриваются в разделе 8.3. А пока рассмотрим пример программы, которая порождает типичную ошибку компоновки.

```
int area(int length, int width); // Площадь прямоугольника

int main()
{
    int x = area(2,3);
}
```

Если только функция **area()** не определена в другом исходном файле и не связана с кодом, сгенерированным из нашего файла, с помощью редактора связей, то он сообщит об отсутствии определения функции **area()**.

Определение функции **area()** должно иметь точно такие же типы (возвращаемого значения и аргументов), что и использованные в нашем файле.

```
int area(int x, int y) { /* ... */ } // "Наша" функция area()
```

Функции с таким же именем, но с другими типами не будут соответствовать "нашей" функции и будут проигнорированы.

```
double area(double x, double y)
        { /* ... */ }     // Не "наша" area()
int area(int x, int y, char unit)
        { /* ... */ }     // Не "наша" area()
```

Обратите внимание на то, что ошибка при наборе имени функции обычно не порождает ошибки времени редактирования связей. Вместо этого

компилятор обнаруживает необъявленную функцию и немедленно выдает сообщение об ошибке. И это хорошо: ошибки времени компиляции обнаруживаются раньше ошибок времени редактирования связей и, как правило, устраняются легче.

Как указывалось выше, правила связывания функций распространяются и на все другие сущности программы, например, на переменные и типы: каждая сущность с заданным именем должна быть определена только один раз, но объявлять ее можно сколько угодно, лишь бы все эти объявления были в точности согласованными по типам. Подробнее об этом говорится в разделах 8.2 и 8.3.

5.5. Ошибки времени выполнения программы

Если программа не содержит ошибок времени компиляции и времени редактирования связей, она выполняется. Здесь-то и начинается настоящее веселье. При написании программы можно выявлять и устранять ошибки, но исправить ошибку, обнаруженную во время выполнения программы, не всегда легко. Рассмотрим пример.

```
int area(int length, int width) // Площадь прямоугольника
{
    return length*width;
}

int framed_area(int x, int y)   // Площадь в пределах рамки
{
    return area(x-2,y-2);
}

int main()
{
    int x = -1;
    int y = 2;
    int z = 4;
    // ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3; // Преобразуем в тип double,
                        // чтобы выполнить деление с плавающей точкой
}
```

Для того чтобы сделать проблему менее очевидной для человека и усложнить задачу компилятора, в качестве аргументов мы решили использовать переменные **x**, **y** и **z**, а не непосредственные числа. Приведенные вызовы функций возвращают отрицательные числа, присвоенные переменным **area1** и **area2**. Можно ли принять эти ошибочные результаты,

противоречащие законам математики и физики? Если нет, то где следует искать ошибку: в исходном тексте, вызвавшем функцию `area()`, или в самой функции? И какое сообщение об ошибке следует выдать?

Прежде чем пытаться ответить на эти вопросы, проанализируем вычисление переменной `ratio` в приведенном выше коде. Оно выглядит довольно невинно. Вы не заметили, что с этим кодом что-то не так? Если нет, посмотрите внимательнее: переменная `area3` будет равна 0, поэтому в выражении `double(area1)/area3` возникает деление на нуль. Это приводит к ошибке, обнаруживаемой аппаратным обеспечением; в результате выполнение программы прекращается с выдачей на экран довольно невнятного сообщения. Вы и ваши пользователи будете сталкиваться с такими проблемами постоянно, если не научитесь выявлять и исправлять ошибки, возникающие на этапе выполнения программы. Большинство людей нервно реагируют на такие сообщения аппаратного обеспечения, так как им сложно понять, что происходит, когда на экране появляется сообщение вроде "Что-то пошло не так!" Этого недостаточно для того, чтобы предпринять какие-то конструктивные действия, поэтому пользователи злятся и проклинают программиста, написавшего такую программу.

Давайте попробуем разобраться с ошибкой, связанной с вызовом функции `area()`. У нас есть две очевидные альтернативы.

1. Некорректные аргументы должен обрабатывать код, вызывающий функцию `area()`.
2. Некорректные аргументы должен обрабатывать код вызываемой функции `area()`.

5.5.1. Обработка ошибок в вызывающем коде

Сначала рассмотрим первую альтернативу ("Пользователь должен быть внимательным!"). Именно ее нам следовало бы принять, например, если бы функция `area()` была библиотечной функцией, которую невозможно модифицировать. Хорошо это или плохо, но данная ситуация является самой распространенной.

Предотвратить ошибку при вызове функции `area(x,y)` в модуле `main()` относительно просто:

```
if (x<=0) error("Неположительное x");
if (y<=0) error("Неположительное y");
int area1 = area(x,y);
```

Нам остается только решить, что делать, обнаружив ошибку. Здесь мы решили вызывать функцию `error()`, которая должна сделать что-то полезное. В заголовочном файле `std_lib_facilities.h` действительно описана функция `error()`, которая по умолчанию останавливает выполнение программы, сопровождая остановку системным сообщением об ошибке со

строкой, которая передается как аргумент функции **error()**. Если вы предпочитаете писать собственные сообщения об ошибках или предпринимать другие действия, то можете перехватывать исключение **runtime_error** (разделы 5.6.2, 7.3, 7.8, Б.2.1). Этот подход достаточен для большинства студенческих программ и является примером стиля, который можно применять для более сложной обработки ошибок.

Если мы не требуем сообщать об ошибках в каждом из аргументов, код можно упростить.

```
if (x<=0 || y<=0) error("Неположительный аргумент функции area()");
      // || означает "ИЛИ"
int area1 = area(x,y);
```

Для того чтобы полностью защитить функцию **area()** от неправильных аргументов, необходимо исправить вызовы функции **framed_area()**. Мы могли бы написать следующий код:

```
if (z<=2)
    error("Неположительный второй аргумент функции area()"
          " при вызове из функции framed_area()");
int area2 = framed_area(1,z);
if (y<=2 || z<=2)
    error("Неположительный аргумент функции area()"
          " при вызове из функции framed_area()");
int area3 = framed_area(y,z);
```

Это не только запутанно, но и неверно в принципе. Такой код можно написать, лишь точно зная, как функция **framed_area()** использует функцию **area()**. Мы должны знать, что функция **framed_area()** вычитает 2 из каждого аргумента. Но ведь мы не должны знать такие детали! А что если кто-нибудь изменит функцию **framed_area()**, и вместо 2 она станет вычитать 1? В этом случае нам пришлось бы искать каждый вызов функции **framed_area()** и соответствующим образом изменять фрагменты кода обработки ошибок. Такой код называется "хрупким", потому что легко выходит из строя. Он также является ярким примером применения "магических констант" (раздел 4.3.1). Код можно сделать более надежным, дав величине, вычитаемой функцией **framed_area()**, имя.

```
constexpr int frame_width = 2;
int framed_area(int x, int y) // Площадь в рамке
{
    return area(x-frame_width,y-frame_width);
}
```

Это имя можно использовать в коде, вызывающем функцию **framed_area()**.

```
if (1-frame_width<=0 || z-frame_width<=0)
    error("неположительный аргумент функции area()"
          " при вызове из функции framed_area()");
```

```
int area2 = framed_area(1,z);
if (y-frame_width<=0 || z-frame_width<=0)
    error("неположительный аргумент функции area()"
          " при вызове из функции framed_area()");
int area3 = framed_area(y,z);
```

Взгляните на этот код! Вы уверены, что он правильный? Он вам нравится? Легко ли его читать? В действительности он уродлив (а значит, подвержен ошибкам). В результате наших неуклюжих попыток размер кода увеличился втрое, а детали реализации `framed_area()` выплыли наружу. Существует более правильное решение!

Посмотрите на первоначальный код.

```
int area2 = framed_area(1,z);
int area3 = framed_area(y,z);
```

Он может быть неверным, но по крайней мере можно легко понять, что должно происходить. Мы можем сохранить эту ясность, поместив проверку ошибки в функцию `framed_area()`.

5.5.2. Обработка ошибок в вызываемом коде

Проверка корректности аргументов в функции `framed_area()` не вызывает затруднений, а выдачу сообщения об ошибках можно по-прежнему поручить функции `error()`.

```
int framed_area(int x, int y) // Площадь в рамке
{
    constexpr int frame_width = 2;
    if (x-frame_width<=0 || y-frame_width<=0)
        error("Неположительный аргумент функции area()"
              " при вызове из функции framed_area()");
    return area(x-frame_width,y-frame_width);
}
```

Это решение выглядит неплохо, и нам больше не надо писать проверку для каждого вызова функции `framed_area()`. Для полезной функции, которая раз 500 вызывается в крупной программе, это большое преимущество. Более того, если обработка ошибок по какой-то причине изменится, нам будет достаточно изменить код только в одном месте.

Отметим нечто интересное: мы почти бессознательно заменили подход "проверять аргументы должен вызывающий код" подходом "свои аргументы должна проверять сама функция". Одним из преимуществ второго подхода является то, что проверка аргументов осуществляется в единственном месте. Теперь в случае чего необязательно искать вызовы функции по всей программе. Более того, проверка производится именно там, где эти аргументы используются, поэтому мы имеем всю информацию, необходимую для проверки.

Давайте применим это решение к функции **area()**.

```
int area(int length, int width) // Площадь прямоугольника
{
    if (length<=0 || width <=0)
        error("Неположительный аргумент area()");
    return length*width;
}
```

Этот код будет перехватывать все ошибки в вызовах функции **area()**, поэтому теперь их необязательно проверять в функции **framed_area()**. Однако вы можете получить большее — более конкретные сообщения об ошибках.

Проверка аргументов в функции выглядит настолько простой, что становится непонятным, почему люди не проводят ее постоянно? Одна из причин — пренебрежение ошибками, вторая — неряшливость при написании программ, но существуют и более уважительные причины.

- *Мы не можем модифицировать определение функции.* Функция находится в библиотеке, которую по тем или иным причинам невозможно изменить. Возможно, она будет использована другими людьми, не разделяющими вашего подхода к обработке ошибок. Возможно, она принадлежит кому-то еще, и вы не имеете доступа к ее исходному коду. Возможно, она включена в постоянно обновляющуюся библиотеку, так что, изменив эту функцию, вы будете вынуждены изменять ее в каждой новой версии.
- *Вызываемая функция не знает, что делать при выявлении ошибки.* Эта ситуация типична для библиотечных функций. Автор библиотеки может выявить ошибку, но только вы знаете, что в таком случае следует делать.
- *Вызываемая функция не знает, откуда ее вызвали.* Получив сообщение об ошибке, вы понимаете, что произошло нечто непредвиденное, но не можете знать, как именно выполняемая программа оказалась в данной точке. Иногда необходимо, чтобы сообщение было более конкретным.
- *Производительность.* Для небольшой функции стоимость проверки может перевесить стоимость вычисления самого результата. Например, в случае с функцией **area()** проверка вдвое увеличивает ее размер (т.е. удваивает количество машинных инструкций, которые необходимо выполнить, а не просто длину исходного кода). В некоторых программах этот факт может оказаться критически важным, особенно если одна и та же информация проверяется постоянно функциями, вызывающими одна другую, и передающими при этом информацию более или менее неизменной.

Итак, что делать? Проверять аргументы в функции, если у вас нет веских причин поступать иначе.

После обсуждения некоторых тем, связанных с этим вопросом, мы вернемся к нему в разделе 5.10.

5.5.3. Сообщения об ошибках

Рассмотрим немного иной вопрос: что делать, если вы проверили набор аргументов и обнаружили ошибку? Иногда можно вернуть сообщение "Неправильное значение". Рассмотрим пример.

```
// Попросим пользователя ввести "да" или "нет";
// Символ 'b' означает неверный ответ (т.е. ни "да", ни "нет")
char ask_user(string question)
{
    cout << question << "? (введите y или n)\n";
    string answer = " ";
    cin >> answer;
    if (answer =="y" || answer=="yes") return 'y';
    if (answer =="n" || answer=="no" ) return 'n';
    return 'b'; // 'b' для некорректного ответа
}

// Вычисляет площадь прямоугольника;
// возвращает -1, если аргумент некорректен
int area(int length, int width)
{
    if (length<=0 || width <=0) return -1;
    return length*width;
}
```

На этот раз мы можем поручить детальную проверку вызывающей функции, оставив каждой вызывающей функции возможность обрабатывать ошибки по-своему. Этот подход кажется разумным, но существует множество проблем, которые во многих ситуациях делают его бесполезным.

- Теперь проверку должны осуществлять и вызываемая функция, и все вызывающие функции. Вызывающая функция должна провести лишь самую простую проверку, но вы все равно должны написать этот код и решить, что делать в случае ошибки.
- В вызывающей функции проверка может оказаться забытой, что приведет к непредсказуемым последствиям при дальнейшей работе программы.
- Многие функции не имеют возможности возвращать дополнительные значения, чтобы сообщить об ошибке. Например, функция, считывающая целое число из потока ввода (скажем, оператор >> потока

cin), может возвращать любое значение типа int, поэтому не существует такого значения, которое можно было бы использовать в качестве индикатора ошибки.

Вторая из описанных выше ситуаций, в которой проверка в вызывающем коде не выполняется, может легко привести к неожиданностям. Рассмотрим пример.

```
int f(int x, int y, int z)
{
    int area1 = area(x,y);
    if (area1<=0) error("Неположительная площадь");
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;
    // ...
}
```

Вы заметили ошибку? Такие ошибки трудно выявить, так как в исходном тексте нет очевидного "неверного кода": ошибка заключается в отсутствии некоторого исходного текста.

ПОПРОБУЙТЕ

Выполните эту программу при разных значениях. Выведите на экран значения переменных area1, area2, area3 и ratio. Вставьте в программу больше проверок разных ошибок. Вы уверены, что перехватываете все возможные ошибки? Это вопрос без подвоха; в данном конкретном примере можно доказать перехват всех возможных ошибок.

Существует другой способ решить описанную проблему: использовать исключения (exceptions).

5.6. Исключения

Как и в большинстве языков программирования, в языке C++ существует механизм обработки ошибок: исключения. Основная идея исключений состоит в отделении выявления ошибки (что можно сделать в вызываемой функции) от ее обработки (что можно сделать в вызывающей функции), чтобы гарантировать, что ни одна выявленная ошибка не останется необработанной. Иначе говоря, исключения предоставляют механизм, позволяющий сочетать наилучшие подходы к обработке ошибок, исследованные нами до сих пор. Какой бы легкой ни была обработка ошибок, исключения сделают ее еще легче.

Основная идея заключается в следующем: если функция обнаруживает ошибку, которую не может обработать, она не выполняет оператор **return**, как обычно, а генерирует исключение с помощью инструкции **throw**, показывая, что произошло нечто неправильное. Любая функция, прямо или косвенно вызывающая данную функцию, может перехватить созданное исключение с помощью конструкции **catch**, т.е. указать, что следует делать, если вызываемый код вызвал инструкцию **throw**. Функция выражает свою заинтересованность в перехвате исключений с помощью блока **try** (мы опишем его в следующих разделах), перечисляя виды исключений, которые она планирует обрабатывать в своих разделах **catch** блока **try**. Если ни одна из вызывающих функций не перехватила исключение, то программа прекращает работу.

Мы еще вернемся к исключениям позже (в главе 19), чтобы использовать их немного более сложным способом.

5.6.1. Неправильные аргументы

Рассмотрим вариант функции **area()** с использованием исключений.

```
class Bad_area { }; // Тип, созданный специально для
                    // сообщений об ошибках в функции area()

// Вычисляет площадь прямоугольника; при некорректном
// аргументе генерирует исключение Bad_area
int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area{};
    return length*width;
}
```

Другими словами, если аргументы корректны, функция, как обычно, возвращает площадь прямоугольника, а если нет, то происходит выход из функции **area()** с помощью инструкции **throw**, в надежде, что соответствующий ответ будет предоставлен в одном из разделов **catch**. **Bad_area** — это новый тип, который мы определили только для генерации исключений в функции **area()**, с тем чтобы один из разделов **catch** мог распознать его как исключение, сгенерированное функцией **area()**. Типы, определенные пользователями (классы и перечисления), обсуждаются в главе 9. Запись **Bad_area{}** означает "Создать объект типа **Bad_area** со значением по умолчанию", а выражение **throw Bad_area{}** означает "Создать объект типа **Bad_area** и сгенерировать соответствующее исключение".

Теперь мы можем написать

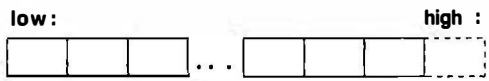
```
int main()
try {
    int x = -1;
    int y = 2;
```

```
    int z = 4;
    // ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = area1/area3;
} catch (Bad_area) {
    cout << "Неверный аргумент функции area()\n";
}
```

Во-первых, заметим, что в этом фрагменте программы обрабатываются все вызовы функции `area()` — как вызов из функции `main()`, так и два вызова из функции `framed_area()`. Во-вторых, обработка ошибок четко отделена от их выявления: функция `main()` ничего не знает о том, какая функция выполнила инструкцию `throw Bad_area{}`, а функция `area()` ничего не знает о том, какая функция (если таковая существует) перехватит исключения `Bad_area`, которые она генерирует. Такое разделение особенно важно в крупных программах, написанных с использованием многочисленных библиотек. В таких программах никто не сможет обработать ошибку, помещая некоторый код там, где это необходимо — поскольку никто не захочет полностью менять код одновременно в приложении и во всех библиотеках.

5.6.2. Ошибки, связанные с диапазоном

Большинство реальных программ работает с наборами данных. Иначе говоря, они используют разнообразные таблицы, списки и другие коллекции данных. В контексте языка C++ такие коллекции данных часто называют *контейнерами* (container). Наиболее часто используемым контейнером стандартной библиотеки является тип `vector`, введенный в разделе 4.6. Объект типа `vector` хранит некоторое количество элементов, которое можно узнать с помощью его функции-члена `size()`. Что произойдет, если мы попытаемся использовать элемент с индексом, не принадлежащим допустимому диапазону `[0,v.size())`? Обычное обозначение `[low,high)` означает, что индексы могут принимать значения от `low` до `high-1`, т.е. включая нижнюю границу, но исключая верхнюю.



Прежде чем ответить на этот вопрос, необходимо ответить на другой:

> "Почему это может произойти?" В конце концов, известно, что индекс вектора `v` должен находиться в диапазоне `[0,v.size())`, поэтому достаточно просто убедиться в этом!

Легко сказать, но трудно сделать. Рассмотрим следующую вполне разумную программу:

```
vector<int> v;                      // Вектор целых чисел
for(int i; cin>>i;)
    v.push_back(i);                 // Чтение в контейнер
for(int i = 0; i<=v.size(); ++i) // Вывод значений
    cout << "v[" << i <<"] == " << v[i] << endl;
```

Видите ошибку? Попытайтесь найти ее, прежде чем двигаться дальше. Это не такая уж редкая ошибка. Мы часто допускаем ее, особенно если программируем поздно ночью, когда устали. Ошибки, как правило, являются результатом спешки или усталости. Мы использовали `0` и `size()`, чтобы попытаться гарантировать, что индекс `i` всегда будет находиться в допустимом диапазоне, когда мы обратимся к элементу `v[i]`.

К сожалению, мы сделали ошибку. Посмотрите на цикл `for`: условие его завершения сформулировано как `i<=v.size()`, в то время как правильно было бы написать `i<v.size()`. В результате, прочитав пять чисел, мы попытаемся вывести шесть. Мы попытаемся обратиться к элементу `v[5]`, индекс которого ссылается за пределы вектора. Эта разновидность ошибок настолько широко известна, что даже получила несколько названий: *ошибка промаха на единицу* (off-by-one error) или, в более общем случае, *ошибка диапазона* (range error), так как индекс не принадлежит допустимому диапазону вектора, и *ошибка выхода за границу* (bounds error), поскольку индекс выходит за границы вектора.

Почему мы не используем для записи такого цикла цикл `for` для диапазона? Да, в этом случае мы не можем получить обращение к неверному элементу в цикле. Однако в данном цикле мы хотели выводить значения не только элементов, но и их индексов. Цикл `for` для диапазона не дает нам такой возможности без дополнительных усилий.

Эту ошибку можно получить еще проще.

```
vector<int> v(5);
int x = v[5];
```

Однако мы сомневаемся, что вы признаете такой пример реалистичным и заслуживающим внимания.

Так что же произойдет на самом деле, если мы допустим такую ошибку диапазона? Операция доступа по индексу в классе `vector` знает размер вектора, поэтому может проверить его (и действительно делает это; см. разделы 4.6 и 19.4). Если проверка заканчивается неудачно, операция доступа по индексу генерирует исключение типа `out_of_range`. Итак, если бы ошибочный код, приведенный выше, являлся частью какой-то программы, перехватывающей исключения, то мы получили бы соответствующее сообщение об ошибке.

```
int main()
try {
    vector<int> v;          // Вектор значений типа int
    for (int x; cin>>x; )
        v.push_back(x);     // Чтение значений в вектор
    for (int i = 0; i<=v.size(); ++i) // Вывод значений
        cout << "v[" << i <<"] == " << v[i] << '\n';
} catch (out_of_range) {
    cerr << "Ошибка диапазона!\n";
    return 1;
} catch (...) {   // Перехват всех остальных исключений
    cerr << "Исключение: что-то пошло не так\n";
    return 2;
}
```

Обратите внимание на то, что ошибка диапазона на самом деле является частным случаем ошибки, связанной с аргументами, которую мы обсуждали в разделе 5.5.2. Не доверяя себе, мы поручили проверку диапазона индексов вектора самой операции доступа по индексу. По очевидным причинам оператор доступа по индексу (`vector::operator[]`) сообщает об ошибке, генерируя исключение[1]. Что же еще может произойти? Оператор доступа по индексу не имеет представления о том, какие действия устроили бы нас в этой ситуации. Автор класса `vector` даже не знает, частью какой программы может стать его код.

5.6.3. Неправильный ввод

Обсуждение действий, которые следует предпринять при неправильном вводе данных, мы отложим до раздела 10.6. Пока лишь отметим, что при обнаружении ошибок ввода используются те же методы и механизмы языка программирования, что и при обработке ошибок, связанных с неправильными аргументами и выходом за пределы допустимого диапазона. Здесь мы лишь покажем, как поступать, если операции ввода прошли успешно. Рассмотрим фрагмент кода, в котором вводится число с плавающей точкой.

```
double d = 0;
cin >> d;
```

Мы можем проверить, успешной ли оказалась последняя операция, проверяя поток `cin`.

---

[1] Вообще говоря, `vector<>::operator[]` для повышения производительности как раз ***не*** генерирует исключений, полностью полагаясь на корректность передаваемого ему аргумента; проверка диапазона (и, соответственно, генерация исключений) выполняется функцией-членом `vector<>::at()`. Однако в заголовочном файле `std_lib_facilities.h` этот класс видоизменен таким образом, чтобы `operator[]` проверял выход за границы диапазона и генерировал исключение. Подробнее об этом можно прочесть в главе 19. — *Примеч. ред.*

```
if (cin) {
    // Все хорошо, можно считывать данные дальше
}
else {
    // Последнее считывание не было выполнено,
    // поэтому следует что-то
}
```

Существует несколько возможных причин сбоя при вводе данных. Одна из них — когда тип данных, которые мы пытаемся считать, отличается от типа **double**.

На ранних стадиях разработки мы часто хотим просто сообщить, что нашли ошибку и прекратить выполнение программы, потому что еще не знаем, как на нее реагировать. Иногда мы впоследствии возвращаемся к этому фрагменту и уточняем свои действия. Рассмотрим пример.

```
double some_function()
{
    double d = 0;
    cin >> d;
    if (!cin)
        error("Не могу считать число double"
              " в 'some_function()' ");
    // Делаем что-то полезное
}
```

Условие **!cin** ("не **cin**", т.е. **cin** не в хорошем состоянии) означает, что предыдущая операция с **cin** была неудачной.

Строку, переданную функции **error()**, можно вывести на печать для облегчения отладки или для передачи сообщения пользователю. Как написать функцию **error()** так, чтобы она оказалась полезной для многих программ? Она не может возвращать никакого значения, потому что неизвестно, что с ним делать дальше. Вместо этого лучше, чтобы функция **error()** прекращала выполнение программы после получения сообщения об ошибке. Кроме того, перед выходом иногда следует выполнить определенные несложные действия, например оставить окно с сообщением активным на достаточно длительное время, чтобы пользователь мог прочесть сообщение. Все это — очевидная работа для исключений (подробнее об этом — в разделе 7.3).

В стандартной библиотеке определено несколько типов исключений, таких как **out_of_range**, генерируемое классом **vector**. Кроме того, в этой библиотеке есть исключение **runtime_error**, идеально подходящее для наших нужд, поскольку в нем хранится строка, которую может использовать обработчик ошибки. Так что нашу простую функцию **error()** можно переписать следующим образом:

```
void error(string s)
{
    throw runtime_error(s);
}
```

Когда нам потребуется работать с исключением **runtime_error**, мы просто перехватим его. Для простых программ идеальным является перехват исключения **runtime_error** в функции **main()**.

```
int main()
try {
    // Наша программа
    return 0;     // 0 указывает на успех
}
catch (runtime_error& e) {
      cerr << "runtime error: " << e.what() << '\n';
      keep_window_open();
      return 1;  // 1 указывает на ошибку
}
```

Вызов **e.what()** извлекает сообщение об ошибке из исключения **runtime_error**. Символ **&** в выражении

```
catch(runtime_error& e) {
```

означает, что мы хотим передать исключение по ссылке. Пожалуйста, пока рассматривайте это выражение просто как не имеющую отношения к делу техническую подробность. В разделах 8.5.4–8.5.6 мы объясним, что означает передача сущности по ссылке.

Обратите внимание на то, что для выдачи сообщений об ошибках мы использовали поток **cerr**. Этот поток очень похож на поток **cout**, за исключением того, что он предназначен для вывода сообщений об ошибках. По умолчанию потоки **cerr** и **cout** выводят данные на экран, но поток **cerr** не оптимизирован и не буферизован, поэтому более устойчив к ошибкам и в некоторых операционных системах может быть перенаправлен в другое устройство, например в файл. Поток **cerr** позволяет отделить документирование ошибок от обычного вывода на экран, и потому для вывода ошибок мы используем именно его.

Исключение **out_of_range** отличается от исключения **runtime_error**, поэтому перехват исключения **runtime_error** не приводит к обработке ошибок **out_of_range**, которые могут возникнуть при неправильном использовании класса **vector** или других контейнерных типов из стандартной библиотеки. Однако и **out_of_range**, и **runtime_error** являются "исключениями", поэтому для работы с ними обоими можно предусмотреть перехват объекта класса **exception**.

```
int main()
try {
    // Наша программа
```

```
    return 0;           // 0 указывает на успех
}
catch (exception& e) {
     cerr << "error: " << e.what() << '\n';
     keep_window_open();
     return 1;          // 1 указывает на ошибку
}
catch (...) {
     cerr << "Неизвестное исключение\n";
     keep_window_open();
     return 2;          // 2 указывает на ошибку
}
```

Здесь для того чтобы перехватить исключения любых типов, мы добавили инструкцию `catch(...)`.

Когда исключения обоих типов (`out_of_range` и `runtime_error`) рассматриваются как разновидности одного и того же типа `exception`, говорят, что тип `exception` является базовым типом (супертипом) для них обоих. Этот исключительно полезный и мощный механизм будет описан в главах 13–16.

Снова обращаем ваше внимание на то, что значение, возвращаемое функцией `main()`, передается "системе", вызвавшей программу. Одни системы (такие, как Unix) часто используют это значения, а другие (такие как Windows) зачастую их игнорируют. Нуль означает, что программа завершилась успешно, а ненулевое значение, возвращенное функцией `main()`, означает какой-то сбой.

При использовании функции `error()` для описания возникшей проблемы часто необходимо передать не одну, а две порции информации. В данном случае эти две порции просто объединяются в одну строку. Этот прием настолько широко распространен, что мы решили представить его в виде второго варианта функции `error()`.

```
void error(string s1, string s2)
{
    throw runtime_error(s1+s2);
}
```

Этой простой обработки ошибки нам будет достаточно, пока ситуация не усложнится и потребуется придумать более изощренный способ исправить ситуацию. Обратите внимание на то, что использование функции `error()` не зависит от количества предыдущих вызовов функций: функция `error()` всегда находит ближайший раздел `catch`, в котором предусмотрен перехват исключения `runtime_error` (обычно — в функции `main()`). Примеры использования исключений и функции `error()` приведены в разделах 7.3 и 7.7. Если исключение осталось неперехваченным, то вы получите системную ошибку по умолчанию (ошибка "неперехваченного исключения").

ПОПРОБУЙТЕ

Для того чтобы увидеть неперехваченное исключение в действии, запустите небольшую программу, в которой используется функция `error()` и при этом не перехватываются никакие исключения.

5.6.4. Сужающие преобразования

В разделе 3.9.2 продемонстрирована ужасная ошибка: когда мы присвоили переменной слишком большое значение, оно было просто усечено. Рассмотрим пример.

```
int x = 2.9;
char c = 1066;
```

Здесь `x` получит значение `2`, а не `2.9`, поскольку переменная `x` имеет тип `int`, а значения типа `int` не могут иметь дробных частей. Аналогично, если используется обычный набор символов ASCII, то переменная `c` получит значение 42 (что соответствует символу `*`), а не `1066`, поскольку переменные типа `char` не могут принимать такие большие значения.

В разделе 3.9.2 показано, как защититься от такого сужения путем проверки. С помощью исключений (и шаблонов; см. раздел 19.3) можно написать функцию, проверяющую и генерирующую исключение `runtime_exception`, если присваивание или инициализация может привести к изменению значения. Рассмотрим пример.

```
int x1 = narrow_cast<int>(2.9);     // Генерирует исключение
int x2 = narrow_cast<int>(2.0);     // OK
char c1 = narrow_cast<char>(1066); // Генерирует исключение
char c2 = narrow_cast<char>(85);    // OK
```

Угловые скобки `<...>` означают то же самое, что и в выражении `vector<int>`. Они используются, когда для выражения идеи возникает необходимость указать тип, а не значение. Аргументы, стоящие в угловых скобках, называют *аргументами шаблона* (template arguments). Если необходимо преобразовать значение и мы не уверены, что оно "поместится" в целевом типе, можно использовать шаблонную функцию `narrow_cast`, определенную в заголовочном файле `std_lib_facilities.h` и реализованную с помощью функции `error()`. Слово *приведение* (cast) означает преобразование типа и отражает роль этой операции в ситуации, когда что-то "сломано" (по аналогии с гипсовой повязкой на сломанной ноге[2]). Обратите внимание на то, что приведение типа не изменяет операнд, а создает

[2] Одно из значений слова *cast* в английском языке — "гипсовый слепок", "гипсовая повязка". — *Примеч. пер.*

новое значение, имеющее тип, указанный в угловых скобках, и соответствующее значению операнда.

5.7. Логические ошибки

После устранения ошибок, выявленных компилятором и редактором связей, программа выполняется. Как правило, после этого программа либо ничего не выдает на печать, либо выдает неверные результаты. Это может происходить по многим причинам. Возможно, вы просто неправильно поняли логику работы; написали не то, что намеревались; сделали глупую ошибку в какой-нибудь инструкции `if` или что-нибудь еще. Логические ошибки обычно труднее всего находить и исправлять, поскольку на этой стадии компьютер делает только то, что вы сами ему приказали. Теперь ваша задача — выяснить, почему он делает не то, что вы хотели. В принципе, компьютер — это очень быстро действующий болван. Он просто покорно делает в точности то, что вы ему сказали.

Попробуем проиллюстрировать сказанное на простом примере. Рассмотрим программу-код для поиска минимальной, максимальной и средней температур в наборе данных.

```
int main()
{
    vector<double> temps;           // Вектор температур

    for (double temp; cin>>temp; ) // Чтение температур
        temps.push_back(temp);      // в вектор

    double sum       = 0;
    double high_temp = 0;
    double low_temp  = 0;

    for (int x : temps)
    {
        if(x > high_temp) high_temp = x;// Поиск максимума
        if(x < low_temp) low_temp = x;  // Поиск минимума
        sum += x;                       // Вычисление суммы
    }

    cout << "Максимальная температура: "
         << high_temp << "\n";
    cout << "Минимальная температура: "
         << low_temp << "\n";
    cout << "Средняя температура:"
         << sum/temps.size() << "\n";
}
```

Мы проверили эту программу, введя почасовые данные о температуре в центре Люббока, штат Техас (Lubbock, Texas), 16 февраля 2005 года (в штате Техас все еще используется шкала Фаренгейта).

```
-16.5, -23.2, -24.0, -25.7, -26.1, -10.6, -9.7,  -2.4,
  7.5,  12.6,  23.8,  25.3,  28.0,  34.8, 36.7,  41.5,
 40.3,  42.6,  39.7,  35.4,  12.6,   6.5, -3.7, -14.3
```

Результаты оказались следующими:

```
Максимальная температура: 42.6
Минимальная температура: -26.1
Средняя температура: 9.3
```

Наивный программист может прийти к выводу, что программа работает просто отлично. Безответственный программист продаст ее заказчику. Благоразумный программист проверит программу еще раз. Поступим благоразумно и воспользуемся данными, полученными 23 июля 2005 года.

```
 76.5,  73.5, 71.0, 73.6, 70.1,  73.5,  77.6,  85.3,
 88.5,  91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,
110.2, 103.6, 94.9, 91.7, 88.4,  85.2,  85.4,  87.7
```

На этот раз результаты таковы:

```
Максимальная температура: 112.4
Минимальная температура: 0.0
Средняя температура: 89.2
```

Что-то не то. Мороз (0.0°F соответствует примерно –18°C) в Люббоке в июле — это же просто конец света! Вы уже нашли ошибку? Поскольку переменная `low_temp` была инициализирована значением `0.0`, она останется равной нулю, если только не встретится хотя бы одно отрицательное значение температуры.

ПОПРОБУЙТЕ

Выполните эту программу. Убедитесь, что она действительно выдает неверные результаты. Попробуйте ее "сломать" (т.е. вынудить выдать неправильные результаты), введя другой набор данных. Каково наименьшее количество входных данных вам для этого потребуется?

К сожалению, в этой программе есть и другие ошибки. Что произойдет, если все значения температуры окажутся отрицательными? Инициализация переменной `high_temp` создает аналогичную проблему: она останется равной нулю, если в исходных данных не встретится хотя бы одно значение температуры, большее нуля. Не пытайтесь использовать эту программу зимой на Северном полюсе!

Такие ошибки достаточно типичны; они не создают никаких проблем при компиляции и не приводят к неправильным ответам при "разумных" условиях. Однако мы забыли указать, что надо понимать под "разумными" условиями. Вот как должна выглядеть исправленная программа.

```
int main()
{
    vector<double> temps;       // Вектор температур

    double sum       = 0;
    double high_temp = -1000;  // Инициализация невозможно
                               // низким значением
    double low_temp  =  1000;  // Инициализация невозможно
                               // высоким значением
    int no_of_temps  = 0;

    for (double temp; cin>>temp; ) { // Чтение температуры
        ++no_of_temps;               // Количество данных
        sum += temp;                 // Вычисление суммы
        if(x > high_temp) high_temp = x; // Поиск максимума
        if(x < low_temp) low_temp = x;   // Поиск минимума
    }

    cout << "Максимальная температура: "
         << high_temp << "\n";
    cout << "Минимальная температура: "
         << low_temp << "\n";
    cout << "Средняя температура:"
         << sum/no_of_temps << "\n";
}
```

Эта программа работает? Почему вы уверены в этом? Вы сможете дать точное определение слова "работает"? Откуда взялись числа 1000 и -1000? Вспомните о "магических" константах (раздел 5.5.1). Указывать числа 1000 и -1000 как литеральные константы в тексте программы — плохой стиль, но, кроме того, может быть, и эти числа неверны? Нет ли мест, где температура опускается ниже —1000°F (–573°C)? Существуют ли места, где температура поднимается выше 1000°F (538°C)?

ПОПРОБУЙТЕ

Просмотрите программу. Используя достоверные источники информации, выберите разумные значения для констант `min_temp` ("минимальная температура") и `max_temp` ("максимальная температура"). Эти значения определят пределы применимости вашей программы.

5.8. Оценка

Представьте себе, что вы написали простую программу, например, вычисляющую площадь правильного шестиугольника с длиной стороны, скажем, равной `2`. Вы запустили ее и получили, что площадь равна `-34.56`. Очевидно, что ответ неверен. Почему? Потому что ни одна фигура не может иметь отрицательную площадь. Так что вы исправляете ошибку (которую, как вам кажется, вы нашли) и получаете новый ответ — `21.65685`. Этот результат правильный? Трудно сказать, потому что мы обычно не помним формулу для вычисления площади шестиугольников. Чтобы не опозориться перед пользователями и не передать им программу, выдающую глупые результаты, необходимо убедиться, что ответ правильный. В данном конкретном случае это просто. Шестиугольник не так далек от квадрата. Набросав на бумаге рисунок, легко убедиться, что площадь шестиугольника недалека от площади квадрата 3×3, которая равна `9`. Итак, ответ `21.65685` не может быть правильным! Переделываем программу еще раз и получаем ответ `10.3923`. Это уже похоже на правду!

Приведенный параграф не о том, как находить площадь шестиугольников. Главная его идея в том, что если у нас нет представления о правильном ответе, пусть даже очень приближенном, мы не имеем права считать результат приемлемым. Всегда следует задавать самому себе следующий вопрос.

1. Является ли данный ответ для данной задачи правдоподобным?

Следует также задать себе более общий (и более трудный) вопрос.

2. Как распознать правдоподобный результат?

Обратите внимание на то, что мы не спрашиваем "Каков точный ответ?" или "Каков правильный ответ?" Этот ответ нам даст сама программа. Все, что мы хотим, — знать, что ответ не является глупым. Только в том случае, если ответ правдоподобен, имеет смысл продолжать работу над программой.

Оценка — это замечательное искусство, сочетающее в себе здравый смысл и очень простую арифметику. Некоторые люди легко выполняют оценку умозрительно, но мы предпочитаем "рисовать каракули на обратной стороне конверта", поскольку в противном случае легко сделать ошибку. В данной книге под оценкой мы понимаем неформальный набор приемов, или *прикидку* (guesstimation[3]), сочетающую в себе интуитивную догадку и немного вычислений.

[3] Непереводимая игра слов: слово *guesstimation* образовано из слов *guess* ("догадка") и *estimation* — ("оценка"). — *Примеч. пер.*

ПОПРОБУЙТЕ

Длина стороны нашего правильного шестиугольника равна 2 см. Получили ли мы правильный ответ? Просто выполните прикидочные вычисления. Возьмите лист бумаги и набросайте эскиз. Не считайте это занятием ниже своего достоинства. Многие знаменитые ученые восхищали людей своими способностями получать примерный ответ с помощью карандаша и клочка бумаги (или салфетки). Эта способность (на самом деле — простая привычка) может сэкономить массу времени и помочь избежать массы неприятностей.

Часто оценка связана с предварительным анализом данных, необходимых для вычисления, но не имеющихся в наличии. Представьте, что вы протестировали программу, оценивающую время путешествия из одного города в другой. Правдоподобно ли, что из Москвы до Сочи можно доехать на автомобиле за 9 часов? А из Киева до Запорожья? Почему "нет" и почему "да"? На каких данных основана ваша догадка об ответах на эти вопросы? Часто на помощь приходит быстрый поиск в вебе. Например, 2000 км — это вполне правдоподобная оценка расстояния между Москвой и Сочи. Было бы трудно (да и незаконно) поддерживать среднюю скорость 200 км/ч (200·9 не сильно меньше 2000), чтобы добраться из Москвы в Сочи за 9 часов. Можете проверить сами: мы переоценили и расстояние, и среднюю скорость, но наша оценка правдоподобности ответа вполне обоснована.

ПОПРОБУЙТЕ

Оцените реальное время поездки на автомобиле, а также время перелета между указанными городами на обычных коммерческих авиарейсах. Затем попытайтесь проверить ваши оценки, используя такие источники, как карты и расписания авиарейсов.

5.9. Отладка

Написанная (вчерне?) программа всегда содержит ошибки. Бывает, что небольшие программы (случайно) компилируются и выполняются правильно с первой же попытки. Но если это происходит с любой не тривиальной программой, то это очень и очень подозрительно. Если программа *правильно* выполнилась с первой попытки, идите к друзьям и празднуйте, поскольку Новый год — праздник более частый.

Итак, написав некоторую программу, вы должны найти и удалить из нее ошибки. Этот процесс обычно называют *отладкой* (debugging), а ошибки — *жучками* (bugs). Говорят, что термин *жучок* возник в те

времена, когда аппаратное обеспечение выходило из строя из-за насекомых, случайно заблудившихся среди электронных ламп и реле, заполнявших комнаты. Некоторые приписывают этот термин Грейс Мюррей Хоппер (Grace Murray Hopper), создателю языка программирования COBOL (раздел 22.2.2.2). Но кто бы ни придумал этот термин более 50 лет назад, ошибки в программах неизбежны и повсеместны. Их поиск и устранение называют *отладкой* (debugging).

Отладка выглядит примерно так.

1. Компилируем программу.
2. Редактируем связи.
3. Выполняем программу и проверяем, делает ли она то, для чего предназначена.

Обычно эта последовательность операций выполняется снова и снова: для действительно крупных программ этот процесс повторяется сотни и тысячи раз, часто годами. Каждый раз что-то работает не так, как ожидалось, и приходится исправлять какую-то ошибку. Я считаю отладку наиболее утомительной и затратной по времени частью программирования и трачу много времени во время проектирования и программирования на то, чтобы минимизировать количество времени, затрачиваемого на отладку. Другие же считают, что отладка — это захватывающее занятие, сама суть программирования, которое затягивает, как видеоигры, и удерживает программиста у компьютера многие дни и ночи (это я могу засвидетельствовать и на собственном опыте).

Вот как ***не*** надо отлаживать программы.

```
while (программа не выглядит работоспособной) { // Псевдокод
    Бегло просматриваем программу в поисках странностей
    Изменяем их так, чтобы программа выглядела лучше
}
```

Почему мы упоминаем об этом? Описанный плохой алгоритм отладки слабо гарантирует успех. К сожалению, это описание — не совсем карикатура. Именно так поступают многие люди, допоздна засиживающиеся за компьютером и ощущающие собственную неполноценность.

Основной вопрос отладки звучит так:

Как понять, что программа действительно работает правильно?

При невозможности ответить на этот вопрос вы обречены на долгую и утомительную отладку, а ваши пользователи, скорее всего, будут вынуждены долго и разочарованно ждать, когда же вы ее закончите. Мы возвращаемся к этому вопросу, потому что все, что помогает ответить на него, способствует минимизации отладки и помогает создавать правильные и

удобные в эксплуатации программы. В принципе, программировать надо так, чтобы жучкам было негде спрятаться. Разумеется, мы хотим слишком многого, но наша цель — структурировать программу так, чтобы минимизировать вероятность ошибок и максимально увеличить вероятность их обнаружения.

5.9.1. Практические советы по отладке

Подумайте об отладке, прежде чем напишете первую строку своей программы. Когда написано много строк, уже слишком поздно пытаться упростить отладку.

Решите, как сообщать об ошибках. По умолчанию в данной книге принят следующий принцип: "Использовать функцию `error()` и перехватывать `exception` в функции `main()`".

Добивайтесь, чтобы программу было легко читать, чтобы в результате все ошибки просто бросались в глаза.

- Хорошо комментируйте свою программу. Это не означает просто добавить комментариев побольше! Вы не сможете сформулировать смысл операции на естественном языке лучше, чем на языке программирования. В комментариях следует ясно и коротко указать то, что невозможно выразить в коде.
  - Название программы.
  - Цель программы.
  - Кто написал код и когда.
  - Номера версий.
  - Какие фрагменты кода могут вызвать сложности.
  - Основные идеи проекта.
  - Как организован код.
  - Какие сделаны предположения о вводных данных.
  - Каких фрагментов кода пока не хватает и какие варианты еще не обработаны.
- Используйте осмысленные имена.
  - Это не означает, что вы должны использовать длинные имена!.
- Используйте последовательную схему кода.
  - Ваша интегрированная среда программирования может помочь, но она не может сделать вместо вас всю работу.
  - Стиль, принятый в книге, может послужить для вас отправной точкой.

- Разбивайте программу на небольшие фрагменты, каждый из которых выражает определенные логически выделенные действия.
  - Старайтесь, чтобы размер функции не превышал страницы или двух; подавляющее большинство функций должно быть намного короче.
- Избегайте сложных выражений.
  - Пытайтесь избегать вложенных циклов, вложенных инструкций **if**, сложных условий и т.д. К сожалению, иногда они необходимы, но не забывайте, что ошибки легче всего прячутся в сложном коде.
- Где только можно, используйте библиотечные функции, а не собственный код.
  - Библиотеки, как правило, лучше продуманы и протестированы, чем код, который вы напишете сами.

Пока все эти советы звучат довольно абстрактно, но скоро мы покажем примеры их применения.

Скомпилируйте программу. Разумеется, для этого понадобится компилятор. Его сообщения об ошибках обычно весьма полезны (даже если нам кажется, что они могли бы быть еще полезнее), и, если вы не профессионал, то должны считать, что компилятор всегда прав. Если же вы на самом деле эксперт, то закройте книгу — она написана не для вас. Иногда программисту кажется, что правила компилятора слишком тупые и слишком строгие (это не так за редчайшими исключениями), и многие вещи можно было бы сделать проще (как бы не так). Но, как говорится, "свой инструмент ругает только плохой мастер". Хороший мастер знает сильные и слабые стороны своего инструмента и соответственно его настраивает. Рассмотрим наиболее распространенные ошибки времени компиляции.

- Закрыты ли кавычки всех строковых литералов?

```
cout << "Привет, << name << '\n'; // Ой!
```

- Закрыты ли кавычки всех символьных литералов?

```
cout << "Привет, " << name << '\n; // Ой!
```

- Закрыты ли фигурные скобки всех блоков?

```
int f(int a)
{
    if (a>0) { /* что-то делаем */
    else     { /* делаем что-то еще */ }
} // Ой!
```

- Совпадает ли количество открывающих и закрывающих скобок?

```
if (a<=0 // Ой!
    x = f(y);
```

- Компилятор обычно сообщает об этих ошибках "поздно"; он просто не знает, что вы имели в виду, когда забыли поставить закрывающую скобку после нуля.
- Каждое ли имя объявлено?
  - Включены ли все необходимые заголовочные файлы (например, `#include "std_lib_facilities.h"`)?
  - Объявлено ли каждое имя до его использования?
  - Правильно ли набраны все имена?

```
int count; /* . . . */ ++Count;    // Ой!
char ch;   /* . . . */ Cin>>c;     // Ой-ой!
```

- Поставлено ли двоеточие после каждой инструкции?

```
x = sqrt(y)+2 // Ой!
z = x+3;
```

В упражнениях приведено еще больше примеров таких ошибок. Кроме того, не забывайте о классификации ошибок в разделе 5.2.

После того как программа скомпилирована, а ее связи отредактированы, наступает самый трудный этап, на котором необходимо понять, почему программа работает не так, как вы предполагали. Вы смотрите на результаты и пытаетесь понять, как ваша программа могла их вычислить. На самом деле чаще программисты смотрят на пустой экран и гадают, почему их программа ничего не вывела. Обычная проблема с консолью Windows заключается в том, что она исчезает, не дав вам шанса увидеть, что было выведено на экран (если что-то все-таки было выведено). Одно из решений этой проблемы — вызвать функцию `keep_window_open()` из заголовочного файла `std_lib_facilities.h` в конце функции `main()`. В таком случае программа попросит вас ввести что-нибудь перед выходом, и вы сможете просмотреть результаты ее работы до того, как окно закроется.

В поисках ошибок тщательно проверьте инструкцию за инструкцией, начиная с того места, до которого, по вашему мнению, программа работала правильно. Представьте себя компьютером, выполняющим вашу программу. Соответствует ли вывод вашим ожиданиям? Разумеется, нет, иначе вы не занимались бы отладкой.

- Часто, когда программист не видит проблемы, причина заключается в том, что вы "видите" не действительное, а желаемое. Рассмотрим пример.

```
for (int i = 0; i<=max; ++j) { // Ой! (Дважды)
    for (int i=0; 0<max; ++i); // Выводим элементы вектора v
        cout << "v[" << i << "]==" << v[i] << '\n';
    // ...
}
```

- Этот пример заимствован из реальной программы, написанной опытным программистом (я подозреваю, что он писал этот фрагмент глубокой ночью).
- Часто, когда вы не видите проблемы, причина заключается в том, что между точкой, в которой программа еще работала правильно, и следующей точкой, в которой программа выдала неверный ответ, содержится слишком много инструкций (или выводится слишком мало информации). Большинство интегрированных сред программирования допускают пошаговую отладку программ. В конце концов вы научитесь пользоваться этими возможностями, но при отладке простых программ достаточно расставить в нескольких местах дополнительные инструкции вывода (в поток **cerr**), например:

```
int my_fct(int a, double d)
{
    int res = 0;
    cerr << "my_fct(" << a << "," << d << ")\n";
    // ... Плохо себя ведущий код ...
    cerr << "my_fct() возвращает " << res << '\n';
    return res;
}
```

- Вставьте инструкции для проверки инвариантов (т.е. условий, которые всегда должны выполняться; см. раздел 9.4.3) в подозрительные фрагменты кода, например:

```
int my_complicated_function(int a, int b, int c)
// Аргументы являются положительными и a < b < c
{
    if (!(0<a && a<b && b<c)) // ! означает "НЕ", a && — "И"
        error("Неверные аргументы функции mcf");
    // ...
}
```

- Если все это не привело к успеху, вставьте инварианты в разделы программы, которые вы считаете правильными; если вы не можете найти ошибку, вы, определенно, ищете ее не в том месте.

Инструкция для проверки инвариантов называется **assert**.

Достаточно интересно, что существует несколько эффективных методик программирования. Разные люди программируют совершенно по-разному. Многие различия между методами отладки объясняются разнообразием программ, но не меньшее их количество проистекает из разных образов мышления программистов. Следует знать, что наилучшего способа отладки не существует. Просто надо помнить, что запутанный код чаще содержит ошибки. Старайтесь писать программы просто и логично, хорошо их форматируйте, и вы сэкономите массу времени на отладке.

5.10. Пред- и постусловия

Теперь вернемся к вопросу, как поступать с неправильными аргументами функции. Вызов функции — это наилучшая отправная точка для того, чтобы подумать о правильном коде и перехвате ошибок: именно здесь происходит разделение вычислений на логические блоки. Рассмотрим следующий пример, который уже был показан выше:

```
int my_complicated_function(int a, int b, int c)
// Аргументы являются положительными и a < b < c
{
    if (!(0<a && a<b && b<c)) // ! означает "НЕ", a && - "И"
        error("Неверные аргументы функции mcf");
    // ...
}
```

Сначала в комментарии указывается, какие аргументы ожидает функция, а затем происходит проверка этого условия (и генерация исключения, если это условие нарушается). Это правильная стратегия. Требования, которые функция предъявляет к своим аргументам, часто называют *предусловиями* (pre-condition): они должны выполняться, чтобы функция работала правильно. Вопрос заключается в том, что же делать, если предусловия нарушаются. У нас есть две основные возможности.

1. Игнорировать предусловия (надеясь, или предполагая, что все вызывающие функции передают правильные аргументы).
2. Проверять их (и каким-то образом сообщать об ошибке).

С этой точки зрения типы аргументов — это лишь способ проверки простейших предусловий на этапе компиляции. Рассмотрим пример.

```
int x = my_complicated_function(1, 2, "horsefeathers");
```

Здесь компилятор обнаружит, что третий аргумент не является целым числом (нарушение предусловия). По сути, в этом разделе мы говорим о предусловиях, которые не в состоянии проверить компилятор.

Наше предложение заключается в том, чтобы вы всегда документировали предусловия в комментариях (чтобы вызывающий функцию программист мог видеть, что ожидает вызываемая функция). Если функция не содержит комментария, в котором указаны условия, накладываемые на аргументы, будем считать, что она может принимать любые значения аргументов. Но можно ли полагаться на то, что программист, вызывающий функцию, станет читать эти комментарии и придерживаться установленных правил? Иногда да, но правило проверки аргументов в вызывающей функции по сути требует проверки выполнения предусловий в вызываемой функции. Это следует делать всегда, если нет веской причины этого не делать. К таким причинам относятся следующие.

- Никто не может передать неправильные аргументы.
- Проверка слишком сильно замедлит выполнение программы.
- Проверка является слишком сложной.

Первую причину можно признать уважительной, только если вы знаете, кто будет вызывать вашу функцию. В реальном мире это практически невозможно.

Вторая причина является веской намного реже, чем многие полагают, и часто игнорируется как пример преждевременной оптимизации. Проверку всегда можно удалить из программы после ее отладки. Не стоит пренебрегать такими проверками, иначе вас ждут бессонные ночи в поисках ошибок, которые можно было бы предотвратить.

Третья причина является довольно серьезной. Опытный программист может легко привести пример, когда проверка предусловия требует существенно больше работы, чем выполнение самой функции. В качестве примера можно назвать поиск в словаре: предусловием является упорядоченность словаря, но проверка, упорядочен ли словарь, намного более дорогостояща, чем поиск в нем. Иногда предусловие сложно закодировать и правильно выразить. Тем не менее, написав функцию, обязательно подумайте, нельзя ли написать быструю проверку ее предусловий, и напишите ее, если, конечно, у вас нет веских причин этого не делать.

Записав предусловия (даже в виде комментариев), вы значительно повысите качество программы: это заставит вас задуматься о том, какие аргументы требует функция. Если вы не можете просто и ясно сформулировать эти требования в виде комментария, то, вероятно, вы плохо продумали свою программу. Опыт показывает, что запись комментариев с предусловиями и проверок предусловий помогают избежать многих ошибок проектирования. Мы уже упоминали, что ненавидим отладку; ясно сформулированные предусловия позволяют избежать ошибок проектирования, а также отследить некорректное применение функций на ранних стадиях разработки программы. Исходный текст

```
int my_complicated_function(int a, int b, int c)
// Аргументы положительны и a < b < c
{
    if (!(0<a && a<b && b<c)) // ! означает "НЕ", a && – "И"
        error("Неверные аргументы функции mcf");
    // ...
}
```

сэкономит ваше и время и силы по сравнению с более простым вариантом

```
int my_complicated_function(int a, int b, int c)
{
    // ...
}
```

5.10.1. Постусловия

Формулировка предусловий позволяет улучшить структуру программы и отследить некорректное применение функций на ранних этапах. Можно ли использовать эту идею где-нибудь еще? Да, на ум сразу приходит оператор **return**! В конце концов, обычно мы указываем, что функция возвращает; т.е. если мы возвращаем из функции какое-то значение, то *всегда* обещаем выполнение какого-то условия для возвращаемого значения (а как иначе вызывающая функция будет знать, чего ей ждать?). Вернемся к нашей функции из раздела 5.6.1, вычисляющей площадь прямоугольника.

```
// Вычисляет площадь прямоугольника; при некорректных
// аргументах генерирует исключение Bad_area
int area(int length, int width)
{
    if (length<=0 || width <=0) throw Bad_area();
    return length*width;
}
```

Эта функция проверяет предусловия, хотя они и не сформулированы в виде комментария (для такой короткой функции это вполне допустимо), и предполагает, что все вычисления выполняются корректно (для таких тривиальных вычислений это также вполне приемлемо). Тем не менее эту функцию можно написать намного яснее.

```
int area(int length, int width)
// Вычисляет площадь прямоугольника.
// Предусловия: аргументы length и width положительны
// Постусловия: возвращает положительное значение,
// являющееся площадью соответствующего прямоугольника
{
     if (length<=0 || width <=0)
               error("area() pre-condition");
     int a = length*width;
     if (a<=0) error("area() post-condition");
     return a;
}
```

Мы не можем проверить выполнение всех постусловий, но можем проверить хотя бы ту их часть, которая гласит, что возвращаемое значение является положительным.

ПОПРОБУЙТЕ

Найдите пару значений, при которых предусловие выполняется, а постусловие — нет.

Пред- и постусловия обеспечивают базовую проверку разумности кода. Они тесно связаны с понятиями инвариантов (раздел 9.4.3), корректности (разделы 4.2 и 5.2), а также с тестированием (глава 26).

5.11. Тестирование

Как определить, когда следует прекратить отладку? Ясно, что отладка должна выполняться до тех пор, пока мы не найдем все ошибки, или по крайней мере попытаемся это сделать. А как узнать, что мы нашли последнюю ошибку? Никак. "Последняя ошибка" — это шутка программистов: такой ошибки, особенно в большой программе, не существует. В большой программе невозможно найти последнюю ошибку.

Кроме отладки, нам необходим систематический подход к поиску ошибок. Он называется *тестированием* (testing) и рассматривается в разделе 7.3, в упражнениях к главе 10 и в главе 26. В принципе, тестирование — это выполнение программы с большим и систематически подобранным множеством входных данных и сравнение результатов с ожидаемыми. Выполнение программы с заданным множеством входных данных называют *контрольным примером* (test case). Для реальных программ могут потребоваться миллионы контрольных примеров. Систематическое тестирование невозможно выполнить вручную, когда программист набирает данные тест за тестом, поэтому немного позже, через несколько глав, мы рассмотрим инструменты, необходимые для правильного тестирования. А пока что запомните, что тестирование основано на убеждении, что поиск ошибок — это хорошо. Рассмотрим пример.

- Точка зрения 1. Я умнее любой программы! Я разберу этот @#$%^ код на кусочки!
- Точка зрения 2. Я вылизывал эту программу две недели. Она идеальна!

Как вы думаете, кто из этих двух программистов найдет больше ошибок? Разумеется, наилучшим вариантом является опытный программист, с небольшим вкраплением первой точки зрения, который спокойно, хладнокровно, терпеливо и систематически работает с возможными сбоями программы. Хорошие тестеры — на вес золота!

Мы будем пытаться системно подходить к выбору контрольных примеров и всегда испытывать как правильные, так и неправильные входные данные. Первый такой пример будет приведен в разделе 7.3.

Задание

Ниже приведены двадцать пять фрагментов кода. Каждый из них предназначен для вставки в определенное место следующей программы.

```
#include "std_lib_facilities.h"

int main()
try {
     << Здесь будет вставлен ваш код >>
     keep_window_open();
     return 0;
}
catch (exception& e) {
     cerr << "Ошибка: " << e.what() << '\n';
     keep_window_open();
     return 1;
}
catch (...) {
     cerr << "Неизвестное исключение!\n";
     keep_window_open();
     return 2;
}
```

В некоторых из них есть ошибки, а в некоторых — нет. Ваша задача — найти и устранить все ошибки. Если все ошибки будут устранены, программа будет успешно компилироваться и при запуске выводить на экран слово "Success!" Даже если вы считаете, что нашли все ошибки, вставьте в программу исходный (неисправленный) вариант и протестируйте его; может быть, ваша догадка об ошибке была неверной или в фрагменте их несколько. Кроме того, одной из целей этого задания является анализ реакции компилятора на разные виды ошибок. Не вводите всю программу двадцать пять раз — можно просто использовать методику "скопировать и вставить" или иную подобную. Не устраняйте проблемы, удаляя инструкции; исправляйте их путем изменения, добавления или удаления небольшого количества символов.

1. `Cout << "Success!\n";`
2. `cout << "Success!\n;`
3. `cout << "Success" << !\n"`
4. `cout << success << '\n';`
5. `string res = 7; vector<int> v(10); v[5] = res; cout << "Success!\n";`
6. `vector<int> v(10); v(5) = 7; if (v(5)!=7) cout << "Success!\n";`
7. `if (cond) cout << "Success!\n"; else cout << "Fail!\n";`

```
 8. bool c = false; if (c) cout << "Success!\n"; else cout <<
    "Fail!\n";
 9. string s = "ape"; boo c = "fool"<s; if (c) cout << "Success!\n";
10. string s = "ape"; if (s=="fool") cout << "Success!\n";
11. string s = "ape"; if (s=="fool") cout < "Success!\n";
12. string s = "ape"; if (s+"fool") cout < "Success!\n";
13. vector<char> v(5); for(int i=0; 0<v.size(); ++i); cout <<
    "Success!\n";
14. vector<char> v(5); for(int i=0; i<=v.size(); ++i); cout <<
    "Success!\n";
15. string s = "Success!\n"; for (int i=0; i<6; ++i) cout << s[i];
16. if (true) then cout << "Success!\n"; else cout << "Fail!\n";
17. int x = 2000; char c = x; if (c==2000) cout << "Success!\n";
18. string s = "Success!\n"; for (int i=0; i<10; ++i) cout << s[i];
19. vector v(5); for (int i=0; i<=v.size(); ++i); cout <<
    "Success!\n";
20. int i=0; int j = 9; while (i<10) ++j; if (j<i) cout <<
    "Success!\n";
21. int x = 2; double d = 5/(x-2); if (d==2*x+0.5) cout <<
    "Success!\n";
22. string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout
    << s[i];
23. int i=0; while (i<10) ++j; if (j<i) cout << "Success!\n";
24. int x = 4; double d = 5/(x-2); if (d=2*x+0.5) cout <<
    "Success!\n";
25. cin << "Success!\n";
```

Контрольные вопросы

1. Назовите четыре основные разновидности ошибок и кратко опишите их.
2. Какие виды ошибок можно игнорировать в студенческих программах?
3. Что должен гарантировать любой законченный проект?
4. Перечислите три подхода, которые можно предпринять для устранения ошибок в программе и разработки правильного программного обеспечения.
5. Почему мы ненавидим отладку?
6. Что такое синтаксическая ошибка? Приведите пять примеров.

7. Что такое ошибка типа? Приведите пять примеров.
8. Что такое ошибка времени редактирования связей? Приведите три примера.
9. Что такое логическая ошибка? Приведите три примера.
10. Перечислите четыре потенциальных источника ошибок, рассмотренные в тексте.
11. Как распознать правдоподобные результаты? Какие методы используются для ответа на вопросы такого рода?
12. Сравните обработку ошибки времени выполнения в вызывающей функции и в вызываемой функции.
13. Почему использование исключений лучше, чем возврат признака ошибки ("значения ошибки")?
14. Как выполнить тестирование при успешном вводе данных?
15. Опишите процесс генерации и перехвата исключений.
16. Почему выражение `v[v.size()]` для вектора `v` приводит к ошибке диапазона? К чему может привести такой вызов?
17. Дайте определение *предусловия* и *постусловия*; приведите пример (который отличается от функции `area()` из данной главы); предпочтительно с вычислениями, требующими применения цикла.
18. Когда можно *не* проверять предусловие?
19. Когда можно *не* проверять постусловие?
20. Назовите этапы отладки программы.
21. Почему комментарии могут помочь при отладке?
22. Чем тестирование отличается от отладки?

Термины

`assertion`	отладка	ошибка типа
`catch`	ошибка	постусловие
`throw`	ошибка аргумента	предусловие
инвариант	ошибка времени выполнения	синтаксическая ошибка
исключение	ошибка времени компиляции	тестирование
контейнер	ошибка времени компоновки	требование
логическая ошибка	ошибка диапазона	

Упражнения

1. Выполните задание из раздела **Попробуйте**, если вы его еще не выполнили.

2. Следующая программа получает температуру в градусах Цельсия и преобразует ее в температуру в градусах Кельвина. Этот исходный текст содержит много ошибок. Найдите ошибки, перечислите их и исправьте программу.

```
double ctok(double c) // Преобразует температуру
{                     // в градусах Цельсия
                      // в температуру в градусах Кельвина
   int k = c + 273.15;
   return int
}

int main()
{
   double c = 0;           // Объявляем входную переменную
   cin >> d;               // Вводим температуру
   double k = ctok("c");   // Преобразуем температуру
   Cout << k << '/n';      // Выводим температуру
}
```

3. Самой низкой температурой является абсолютный нуль, т.е. –273.15°C, или 0K. Даже после исправления приведенная выше программа выводит неверные результаты для температуры ниже абсолютного нуля. Поместите в функцию `main()` проверку, которая будет выводить сообщение об ошибке для температур ниже –273.15°C.
4. Повторите упр. 3, но в этот раз ошибку обрабатывайте в функции `ctok()`.
5. Измените программу так, чтобы она преобразовывала температуру в градусах Кельвина в температуру в градусах Цельсия.
6. Напишите программу, преобразовывающую температуру в градусах Цельсия в температуру в градусах Фаренгейта и обратно (по формуле из раздела 4.3.3). Воспользуйтесь оценками (раздел 5.8) для того, чтобы убедиться в правдоподобности результатов.
7. Квадратное уравнение имеет вид $a \cdot x^2 + b \cdot x + c = 0$. Для его решения используется формула

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Тем не менее есть одна проблема: если $b^2 - 4ac$ меньше нуля, возникнет ошибка. Напишите программу, вычисляющую решение квадратного уравнения. Создайте функцию, которая выводит на печать все корни квадратного уравнения при заданных коэффициентах a, b и c. Когда программа обнаруживает, что у уравнения нет действительных корней, она должна выводить соответствующее сообщение. Как

распознать правдоподобность результатов? Можете ли вы проверить, что они являются правильными?

8. Напишите программу, считывающую и сохраняющую ряд целочисленных значений, а затем вычисляющую сумму первых *N* чисел. Запросите значение *N*, считайте значения в `vector` и вычислите сумму первых *N* значений. Например:

```
Введите количество суммируемых значений:
3
Введите несколько целых чисел (| для окончания ввода):
12  23  13  24  15  |
Сумма первых 3 чисел (12 23 13) равна 48.
```

Обрабатывайте любые входные данные. Например, если пользователю требуется просуммировать чисел больше, чем сохранено в векторе, выводите соответствующее сообщение.

9. Измените программу из упр. 8 так, чтобы она выводила сообщение об ошибке, если результат не может быть представлен в виде значения типа `int`.

10. Измените программу из упр. 8 так, чтобы она использовала тип `double` вместо `int`. Кроме того, создайте вектор действительных чисел, содержащий *N*–1 разностей между соседними величинами, и выведите этот вектор на печать.

11. Напишите программу, вычисляющую начальный отрезок последовательности Фибоначчи, т.е. последовательности, начинающиеся с чисел 1 1 2 3 5 8 13 21 34. Каждое число в этой последовательности равно сумме двух предыдущих. Найдите наибольшее число Фибоначчи, которое можно записать в переменную типа `int`.

12. Реализуйте простую игру на угадывание "Быки и коровы". Программа должна хранить вектор из четырех различных чисел в диапазоне от 0 до 9 (т.е., например, 2345, но не 4455), а пользователь должен угадать загаданное число. Допустим, программа загадала число 1234, а пользователь назвал число 1359; программа должна ответить "1 бык и 1 корова", поскольку пользователь угадал одну правильную цифру (1) в правильной позиции (бык) и одну правильную цифру (3) в неправильной позиции (корова). Угадывание продолжается, пока пользователь не получит четырех быков, т.е. не угадает четыре правильные цифры в четырех правильных позициях.

13. Следующая программа довольно сложная. Создайте вариант, в котором игрок может играть постоянно (без остановки и повторного запуска) и в каждой новой игре генерируются новые четыре цифры. Четыре случайные цифры можно сгенерировать с помощью четырех вызовов генератора случайных целых чисел `randint(10)` из заголовочного

файла `std_lib_facilities.h`. Обратите внимание на то, что при повторном выполнении программы вы каждый раз будете получать одинаковые последовательности из четырех цифр. Для того чтобы избежать этого, предложите пользователю ввести любое число и до вызова функции `randint(10)` вызовите функцию `srand(n)`, где `n` — число, введенное пользователем. Такое число `n` называется *начальным значением* (seed), причем разные начальные значения приводят к разным последовательностям случайных чисел.

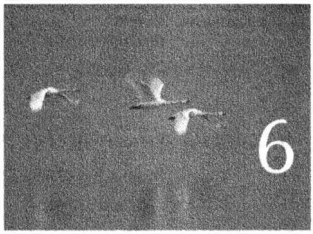
14. Введите пары (день недели, значение) из стандартного потока ввода. Например:

    ```
    Tuesday 23 Friday 56 Tuesday -3 Thursday 99
    ```

    Запишите все значения для каждого дня недели в вектор `vector<int>`. Выведите значения семи векторов для каждого из дней недели. Напечатайте сумму чисел в каждом из векторов. Неправильный день недели, например `Funday`, можно игнорировать, но обычные синонимы допускаются, например `Mon` и `monday`. Выведите на печать количество отвергнутых значений.

Послесловие

Не считаете ли вы, что мы придаем ошибкам слишком большое значение? Новички могут подумать именно так. Очевидная и естественная реакция — "Все просто не может быть настолько плохо!" Именно так, все именно настолько плохо... Лучшие умы планеты пасуют перед сложностью создания правильных программ. По нашему опыту, хорошие математики, как правило, недооценивают проблему ошибок, но всем ясно, что программ, которые с первого раза выполняются правильно, очень немного. Мы вас предупредили! К счастью, за пятьдесят лет программисты научились организовывать код так, чтобы минимизировать количество проблем, и разработали методы поиска ошибок, которые, несмотря на все наши усилия, неизбежны. Методы и примеры, описанные в этой главе, являются хорошей отправной точкой для дальнейшей борьбы с этим неизбежным злом.



Написание программ

Программирование — это понимание.
— Кристен Нюгорд (Kristen Nygaard)

Написание программы предполагает последовательное уточнение того, что вы хотите сделать и как вы желаете это выразить. В этой и следующей главах мы разработаем одну программу, пройдя весь путь от первой еще неясной идеи через этапы анализа, проектирования, реализации, тестирования, повторного проектирования и повторной реализации. Наша цель — дать вам представление о способе мышления программиста, создающего свою программу. В ходе изложения мы обсудим структуру программы, пользовательские типы и обработку входной информации.

6.1. Задача

Создание программы начинается с постановки задачи; иначе говоря, должна существовать задача, которую вы хотите решить с помощью своей программы. Понимание этой задачи является ключевым моментом хорошей программы. В конце концов, программа, решающая неправильно поставленную задачу, вряд ли будет часто применяться, хотя и может быть при этом очень элегантной. Существуют счастливые случаи, когда программа оказывается полезной для решения задачи, для которой она не предназначалась изначально, но на такое везение рассчитывать не стоит. Мы хотим, чтобы программа просто и ясно решала поставленную задачу.

Как может выглядеть программа на этом этапе? Перечислим основные характеристики такой программы.

- Иллюстрирует методы проектирования и программирования.
- Дает возможность исследовать разные варианты решения, которые должен найти программист, и учитывает ограничения, которые наложены на это решение.
- Не требует слишком большого количества новых языковых конструкций.
- Достаточно сложная, чтобы потребовать размышлений над ее проектом.
- Допускает много вариантов решения.
- Решает понятную задачу.

- Решает задачу, которая заслуживает решения.
- Имеет решение, которое достаточно невелико, чтобы его можно было просто и ясно описать и понять.

Мы выбрали следующую задачу: "Поручить компьютеру выполнить простые арифметические операции, введенные пользователем". Иначе говоря, мы хотим создать простой калькулятор. Совершенно очевидно, что такие программы полезны; каждый настольный компьютер поставляется с такой программой, и вы можете даже купить карманный калькулятор, который выполняет только такие программы.

Например, если вы введете строку

```
2+3.1*4
```

то программа должна ответить

```
14.4
```

К сожалению, такая программа не способна на большее, чем уже имеющийся на вашем компьютере готовый калькулятор, но от первой программы не следует требовать слишком многого.

6.2. Размышления над задачей

С чего начать? Просто немного подумайте о задаче и о том, как ее можно решить. Сначала поразмышляйте о том, что должна делать программа и как бы вы хотели с ней взаимодействовать. Затем подумайте о том, как написать такую программу. Попытайтесь написать краткое изложение идеи, лежащей в основе решения, и найдите изъяны в своей первой идее. Возможно, стоит обсудить задачу и способы ее решения со своими друзьями. Объяснение идеи другим удивительным образом позволяет понять ее недостатки и способы ее создания; бумага (или компьютер) не разговаривает с вами и не критикует ваши предложения. В идеале проектирование — коллективная деятельность.

К сожалению, не существует универсальной стратегии решения задач, которая работала бы для всех задач и всех программистов. Есть множество книг, авторы которых обещают научить вас лучше решать задачи, и еще больше книг, посвященных проектированию программ. Наша книга не относится к такой литературе. Мы изложим не более чем предложения по общей стратегии для решения небольших задач, с которыми вы можете столкнуться. После этого мы тут же опробуем эту стратегию на разработке крошечного калькулятора.

Рекомендуем при чтении обсуждения программы калькулятора отнестись к нему достаточно скептично. Для большего реализма мы выполним несколько итераций разработки программы, создав несколько версий и

продемонстрировав идеи, лежащие в основе каждой из них. Очевидно, что большинство из этих идей являются незавершенными и даже ошибочными, иначе нам пришлось бы слишком рано закончить эту главу. По мере продвижения вперед мы приведем примеры разных принципов и рассуждений, которых постоянно придерживаются проектировщики и программисты. В этой главе мы еще не создадим окончательно удовлетворяющий нас вариант программы, отложив эту задачу до следующей главы.

Пожалуйста, в этой и следующей главах помните, что путь, ведущий к окончательному варианту программы и проходящий через промежуточные решения, идеи и ошибки, не менее важен, чем окончательная программа, и более важен, чем языково-технические детали, с которыми мы столкнемся на этом пути (мы вернемся к ним позже).

6.2.1. Стадии разработки программы

Рассмотрим некоторые термины, связанные с разработкой программ. Работая над решением задачи, вы неоднократно проходите следующие этапы.

- *Анализ.* Осознание того, что следует сделать, и описание своего (текущего) понимания задачи. Такое описание называют *набором требований* или *спецификацией*. Мы не будем углубляться в детали разработки и записи этих требований. Этот вопрос выходит за рамки рассмотрения нашей книги, но он становится все более важным по мере увеличения масштаба задачи.
- *Проектирование.* Создание общей структуры системы, принятие решения, из каких частей она должна состоять и как эти части должны взаимодействовать между собой. В качестве составной части проектирования следует решить, какие инструменты (такие, как библиотеки) могут пригодиться при разработке программы.
- *Реализация.* Написание кода, его отладка и проверка, что он в действительности делает то, для чего предназначен.

6.2.2. Стратегия

Приведем некоторые советы, которые при вдумчивом и творческом подходе помогут при создании многих проектов.

- Какая задача должна быть решена? Первое, о чем надо задуматься, — как сформулировать, что именно вы пытаетесь сделать. Обычно это включает формулировку описания задачи, или — если формулировка получена, скажем, от заказчика — попытку разобраться, что эта формулировка означает на самом деле. На этом этапе вы должны поставить себя на место пользователя (а не программиста); иначе

говоря, вы должны задавать вопросы о том, что программа должна делать, а не как она будет это делать. Спросите "Что эта программа может сделать для меня?" и "Как бы я хотел взаимодействовать с этой программой?" Не забывайте, что большинство из нас являются весьма опытными компьютерными пользователями.

- Ясна ли постановка задачи? Для реальных задач на этот вопрос никогда нельзя ответить положительно. Даже студенческое упражнение иногда трудно сформулировать достаточно точно и конкретно. Поэтому попытайтесь уточнить постановку задачи. Было бы обидно решить неправильно поставленную задачу. Другая ловушка состоит в том, что вы можете поставить слишком много вопросов. Пытаясь понять, что вы хотите, легко увлечься и стать слишком претенциозным. Почти всегда лучше задавать поменьше вопросов, чтобы программу было легче описать, понять, использовать и (надеюсь) реализовать. Если ваш подход работает, всегда можно создать более изощренную "версию 2.0", основанную на вашем опыте.
- Выглядит ли задача решаемой при имеющихся времени, умении и инструментах? Мало смысла начинать проект, который вы не сможете завершить. Если у вас мало времени на реализацию (включая тестирование) программы, которая делает все, что от нее требуется, то обычно разумнее и не начинать ее писать. Потребуйте больше ресурсов (особенно времени) или (лучше всего) измените требования так, чтобы упростить задачу.

- Постарайтесь разбить программу на небольшие части. Даже самая маленькая программа, решающая реальную задачу, достаточно велика, чтобы ее можно было разбить на части.
  - Знаете ли вы, какие инструменты, библиотеки и другие подобные ресурсы могут вам помочь? Ответ почти всегда положительный. Даже на самых ранних этапах изучения языка программирования в вашем распоряжении есть части стандартной библиотеки C++. Позднее вы познакомитесь с этой библиотекой поближе и узнаете больше о способах ее эффективного использования. Вам понадобятся графика и библиотеки графического интерфейса пользователя, библиотеки для работы с матрицами и т.п. Получив небольшой опыт, вы сможете найти тысячи таких библиотек в вебе. Помните: не стоит изобретать колесо, разрабатывая программное обеспечение для решения реальных задач. Однако при обучении программированию все обстоит в точности наоборот: это хорошая мысль — заставить ученика заново изобрести колесо, чтобы он понял, как оно действует. Время, которое вы сэкономите, используя хорошую библиотеку, можно посвятить разработке других частей

программы или отдыху. Как понять, что та или иная библиотека подходит для решения вашей задачи и имеет достаточно высокое качество? Это трудная проблема. Можно поспрашивать у коллег, в дискуссионных группах по интересам или попытаться поэкспериментировать с библиотекой на небольших примерах, прежде чем подключать ее к своему проекту.

- Проанализируйте части решения, которые можно описать отдельно (и потенциально использовать в разных местах программы или даже в других программах). Для поиска таких частей требуется опыт, поэтому в данной книге мы приводим много примеров. Мы уже использовали векторы (класс `vector`), строки (класс `string`), а также потоки ввода и вывода (`cin` и `cout`). Эта глава содержит первые завершенные примеры проектирования, реализации и использования программы, содержащей типы, определенные пользователем (`Token` и `Token_stream`). В главах 8 и 13–15 представлено много других примеров вместе с принципами их проектирования. Приведем такую аналогию. Если бы мы конструировали автомобиль, то начали бы с идентификации его составных частей, например колес, двигателя, сидений, дверных ручек и т.д. Современный автомобиль состоит из десятков тысяч таких компонентов. Реальная программа в этом отношении не отличается от автомобиля — за исключением того, что состоит из фрагментов кода. Мы же не пытаемся создавать автомобили непосредственно из исходного сырья, т.е. из стали, пластика и дерева. Поэтому и программы не следует конструировать непосредственно из выражений, инструкций и типов, предоставляемых языком. Проектирование и реализация составных компонентов является основной темой нашей книги и проектирования программного обеспечения вообще (см. описание пользовательских типов в главе 9, иерархии классов — в главе 14 и обобщенных типов — в главе 20).

- Создавайте небольшие и ограниченные версии программы, решающие ключевые части вашей задачи. Начиная работу, мы редко хорошо понимаем задачу. Мы часто думаем, что все прекрасно понимаем (разве мы не знаем, что такое калькулятор?!), но на самом деле это не так. Только сочетание размышлений над задачей (анализ) и экспериментирования (проектирование и реализация) дает глубокое понимание того, что требуется для написания хорошей программы. Итак, пишите небольшие и ограниченные версии, чтобы достичь следующих целей:
  - выявить проблемы в понимании, идеях и требуемом инструментарии;

  - выявить необходимые изменения в формулировке задачи, чтобы упростить работу с ней. Анализируя задачу и создавая первоначальные варианты программы, не стремитесь решить все задачи сразу. Используйте возможности обратной связи, которую дает тестирование.

- Иногда такая ограниченная первоначальная версия называется *прототипом* (prototype). Если первая версия не работает или работает очень плохо (что очень вероятно), отбросьте ее и создайте другую. Повторяйте этот процесс до тех пор, пока не достигнете желаемого. Не пытайтесь навести порядок в имеющейся путанице; со временем она будет лишь возрастать.
- Создавайте полномасштабное решение, в идеале используя части первоначальной версии. Недостижимый идеал заключается в том, что программа должна вырастать из отдельных компонентов, а не быть написана вся и сразу в один присест. В противном случае придется рассчитывать на чудо и надеяться, что непроверенная идея окажется работоспособной и позволит достичь желаемого.

6.3. Вернемся к калькулятору

Как мы хотим взаимодействовать с калькулятором? Это просто: мы знаем, как использовать потоки `cin` и `cout`, зато графические пользовательские интерфейсы (GUI) будут рассмотрены лишь в главе 16, поэтому остановимся на клавиатуре и консольном окне. Получая выражение с клавиатуры, мы вычисляем его и выводим результат на экран, например так:

```
Выражение: 2+2
Результат: 4
Выражение: 2+2*3
Результат: 8
Выражение: 2+3-25/5
Результат: 0
```

Эти выражения, такие как `2+2` и `2+2*3`, должны быть введены пользователем; все остальное делает программа. Для приглашения к вводу мы используем слово "`Выражение:`". Мы могли бы использовать фразу наподобие "`Пожалуйста, введите выражение и символ перехода на новую строку`", но этот вариант выглядит слишком многословным и бессмысленным. С другой стороны, такие короткие приглашения, как `>`, выглядят чересчур загадочно. Анализировать такие варианты использования на ранней стадии проектирования программы весьма важно. Это позволяет сформулировать очень практичное определение минимальных действий программы. При обсуждении проектирования и анализа мы будем называть такие примеры *прецедентами использования* (use cases). Впервые сталкиваясь с

разработкой калькулятора, большинство людей сразу приходят к следующей логике программы:

```
read_a_line
calculate      // Выполнение вычислений
write_result
```

Этот набросок, конечно, — не программа; он называется *псевдокодом* (pseudo code). Псевдокоды обычно используются на ранних этапах проектирования, когда еще не совсем ясно, какой смысл мы вкладываем в обозначения. Например, является ли слово "calculate" вызовом функции? Если да, то с какими аргументами? Для ответа на эти вопросы просто еще не настало время.

6.3.1. Первая попытка

На этом этапе мы действительно еще не готовы написать программу, имитирующую функции калькулятора. Мы просто не думали об этом всерьез, но размышления — трудная работа, а мы, как и большинство программистов, стремимся сразу написать какой-то код. Что ж, попробуем написать простую программу-калькулятор и посмотрим, к чему это приведет. Первое приближение может выглядеть примерно так:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Введите выражение (разрешены + и -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin>>lval>>op>>rval;   // Чтение чего-то наподобие 1 + 3
    if (op=='+')
        res = lval + rval; // Сложение
    else if (op=='-')
        res = lval - rval; // Вычитание

    cout << "Результат: " << res << '\n';
    keep_window_open();
    return 0;
}
```

То есть программа считывает пару значений, разделенных оператором, например `2+2`, вычисляет результат (в данном случае `4`) и выводит его на экран. Здесь переменная, стоящая слева от оператора, обозначена как `lval`, а переменная, стоящая справа от оператора, — как `rval`.

Это (худо-бедно) работает! Ну и что, что программа слишком простая? Очень хорошо получить хоть что-то работающее! Возможно, программирование и компьютерные науки проще, чем о них говорят. Но, может

быть, но не стоит слишком увлекаться ранним успехом. Давайте сделаем кое-что.

1. Сделаем код яснее.
2. Добавим операции умножения и деления (например, `2*3`).
3. Добавим возможность обработки нескольких операторов (например, `1+2+3`).

В частности, мы знаем, что следует проверять корректность входной информации (в нашем варианте мы "забыли" это сделать) и что сравнивать значения с несколькими константами лучше всего с помощью инструкции `switch`, а не `if`.

Цепочку операций, например `1+2+3+4`, мы будем выполнять по мере считывания значений; иначе говоря, начнем с `1`, потом увидим `+2` и добавим `2` к `1` (получим промежуточный результат, равный `3`), увидим `+3` и добавим `3` к промежуточному результату, равному `3`, и т.д. После нескольких неудачных попыток и исправления синтаксических и логических ошибок мы получаем следующий код:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Введите выражение (разрешены +, -, * и /)\n";
    cout << "Добавьте x к концу выражения (1+2*3x): ";
    int lval = 0;
    int rval;
    cin >> lval;    // Чтение крайнего слева операнда

    if (!cin)
    {
        error("нет первого операнда");
    }

    for (char op; cin >> op; ) // Чтение оператора и
    {                          // правого операнда
        // Многократно...
        if (op != 'x')
        {
            cin >> rval;
        }

        if (!cin)
        {
            error("нет второго операнда");
        }

        switch (op)
        {
        case '+':
```

```
            lval += rval; // Сложение: lval = lval + rval
            break;

        case '-':
            lval -= rval; // Вычитание: lval = lval - rval
            break;

        case '*':
            lval *= rval; // Умножение: lval = lval * rval
            break;

        case '/':
            lval /= rval; // Деление: lval = lval / rval
            break;

        default:  // Операторов больше нет; вывод результата
            cout << "Результат: " << lval << '\n';
            keep_window_open();
            return 0;
        }
    }

    error("неверное выражение");
}
```

Выглядит неплохо, но попытайтесь вычислить выражение `1+2*3`, и вы увидите, что результат равен `9`, а не `7`, как утверждают учителя математики. Аналогично `1-2*3` равно `-3`, а не `-5`, как мы думали. Мы выполняем операции в неправильном порядке: `1+2*3` вычисляется как `(1+2)*3`, а не `1+(2*3)`, как обычно. Аналогично `1-2*3` вычисляется как `(1-2)*3`, а не `1-(2*3)`, как обычно. Увы и ах! Конечно, можно объявить правило, согласно которому умножение выполняется раньше, чем сложение, устаревшим, но все же не стоит отменять многовековые правила просто для того, чтобы упростить себе программирование.

6.3.2. Лексемы

Значит, (каким-то образом) мы должны заранее узнать, содержит ли обрабатываемая строка символ `*` (или `/`). Если да, то мы должны (каким-то образом) скорректировать порядок выполнения вычислений. К сожалению, пытаясь заглянуть вперед, мы сразу же наталкиваемся на многочисленные препятствия.

1. Выражение не обязательно занимает ровно одну строку. Рассмотрим пример.

```
1
+
2
```

Такое выражение до сих пор вычислялось без проблем.

2. Как выполнить поиск символа * (или /) среди цифр и символов +, –, (и) в нескольких строках ввода?
3. Как запомнить, в каком месте стоит символ *?
4. Как вычислить выражение, которое не выполняется слева направо (как 1+2*3)?

Решив быть безоглядными оптимистами, мы сначала решим задачи 1–3, отложив задачу 4 на более позднее время.

Кроме того, нам понадобится помощь. Кто-то ведь должен знать, как считывать, например, числа и операторы, из входного потока и сохранять их так, чтобы с ними было удобно работать? Общепринятый и самый полезный ответ на эти вопросы таков: сначала считайте символы, а затем объедините их в *лексемы* (tokens). В этом случае после ввода символов

```
45+11.5/7
```

программа должна создать список лексем

```
45
+
11.5
/
7
```

Лексема (token) — это последовательность символов, выражающих нечто, что мы считаем отдельной единицей, например число или оператор. Именно так компилятор языка C++ работает с исходным кодом программы. Фактически в большинстве случаев анализ исходных текстов начинается с разложения на лексемы в том или ином виде. Анализируя примеры выражений на языке C++, можно выделить три вида необходимых нам лексем.

- Литералы с плавающей точкой, определенные в языке C++, например `3.14`, `0.274e2` и `42`.
- Операторы, такие как `+`, `-`, `*`, `/`, `%`.
- Скобки `(`, `)`.

Внешний вид литералов с плавающей точкой выглядит как могущий стать проблемным: прочесть число `12` намного легче, чем `12.3e-3`, но ведь калькуляторы обычно выполняют вычисления над числами с плавающей точкой. Аналогично следует ожидать, что весьма полезными в программе, имитирующей вычисления калькулятора, окажутся и скобки.

Как представить такие лексемы в нашей программе? Можно попытаться отслеживать начало и конец лексем, но это может привести к путанице (особенно если позволить выражениям занимать несколько строк). Кроме

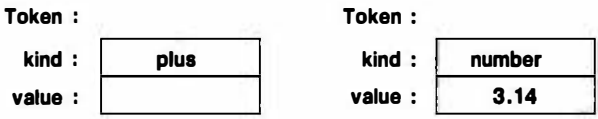
того, если хранить числа в виде строки символов, то позднее придется получать значение числа по его цифрам; например, если мы видим строку `42` и где-то храним символы `4` и `2`, то позднее должны выяснить, что эта строка представляет число `42` (т.е. `4*10+2`). Общепринятое решение этой задачи — хранить каждую лексему в виде пары (*вид*, *значение*). *Вид* идентифицирует лексему — что именно она собой представляет: число, оператор или скобку? Для чисел (в нашем примере — только для чисел) в качестве значения используется само число.

Так как же выразить идею о паре (*вид*, *значение*) в программе? Для этого определим тип `Token`, представляющий лексемы. Почему? Вспомните, почему мы вообще используем типы: они хранят данные, которые нам нужны, и предоставляют возможность выполнять полезные операции над этими данными. Например, тип `int` позволяет хранить целые числа и выполнять операции сложения, вычитания, умножения и вычисления остатка, в то время как тип `string` позволяет хранить последовательности символов и выполнять конкатенацию и доступ к символу по индексу.

В языке C++ и его стандартной библиотеке определено много типов, например `char`, `int`, `double`, `string`, `vector` и `ostream`, но не тип `Token`. Фактически имеется огромное количество типов — тысячи или десятки тысяч, — которые мы хотели бы иметь, но которых нет в языке и в стандартной библиотеке. Среди наших любимых типов, которых нет в библиотеке, — классы `Matrix` (см. главу 24), `Date` (см. главу 9) и целые числа с бесконечной точностью (поищите в вебе "`Bignum`"). Если вы немного подумаете, то поймете, что язык просто не может поддерживать десятки тысяч типов: кто их определит, кто их реализует, как их найти и насколько толстое руководство по использованию языка при этом получится? Как и большинство современных языков программирования, язык C++ решает эту проблему, позволяя программисту при необходимости определять собственные, *пользовательские типы* (т.е. типы, определенные пользователем).

6.3.3. Реализация лексем

Как должна выглядеть лексема в нашей программе? Иначе говоря, как должен выглядеть тип `Token`? Класс `Token` должен быть способен представлять операторы, такие как `+` и `-`, а также числовые значения, такие как `42` и `3.14`. Очевидная реализация должна иметь возможность представлять разновидность лексемы, а также хранить числовое значение лексем, которые его имеют.



Существует много способов реализации этой идеи в исходном тексте на языке C++. Вот простейший вариант, который мы сочли полезным:

```
class Token { // Очень простой пользовательский тип
public:
    char kind;
    double value;
};
```

Класс **Token** — это тип (такой же, как **int** или **char**), поэтому его можно использовать для определения переменных и хранения значений. Он состоит из двух частей (именуемых *членами*): **kind** и **value**. Ключевое слово **class** означает "тип, определенный пользователем"; оно указывает, что тип содержит нуль или более членов. Первый член, **kind**, имеет тип **char**, так что с его помощью удобно хранить символы **'+'** и **'*'** для представления операций * и +. Рассмотрим пример использования этого типа.

```
Token t;           // t - объект класса Token
t.kind = '+';      // t представляет операцию +
Token t2;          // t2 - другой объект класса Token
t2.kind = '8';     // Цифра 8 использована для
t2.value = 3.14;   // обозначения лексем-чисел
```

Для доступа к члену класса используется обозначение *имя_объекта.имя_члена*. Выражение **t.kind** читается как "член **kind** объекта **t**", а выражение **t2.value** — как "член **value** объекта **t2**". Объекты класса **Token** можно копировать так же, как и переменные типа **int**.

```
Token tt = t;      // Инициализация копированием
if (tt.kind != t.kind) error("невозможно!");
t = t2;            // Присваивание
cout << t.value;   // Вывод числа 3.14
```

Имея класс **Token**, можно выразить выражение **(1.5+4)*11** с помощью семи лексем.

'('	'8'	'+'	'8'	')'	'*'	'8'
	1.5		4			11

Обратите внимание на то, что для простых лексем (таких, как +) значение не требуется, поэтому мы не используем член **value**. Нам нужен символ для обозначения чисел. Мы выбрали символ **'8'** просто потому, что он точно не является ни оператором, ни знаком пунктуации. Использование символа **'8'** для обозначения чисел немного загадочно, но это лишь на первых порах.

Класс **Token** представляет пример пользовательского типа C++. Тип, определенный пользователем, может иметь как функции-члены (операции), так и данные-члены. В случае нашего класса **Token** нам не надо

определять какие-либо функции, так как для простых пользовательских типов предоставляются методы чтения и записи членов по умолчанию.

```
class Token {
public:
    char kind;    // Разновидность лексемы
    double value; // Для числел: значение
};
```

Теперь мы можем инициализировать ("конструировать") объекты **Token**, например:

```
Token t1{'+');       // Инициализируем t1 (t1.kind = '+')
Token t2{'8',11.5}; // Инициализируем t2,
                     // (t2.kind = '8' и t2.value = 11.5)
```

Дополнительную информацию об инициализации объектов классов вы найдете в разделах 9.4.2 и 9.7.

6.3.4. Использование лексем

Итак, похоже, что мы сможем завершить наш калькулятор! Однако, пожалуй, стоит уделить немного времени планированию. Как мы будем использовать класс **Token** в калькуляторе? Можно считать входные данные в вектор объектов **Token**.

```
Token get_token(); // Функция чтения лексемы из потока cin

vector<Token> tok; // Сюда помещаем лексемы

int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}
```

Теперь можно сначала считать выражение, а вычислить его позднее. Например, для выражения `11*12` получим следующие лексемы:

'8'	'*'	'8'
11		12

Эти лексемы можно использовать для поиска операции умножения и ее операндов. Это облегчает выполнение умножения, поскольку числа `11` и `12` хранятся как числовые значения, а не как строки.

Рассмотрим теперь более сложные выражения. Выражение `1+2*3` состоит из пяти объектов класса **Token**.

'8'	'+'	'8'	'*'	'8'
1		2		3

Теперь операцию умножения можно выполнить с помощью простого цикла.

```
for (int i = 0; i<tok.size(); ++i) {
    if (tok[i].kind=='*') {    // Найдено умножение!
        double d = tok[i-1].value*tok[i+1].value;
        // И что дальше?
    }
}
```

Да, и что дальше? Что делать с произведением `d`? Как определить порядок вычисления подвыражений? Хорошо, символ `+` предшествует символу `*`, поэтому мы не можем выполнить операции просто слева направо. Можно попытаться выполнить их справа налево! Этот подход сработает для выражения `1+2*3`, но не для выражения `1*2+3`. Или рассмотрим еще худшее выражение `1+2*3+4`. Это пример вычисления "изнутри наружу": `1+(2*3)+4`. А как обработать скобки, раз уж мы решили работать и с ними? Похоже, мы зашли в тупик. Теперь необходимо вернуться назад, прекратить на время программирование и подумать о том, как считывается и интерпретируется входная строка и как вычисляется арифметическое выражение.

Итак, первая попытка решить поставленную задачу (написать калькулятор) оказалась относительно удачной. Это нетипично для первого приближения, которое играет важную роль для понимания задачи. В данном случае это даже позволило нам ввести полезное понятие лексемы, которое является примером понятия пары (*имя, значение*), с которым нам придется встречаться вновь и вновь. Тем не менее всегда следует помнить, что "стихийное" программирование не должно занимать слишком много времени. Необходимо программировать как можно меньше, пока не будут выполнены хотя бы небольшой анализ (понимание задачи) и проектирование (выявление общей структуры решения).

ПОПРОБУЙТЕ

С другой стороны, почему мы не должны иметь возможности найти простое решение этой задачи? Она не выглядит слишком уж сложной. Если даже ничего не получится, такая попытка позволит глубже понять задачу и ее окончательное решение. Рассмотрим, что мы можем сделать прямо сейчас. Взглянем, например, на строку `12.5+2`. Ее можно разбить на лексемы, понять, что выражение простое, и вычислить ответ. Это может оказаться несколько запутанным, но при этом достаточно прямым решением, поэтому, возможно, следовало бы попробовать пойти в этом направлении! Подумайте, что следует делать, если строка содержит операции `+` и `*`, как в выражении `2+3*4`. Его также можно вычислить

с помощью "грубой силы". Но что делать с более сложным выражением, например `1+2*3/4%5+(6-7*(8))`? И как выявлять ошибки, такие как `2+*3` и `2&3`? Подумайте об этом. Будет неплохо, если вы опишете на бумаге возможные решения для интересных или важных арифметических выражений.

6.3.5. Назад к школьной доске!

Теперь настало время снова проанализировать задачу и попытаться не бросаться сломя голову программировать, руководствуясь плохо продуманным планом. Одну важную вещь мы уже выяснили — калькулятор, вычисляющий только одно выражение, никому не интересен. Хотелось бы вычислять несколько выражений одним запуском программы. По этой причине наш псевдокод усложняется.

```
while (not_finished) {
    read_a_line
    calculate          // Выполняем вычисления
    write_result
}
```

Очевидно, что задача усложнилась, но, размышляя о применении калькуляторов, мы понимаем, что они могут вычислять разные арифметические выражения. Следует ли заставлять пользователя вызывать программу несколько раз, чтобы выполнить больше одного вычисления? Это возможно, но такая программа будет неоправданно долго запускаться, так что лучше поискать иное решение.

Проанализировав приведенный псевдокод, наши первые попытки решить задачу, а также примеры использования калькулятора, мы столкнемся с рядом вопросов, на некоторые из которых сможем дать только предварительные ответы.

1. Если мы введем выражение `45+5/7`, то как выделить его отдельные части — `45`, `+`, `5`, `/` и `7`? (Выделением лексем!)
2. Как идентифицировать конец ввода выражения? Разумеется, с помощью символа перехода на новую строку! (Слово "разумеется" всегда подозрительно: "разумеется" — это не причина.)
3. Как представить выражение `45+5/7` в виде данных, чтобы потом вычислить его? Прежде чем выполнить сложение, необходимо из цифр `4` и `5` образовать целое число `45` (т.е. вычислить выражение `4*10+5`). (Таким образом, выделение лексем — всего лишь часть решения.)
4. Как гарантировать, что выражение `45+5/7` вычисляется как `45+(5/7)`, а не как `(45+5)/7`?

5. Чему равно значение `5/7`? Около `.71`, но это число не целое. Используя свой опыт работы с калькуляторами, легко понять, что ответ должен быть числом с плавающей точкой. Следует ли разрешить ввод таких чисел? Конечно!

6. Можно ли использовать переменные? Например, можно ли написать

```
v=7
m=9
v*m
```

Хорошая идея, но пока что это подождет. Сначала следует освоить азы.

Возможно, ответ на шестой вопрос является самым важным. В разделе 7.8 мы увидим, что, ответив "да", мы практически вдвое увеличим размер программы. Это приведет к более чем удвоенным затратам времени на разработку первого приближения. Если вы новичок, то ваши усилия увеличатся даже вчетверо, и проект окончательно выйдет из-под вашего контроля. Очень важно избегать "расползания возможностей" на ранних этапах проекта. Сначала создайте простую версию, реализовав лишь основные функции. Получив работоспособную программу, вы станете более уверенными. Намного проще разрабатывать программу поэтапно, а не сразу всю. Ответив "да" на шестой вопрос, вы столкнетесь с еще одним неприятным эффектом: теперь вам будет сложнее устоять перед соблазном реализовать еще несколько "красивых возможностей". Что вы скажете о вычислении математических функций? А о циклах? Начав накапливать "красивые возможности", очень трудно остановиться.

С точки зрения программиста вопросы 1, 3 и 4 наиболее беспокоящие. Они также связаны друг с другом, поскольку, обнаружив число `45` и оператор `+`, мы должны что-то с ними сделать. Иначе говоря, мы должны решить, как их хранить в программе? Очевидно, что выделение лексем является частью решения, но только частью.

Как поступает опытный программист? Сложные технические вопросы часто имеют стандартные ответы. Известно, что люди пишут программы-калькуляторы столько же времени, сколько существует ввод символов с клавиатуры, т.е. как минимум пятьдесят лет. Должен быть стандартный ответ! В такой ситуации опытный программист консультируется с коллегами или изучает литературу. Глупо надеяться, что за одно утро вы сможете придумать что-то лучшее, чем то, что было сделано за пятьдесят лет.

6.4. Грамматики

Существует стандартный ответ на вопрос о том, как придать выражениям смысл: следует сначала ввести символы, а затем собрать их в лексемы (как мы и сделали). Поэтому, если мы введем выражение

```
45+11.5/7
```

программа должна будет создать список лексем

```
45
+
11.5
/
7
```

Лексема — это последовательность символов, представляющих собой отдельную единицу языка, такую как число или оператор.

После создания лексем программа должна обеспечить корректную интерпретацию завершенных выражений. Например, нам известно, что выражение `45+11.5/7` означает `45+(11.5/7)`, а не `(45+11.5)/7`, но как объяснить программе, что деление имеет более высокий приоритет, чем сложение?

Стандартный ответ — написать *грамматику*, определяющую синтаксис ввода, а затем программу, реализующую правила этой грамматики. Рассмотрим пример.

```
// Пример простой грамматики выражений:
Выражение:
    Терм
    Выражение "+" Терм              // Сложение
    Выражение "–" Терм              // Вычитание
Терм:
    Первичное_выражение
    Терм "*" Первичное_выражение    // Умножение
    Терм "/" Первичное_выражение    // Деление
    Терм "%" Первичное_выражение    // Остаток (деление по модулю)
Первичное_выражение:
    Число
    "(" Выражение ")"               // Группировка
Число:
    Литерал_с_плавающей_точкой
```

Это набор простых правил. Последнее правило читается так: "**Число** — это литерал с плавающей точкой". Предпоследнее гласит: "Нетерминал **Первичное_выражение** представляет собой **Число** или **Выражение** в скобках". Правила для нетерминалов **Выражение** и **Терм** аналогичны; каждый из них определяется в терминах одного из правил, следующих за ним.

Как показано в разделе 6.3.2, наши лексемы, заимствованные из определения языка C++, таковы:

- **Литерал_с_плавающей_точкой** (соответствующий правилам языка C++, например `3.14`, `0.274e2` или `42`);
- `+`, `–`, `*`, `/`, `%` (операторы);
- `(`, `)` (скобки).

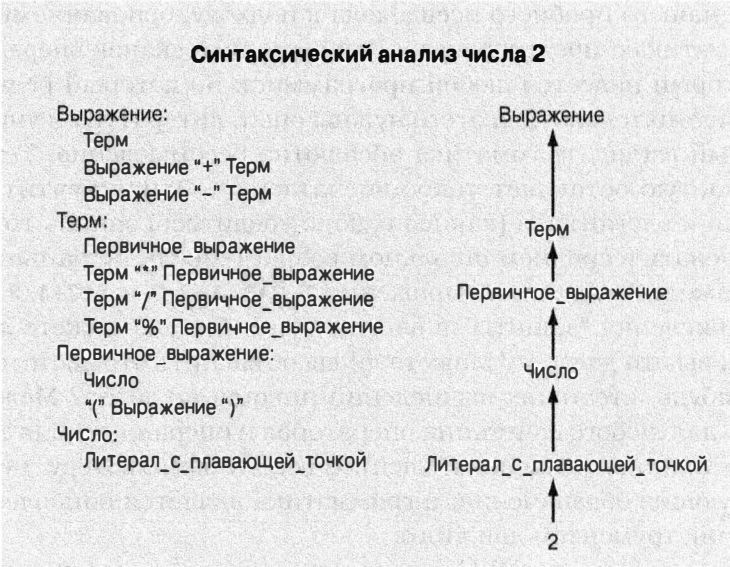
Переход от нашего пробного псевдокода к подходу, основанному на лексемах и грамматиках, представляет собой огромный скачок вперед. Это скачок, на который надеется любой программист, но который редко удается сделать самостоятельно: для этого нужны опыт, литература и учителя.

На первый взгляд, грамматика абсолютна бессмысленна. Техническая запись зачастую оставляет такое впечатление. Однако учтите, что это обобщенная и элегантная (как вы вскоре убедитесь) запись того, что вы способны делать в среднем школьном возрасте (если не раньше). Вы без проблем можете вычислить выражения `1-2*3`, `1+2-3` и `3*2+4/2`. Кажется, что эти вычисления "зашиты" в вашем мозге. Однако можете ли вы объяснить, как вы это делаете? Можете ли вы объяснить это достаточно хорошо кому-нибудь, кто таких вычислений никогда не делал? Можете ли вы сделать это для любого сочетания операторов и операндов? Для того чтобы достаточно точно и подробно объяснить все это компьютеру, необходимы соответствующие обозначения, и грамматика является наиболее мощным и удобным инструментом для этого.

Как читать грамматику? Получив некое входное выражение, мы начинаем с правила "верхнего уровня", **Выражения**, и ищем среди правил совпадения для считанных лексем. Чтение потока лексем в соответствии с грамматикой называется *синтаксическим анализом* (parsing), а программа, выполняющая эту работу, называется *синтаксическим анализатором* (parser или syntax analyzer). Наш синтаксический анализатор считывает лексемы слева направо, в том же порядке, в каком мы их вводим. Рассмотрим простой пример: является ли `2` выражением?

1. **Выражение** должно быть **Термом** или заканчиваться **Термом**. Этот **Терм** должен быть **Первичным_выражением** или завершаться **Первичным_выражением**. В свою очередь, **Первичное_выражение** должно начинаться с открывающей скобки `(` или быть **Числом**. Очевидно, что раз `2` — не открывающая скобка `(`, эта цифра может рассматриваться как **Литерал_с_плавающей_точкой**, т.е. как нетерминал **Число**, который, в свою очередь, является **Первичным_выражением**.
2. Этому **Первичному выражению** (**Число** `2`) не предшествует ни символ `/`, ни `*`, ни `%`, поэтому оно является завершенным **Термом** (а не завершением выражения с символом `/`, `*` или `%`).
3. Этому **Терму** (**Первичное_выражение** `2`) не предшествует ни символ `+`, ни `-`, поэтому он является завершенным **Выражением** (а не завершением выражения с символом `+` или `-`).

Таким образом, в соответствии с нашей грамматикой `2` — это выражение. Данный синтаксический анализ можно описать следующим образом.

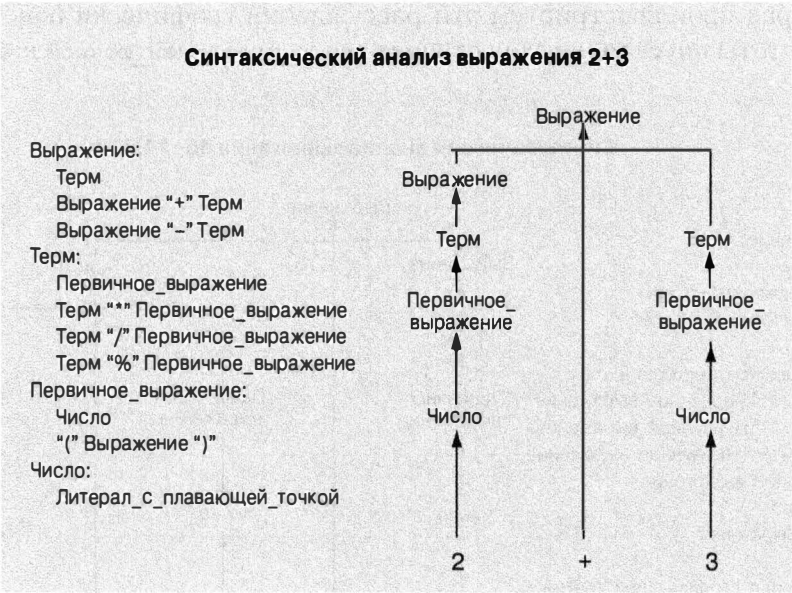


Этот рисунок иллюстрирует путь, который мы прошли, перебирая определения. Повторяя этот путь, мы видим, что **2** — это выражение, поскольку **2** — это **Литерал_с_плавающей_точкой**, который является **Числом**, которое является **Первичным_выражением**, которое является **Термом**, который является **Выражением**.

Попробуем проделать более сложное упражнение: **2+3** — это **Выражение**? Естественно, большинство рассуждений совпадает с рассуждениями для числа **2**.

1. **Выражение** должно быть **Термом** или заканчиваться **Термом**, который должен быть **Первичным_выражением** или заканчиваться **Первичным_выражением**, а **Первичное_выражение** должно начинаться с открывающей скобки **(** или быть **Числом**. Очевидно, что **2** является не открывающей скобкой **(**, а **Литералом_с_плавающей_точкой**, который является **Числом**, которое является **Первичным_выражением**.
2. Этому **Первичному_выражению** (**Число 2**) не предшествует ни символ **/**, ни *****, ни **%**, поэтому оно является завершенным **Термом** (а не завершением выражения с символом **/**, ***** или **%**).
3. За этим **Термом** (**Первичным_выражением 2**) следует символ **+**, поэтому он является окончанием первой части **Выражения**, и мы должны поискать **Терм**, который следует за символом **+**. Точно так же, как мы вывели, что **2** — это **Терм**, мы приходим к выводу, что **3** — это тоже **Терм**. Поскольку за **Термом 3** не следует ни символ **+**, ни **-**, он является завершенным **Термом** (а не первой частью **Выражения**, содержащего символ **+** или **-**). Следовательно, **2+3** соответствует правилу **Выражение + Term** и является **Выражением**.

Снова проиллюстрируем эти рассуждения графически (для простоты опуская переход от **Литерала_с_плавающей_точкой** к **Числу**).



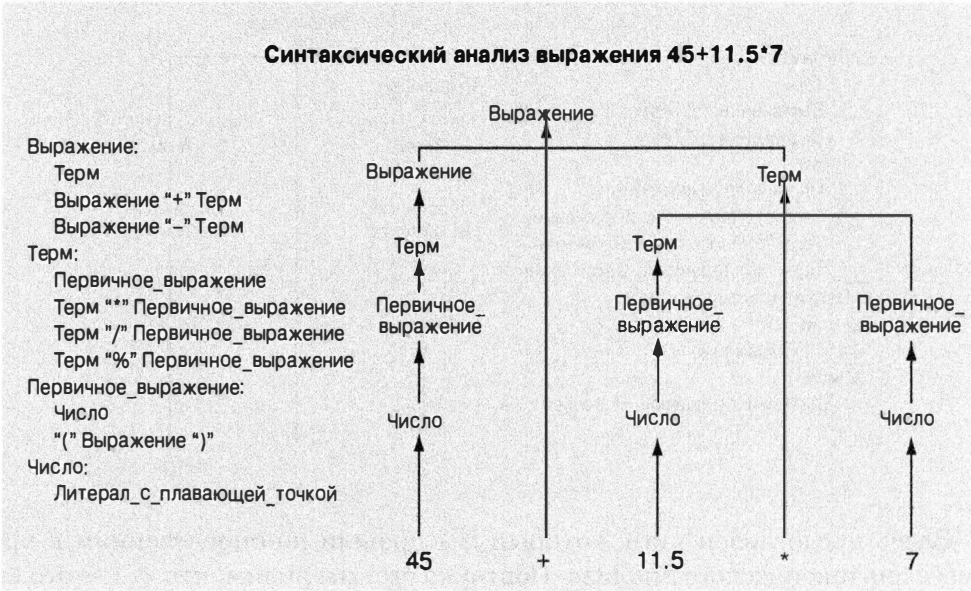
Здесь представлен путь, который мы прошли по определениям в процессе синтаксического анализа. Повторяя его, мы видим, что `2+3` — это **Выражение**, так как `2` — это **Терм**, который является **Выражением**, `3` — это **Терм**, а **Выражение**, за которым следует символ `+` и **Терм**, является **Выражением**.

Фактически причина, по которой мы так интересуемся грамматиками, заключается в том, что с их помощью можно решить проблему корректного синтаксического анализа выражений, содержащих символы `+` и `*`, такие как `45+11.5*7`. Однако играть роль компьютера и анализировать правила так детально весьма утомительно. Поэтому пропустим промежуточные этапы, которые мы проделали для выражений `2` и `2+3`. Очевидно, что `45`, `11.5` и `7` являются **Литералами_с_плавающей_точкой**, которые являются **Числами**, которые являются **Первичными_выражениями**, так что можно игнорировать все правила, располагающиеся ниже **Первичного_выражения**.

1. `45` — это **Выражение**, за которым следует символ `+`, поэтому мы ищем **Терм**, чтобы применить правило **Выражение+Терм**.
2. `11.5` — это **Терм**, за которым следует символ `*`, поэтому следует искать **Первичное_выражение**, чтобы применить правило **Терм*Первичное выражение**.
3. `7` — это **Первичное_выражение**, поэтому `11.5*7` — это **Терм** в соответствии с правилом **Терм*Первичное_выражение**. Теперь можно убедиться, что `45+11.5*7` — это **Выражение** в соответствии с правилом **Выражение + Терм**. В частности, это **Выражение**, которое сначала выполняет

умножение `11.5*7`, а затем сложение `45+11.5*7` так, будто мы написали выражение `45+(11.5*7)`.

Еще раз проиллюстрируем эти рассуждения графически (как и ранее, для простоты опуская переход от **`Литерала_с_плавающей_точкой`** к **`Числу`**).



Как и ранее, этот рисунок иллюстрирует путь, пройденный по определениям. Обратите внимание на то, что правило **`Терм*Первичное_выражение`** обеспечивает умножение `11.5` на `7`, а не сложение с `45`.

Эта логика может показаться запутанной, но многие люди легко читают грамматики, и простые грамматики не так уж трудно понять. Однако мы не собираемся учить *вас* вычислять выражение `2+2` или `45+11.5*7`. Очевидно, вы это и так знаете. Мы лишь пытаемся найти способ заставить компьютер "понимать" выражение `45+11.5*7` и все другие сложные выражения, которые вы можете передать ему для вычисления. Разумеется, сложные грамматики предназначены не для людей, а для компьютеров, которые быстро и правильно следуют их правилам. Следовать точным правилам — это именно то, что компьютеры умеют лучше всего.

6.4.1. Отступление: грамматика английского языка

Если вы еще никогда не работали с грамматиками, то ваша голова может закружиться. Но даже если вы уже сталкивались с грамматиками, ваша голова может закружиться, когда вы увидите следующую грамматику, описывающую очень небольшую часть английского языка.

```
Предложение:
    Имя_существительное Глагол    // Например, C++ rules
    Предложение Союз Предложение // Например, Birds fly but
                                 // fish swim

Союз :
    "and"
    "or"
    "but"

Имя_существительное:
    "birds"
    "fish"
    "C++"

Глагол:
    "rules"
    "fly"
    "swim"
```

Предложение состоит из частей речи (например, имен существительных, глаголов и союзов). В соответствии с этими правилами предложение можно разложить на слова — имена существительные, глаголы и т.д. Эта простая грамматика включает в себя и семантически бессмысленные предложения, такие как "C++ fly and birds rules", но решение этой проблемы выходит далеко за рамки рассмотрения нашей книги.

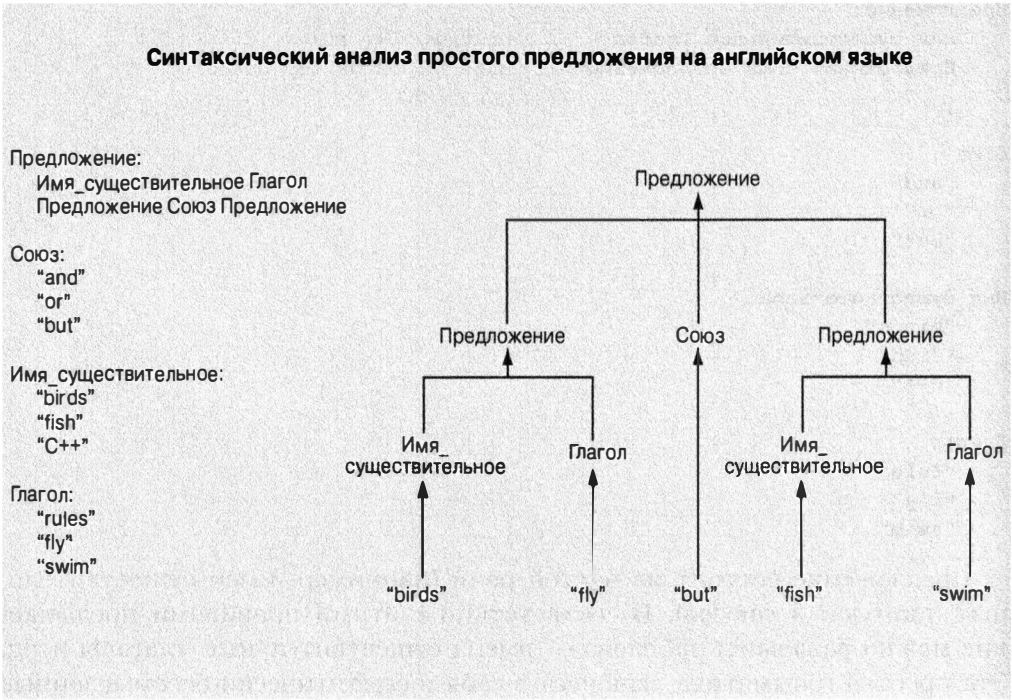
Многие читатели наверняка уже встречались с такими правилами в средней школе при изучении иностранных языков. Эти правила носят фундаментальный характер. В их основе лежат серьезные неврологические аргументы, утверждающие, что эти правила каким-то образом "встроены" в наш мозг!

Рассмотрим дерево синтаксического анализа простого предложения на английском языке.

Сложности еще не закончились. Если вы не уверены, что все правильно поняли, то вернитесь и перечитайте раздел 6.4 с самого начала. Возможно, при втором чтении вы поймете, о чем идет речь!

6.4.2. Написание грамматики

Как выбираются грамматические правила для анализа указанных выше выражений? Самым честным ответом является "опыт". Способ, который мы применили, просто повторяет способ, с помощью которого люди обычно создают грамматики. Однако написание простой грамматики — дело не слишком сложное: нам необходимо лишь знать следующее.



1. Как отличить правило от лексемы.
2. Как размещать правила одно за другим (последовательность).
3. Как выражать альтернативные варианты (ветвление).
4. Как выражать повторяющиеся варианты (повторение).
5. Как распознавать начальное правило грамматики.

В разных учебниках и системах синтаксического анализа используются разные соглашения и терминология. Например, лексемы (которые в других книгах могут называть *токенами*) иногда называют *терминалами* (terminals), а правила — *нетерминалами* (non-terminals) или *продукциями* (productions). Мы просто заключаем лексемы в двойные кавычки и начинаем с первого правила. Альтернативы размещаются по одной на строке. Рассмотрим пример.

```
Список:
    "{" Последовательность "}"
Последовательность:
    Элемент
    Элемент "," Последовательность
Элемент:
    "A"
    "B"
```

Таким образом, **Последовательность** — это или **Элемент**, или **Элемент**, за которым следуют разделяющая запятая и другая **Последовательность**. **Элемент** — это либо буква **A**, либо буква **B**. **Список** — это **Последовательность** в фигурных скобках. Можно сгенерировать следующие **Списки** (как?):

```
{A}
{B}
{A,B}
{A,A,A,A,B}
```

Однако то, что перечислено ниже, списком не является (почему?):

```
{}
A
{A,A,A,A,B
{A,A,C,A,B }
{A B C}
{A,A,A,A,B,}
```

Это не те правила, которым вас учили в детском садике, и в ваш мозг они не "вшиты"; тем не менее понять их несложно. Примеры использования грамматик для выражения синтаксических идей можно найти в разделах 7.4 и 7.8.1.

6.5. Превращение грамматики в программу

Существует много способов заставить компьютер следовать грамматическим правилам. Мы используем простейший из них: просто напишем функцию для каждого грамматического правила, а для представления лексем применим класс **Token**. Программу, реализующую грамматику, часто называют *программой синтаксического анализа* (parser), или просто синтаксическим анализатором.

6.5.1. Реализация грамматических правил

Для реализации калькулятора нам нужны четыре функции: одна — для считывания лексем и по одной для каждого грамматического правила.

```
get_token()   // Считывает символы и составляет лексемы
              // Использует поток cin
expression()  // Работает с операциями + и -
              // Вызывает функции term() и get_token()
term()        // Работает с операциями *, / и %
              // Вызывает функции primary() и get_token()
primary()     // Работает с числами и скобками
              // Вызывает функции expression() и get_token()
```

Примечание: каждая функция работает с определенными частями выражения, оставляя все остальное другим функциям; это позволяет радикально упростить каждую функцию. Такая ситуация напоминает группу людей, пытающихся решить задачу, разделив ее на части и поручив решение отдельных подзадач каждому из членов группы.

Что же эти функции должны делать в действительности? Каждая из них должна вызывать другие грамматические функции в соответствии с грамматическим правилом, которое она реализует, а также функцию **get_token()**, если в правиле упоминается лексема. Например, когда функция **primary()** пытается следовать правилу **(Выражение)**, она должна вызвать следующие функции:

```
get_token()    // Чтобы обработать скобки ( и )
expression()   // Чтобы обработать Выражение
```

Что должен возвращать такой синтаксический анализатор? Как насчет интересующего нас результата вычислений? Например, для выражения **2+3** функция **expression()** должна была бы возвращать **5**, как и подобает калькулятору. Именно этого и следует добиться. Поступая таким образом, мы избегаем ответа на один из труднейших вопросов: "Как представить выражение **45+5/7** в виде данных, чтобы его можно было вычислить?" Вместо того чтобы хранить в памяти представление этого выражения, мы просто вычисляем его по мере считывания входных данных. Эта простая идея коренным образом изменяет ситуацию! Она позволяет в четыре раза уменьшить размер программы по сравнению с вариантом, в котором функция **expression()** возвращает что-то сложное для последующего вычисления. Таким образом, мы сэкономим около 80% объема работы.

Функция **get_token()** стоит особняком: поскольку она обрабатывает лексемы, а не выражения, она не может возвращать значения подвыражений. Например, **+** и **(** — это не выражения. Таким образом, функция **get_token()** должна возвращать объект класса **Token**. Итак, нам нужны следующие функции.

```
// Функции, соответствующие правилам грамматики
Token get_token()    // Считывает символы и составляет лексемы
double expression()  // Работает с операциями + и -
double term()        // Работает с операциями *, / и %
double primary()     // Работает с числами и скобками
```

6.5.2. Выражения

Сначала напишем функцию **expression()**. Грамматическое правило **Выражение** выглядит следующим образом:

```
Выражение:
    Терм
```

```
Выражение '+' Терм
Выражение '-' Терм
```

Поскольку это первая попытка реализовать набор грамматических правил в виде программного кода, начнем с пары неверных попыток. Это путь, по которому обычно идет изучение новых методик, и на этом пути мы научимся некоторым полезным вещам. В частности, новичок может многое понять, обнаружив, что похожие фрагменты кода могут вести себя совершенно по-разному. Чтение программного кода — это полезный навык, который следует культивировать.

6.5.2.1. Выражения: первая попытка

Посмотрев на правило **Выражение '+' Терм**, сначала попытаемся вызвать функцию `expression()`, поищем операцию + (и –), а затем вызовем функцию `term()`.

```
double expression()
{
    double left = expression();// Считываем и вычисляем выражение
    Token t = get_token();     // Получаем следующую лексему
    switch (t.kind) {          // Определяем вид лексемы
    case '+':
        return left + term();  // Считываем и вычисляем Терм,
                               // затем выполняем сложение
    case '-':
        return left - term();  // Считываем и вычисляем Терм,
                               // затем выполняем вычитание
    default:
        return left;           // Возвращаем значение выражения
    }
}
```

Исходный текст выглядит неплохо. Это почти тривиальный перевод грамматики в код. Функция довольно проста: сначала считываем **Выражение**, а затем проверяем, следует ли за ним символ + или –, и в случае положительного ответа считываем **Терм**.

К сожалению, на самом деле в этом программном коде мало смысла. Как узнать, где кончается выражение, чтобы искать символ + или –? Напомним, что наша программа считывает символы слева направо и не может заглянуть вперед, чтобы узнать, нет ли там символа +. В действительности данный вариант функции `expression()` никогда не продвинется дальше своей первой строки: функция `expression()` начинает работу с вызова функции `expression()`, которая, в свою очередь, начинается с вызова функции `expression()`, и так до бесконечности. Этот процесс называется *бесконечной рекурсией*, но на самом деле он довольно быстро заканчивается, исчерпав память компьютера. Термин *рекурсия* используется для описания процесса, который выполняется, когда функция вызывает саму

себя. Не любая рекурсия является бесконечной; рекурсия на самом деле является очень полезным методом программирования (раздел 8.5.8).

6.5.2.2. Выражения: вторая попытка

Итак, что же делать? Каждый **Терм** является **Выражением**, но не каждое **Выражение** является **Термом**; то есть можно начать поиск с **Терма** и переходить к поиску полного **Выражения**, только обнаружив символ + или –. Рассмотрим пример.

```
double expression()
{
    double left = Term();              // Считываем и вычисляем терм
    Token t = get_token();             // Получаем следующую лексему
    switch (t.kind) {                  // Определяем вид лексемы
    case '+':
        return left + expression(); // Считываем и вычисляем
                        // выражение, затем выполняем сложение
    case '-':
        return left - expression(); // Считываем и вычисляем
                        // выражение, затем выполняем вычитание
    default:
        return left;                   // Возвращаем значение терма
    }
}
```

Этот программный код действительно — более или менее — работает. Мы попробуем включить его в окончательный вариант программы синтаксического анализа правильных выражений и отбраковки неправильных. Он даже правильно вычисляет большинство выражений. Например, выражение `1+2` считывается как **Терм** (имеющий значение `1`), за которым следует символ `+`, а за ним — **Выражение** (которое оказывается **Термом**, имеющим значение `2`). В итоге получаем ответ, равный `3`. Аналогично выражение `1+2+3` дает ответ `6`. Можно было бы долго говорить о том, что эта функция делает хорошо, но у нас нет столько времени, так что сразу перейдем к вопросу, чему равно выражение `1-2-3`. Функция `expression()` считывает число `1` как **Терм**, затем переходит к считыванию `2-3` как **Выражения** (состоящего из **Терма** `2`, за которым следует **Выражение** `3`). Таким образом, из `1` будет вычтено значение выражения `2-3`. Иначе говоря, программа вычисляет выражение `1-(2-3)`, которое равно `2`. Однако мы еще со школьной скамьи знаем, что выражение `1-2-3` означает `(1-2)-3` и, следовательно, равно `-4`.

Итак, мы написали красивую программу, которая выполняет вычисления неправильно. Это опасно. Это особенно опасно, поскольку во многих случаях программа дает правильный ответ. Например, выражение `1+2+3` будет вычислено правильно (`6`), так как `1+(2+3)` эквивалентно `(1+2)+3`. Что же такое фундаментальное с точки зрения программирования мы сделали неправильно? Этот вопрос следует задавать себе каждый раз, когда

обнаружите ошибку. Только так и можно избежать повторения одних и тех же ошибок снова и снова.

По существу, мы просто просмотрели программный код и угадали правильное решение. Это крайне редко срабатывает! Мы должны понимать, как работает программа, и уметь объяснить, почему она работает правильно.

Анализ ошибок — часто лучший способ найти правильное решение. В данном случае функция **expression()** сначала искала **Терм**, а затем, если за **Термом** следовал символ + или –, искала **Выражение**. На самом деле функция реализует немного отличающуюся грамматику.

```
Выражение:
    Терм
    Терм '+' Выражение // Сложение
    Терм '-' Выражение // Вычитание
```

Отличие от нашей грамматики заключается именно в том, что выражение `1-2-3` должно трактоваться как **Выражение** `1-2`, за которым следуют символ – и **Терм** `3`, а на самом деле функция интерпретирует выражение `1-2-3` как **Терм** `1`, за которым следует символ – и **Выражение** `2-3`. Иначе говоря, мы хотели, чтобы выражение `1-2-3` было эквивалентно `(1-2)-3`, а получили `1-(2-3)`.

Да, отладка утомительна, скучна и требует много времени, но в данном случае мы действительно работаем с правилами, известными со школьной скамьи, и не должны испытывать больших затруднений. Проблема заключается лишь в том, чтобы научить этим правилам компьютер, а он учится намного медленнее вас.

Заметим, что мы могли бы определить выражение `1-2-3` как `1-(2-3)`, а не `(1-2)-3` и вообще избежать этой дискуссии. Довольно часто самые трудные программистские проблемы возникают тогда, когда мы работаем с привычными для людей правилами, которые изобрели задолго до компьютеров.

6.5.2.3. Выражения: третья (удачная) попытка

Что дальше? Еще раз взгляните на грамматику (правильная грамматика приведена в разделе 6.5.2): любое **Выражение** начинается с **Терма**, за которым может следовать символ + или –. Следовательно, мы должны найти **Терм**, проверить, следует ли за ним символ + или –, и делать это, пока символы "плюс" и "минус" не закончатся. Рассмотрим пример.

```
double expression()
{
    double left = term();       // Считываем и вычисляем Терм
    Token t = get_token();      // Получаем следующую лексему
    while ( t.kind=='+' ||
            t.kind=='-') {      // Ищем + или -
        if (t.kind == '+')
            left += term();     // Вычисляем Терм и добавляем его
```

```
        else
            left -= term();     // Вычисляем Терм и вычитаем его
        t = get_token();
    }
    return left;    // Конец: символов + и - нет; возвращаем ответ
}
```

Этот вариант немного сложнее: мы должны были ввести цикл для поиска символов + и -. Кроме того, проверка символов + и - повторяется дважды, и дважды же вызывается функция **get_token()**. Чтобы не запутывать логику программы, давайте устраним повторную проверку символов + и -.

```
double expression()
{
    double left = term();       // Считываем и вычисляем Терм
    Token t = get_token();      // Получаем следующую лексему
     while(true) {
        switch(t.kind) {
        case '+':
            left += term();     // Вычисляем Терм и суммируем
            t = get_token();
            break;
        case '-':
            left -= term();     // Вычисляем Терм и вычитаем
            t = get_token();
            break;
        default:
            return left;        // Конец: символов + и - нет;
                                // возвращаем ответ
        }
    }
}
```

Обратите внимание на то, что — за исключением цикла — этот вариант очень похож на первую попытку (раздел 6.5.2.1). Мы просто удалили вызов функции **expression()** в функции **expression()** и заменили его циклом. Другими словами, мы транслировали **Выражение** из правила грамматики в цикл поиска **Терма**, за которым следует символ + или -.

6.5.3. Термы

Правило грамматики для **Терма** очень похоже на правило для **Выражения**.

```
Терм:
    Первичное_выражение
    Терм '*' Первичное_выражение
    Терм '/' Первичное_выражение
    Терм '%' Первичное_выражение
```

Следовательно, программный код также должен быть очень похож на код для **Выражения**. Вот как выглядит первая попытка:

```
double term()
{
    double left = primary();
    Token t = get_token();
    while (true)
    {
        switch (t.kind)
        {
        case '*':
            left *= primary();
            t = get_token();
            break;
        case '/':
            left /= primary();
            t = get_token();
            break;
        case '%':
            left %= primary();
            t = get_token();
            break;
        default:
            return left;
        }
    }
}
```

К сожалению, этот код не компилируется: операция вычисления остатка (`%`) для чисел с плавающей точкой не определена. Компилятор вежливо предупредит нас об этом. Когда мы утвердительно ответили на вопрос 5 из раздела 6.3.5 — "Следует ли позволить ввод чисел с плавающей точкой?", — мы не думали о таких последствиях и просто поддались искушению добавить в программу дополнительные возможности. Вот так всегда! Что же делать? Можно во время выполнения программы проверить, являются ли оба операнда операции `%` целыми числами, и сообщить об ошибке, если это не так. А можно просто исключить операцию `%` из возможностей нашего калькулятора. Эту функцию всегда можно будет добавить позже (см. раздел 7.5). Исключив операцию `%`, мы получим вполне работоспособную функцию: термы правильно распознаются и вычисляются. Однако опытный программист заметит нежелательную деталь, которая делает функцию `term()` неприемлемой. Что произойдет, если ввести выражение `2/0`? На нуль делить нельзя. Если попытаться это сделать, аппаратное обеспечение компьютера обнаружит такую попытку и пресечет ее, прекратив выполнение программы и выдав сообщение об ошибке. Неопытный программист обязательно столкнется с этой проблемой. По этой причине лучше провести проверку самостоятельно и выдать соответствующее сообщение.

```
double term()
{
    double left = primary();
    Token t = get_token();
    while (true)
    {
        switch (t.kind)
        {
        case '*':
            left *= primary();
            t = get_token();
            break;
        case '/':
        {
            double d = primary();
            if (d == 0)
            {
                error("деление на нуль");
            }
            left /= d;
            t = get_token();
            break;
        }
        default:
            return left;
        }
    }
}
```

Почему мы поместили обработку операции / внутри блока? На этом настоял компилятор. Если мы хотим определять и инициализировать переменные в операторе `switch`, их следует разместить в блоке.

6.5.4. Первичные выражения

Грамматическое правило для первичных выражений также простое.

```
Первичное_выражение:
    Число
    '(' Выражение ')'
```

Программный код, реализующий это правило, немного сложнее, поэтому он открывает больше возможностей для синтаксических ошибок.

```
double primary()
{
    Token t = get_token();
    switch (t.kind)
    {
    case '(':           // Обработка '(' expression ')'
    {
        double d = expression();
```

```
        t = get_token();
        if (t.kind != ')')
        {
            error("требуется ')'");
        }
        return d;
    }
    case '8':            // Используем '8' для представления числа
        return t.value; // Возвращаем значение числа

    default:
        error("требуется первичное выражение");
    }
}
```

В основном здесь нет ничего нового по сравнению с функциями **expression()** и **term()**: используются те же языковые примитивы, те же способы работы с **Token** и те же приемы программирования.

6.6. Испытание первой версии

Чтобы выполнить эти функции калькулятора, необходимо реализовать функции **get_token()** и **main()**. Функция **main()** тривиальна: мы просто вызываем функцию **expression()** и выводим результат на печать.

```
int main()
try
{
    while (cin)
    {
        cout << expression() << '\n';
    }

    keep_window_open();
}
catch (exception& e)
{
    cerr << e.what() << '\n';
    keep_window_open ();
    return 1;
}
catch (...)
{
    cerr << "исключение\n";
    keep_window_open ();
    return 2;
}
```

Обработка ошибок выполняется по обычному шаблону из раздела 5.6.3. Отложим описание реализации функции `get_token()` до раздела 6.8 и протестируем эту первую версию калькулятора.

ПОПРОБУЙТЕ

Первая версия программы калькулятора (включающая функцию `get_token()`) содержится в файле `calculator00.cpp`. Попробуйте ее запустить и поработать с ней.

Нет ничего удивительного в том, что эта первая версия калькулятора работает не совсем так, как мы ожидали. Мы пожимаем плечами и спрашиваем себя "Почему?" или "Почему программа делает то, что она делает?", или "Что же она делает?" Введите число `2` и символ перехода на новую строку. Ответа вы не получите! Введите символ перехода на новую строку еще раз, чтобы убедиться, что компьютер не завис. Ответа по-прежнему нет. Введите число `3` и символ перехода на новую строку. Ответа нет! Введите число `4` и символ перехода на новую строку. Ответ равен `2`! Теперь экран выглядит так:

```
2

3
4
2
```

Введем выражение `5+6`. Ответ равен 5, а экран выглядит так:

```
2

3
4
2
5+6
5
```

Если только вы не программировали раньше, скорее всего, вы будете сильно озадачены. Такое поведение может озадачить даже программиста с определенным опытом. Что же происходит? Попробуйте теперь выйти из программы. Как это сделать? Да, мы "забыли" указать в программе команду выхода, но прекращение работы может спровоцировать ошибка, поэтому введите символ `x`. В ответ программа выведет на экран фразу `Неправильная лексема` и закончит работу. Наконец-то хоть что-то работает, как запланировано!

Однако мы забыли обеспечить различие между вводом и выводом на экран. Прежде чем перейти к решению основной задачи, давайте исправим ситуацию, чтобы экран лучше отражал то, что мы делаем. Добавим для указания выводимого результата символ `=`:

```
while (cin) cout << "= " << expression() << '\n'; // Версия 1
```

Теперь введем ту же самую последовательность символов, что и раньше. На экране появится следующее:

```
2

3
4
= 2
5+6
= 5
x
Неправильная лексема
```

Странно! Попробуйте разобраться, что и почему делает программа. Можно попробовать еще несколько примеров подобного ввода, но давайте рассмотрим уже имеющийся пример.

- Почему программа не реагирует на ввод символов 2, 3 и символов новой строки?
- Почему после ввода числа 4 программа выводит на экран число 2, а не 4?
- Почему при вычислении выражения 5+6 программа выводит число 5, а не 11?

Существует множество способов получить такие загадочные результаты. Некоторые из них мы рассмотрим в следующей главе, а пока просто подумаем. Может быть, программа неверно выполняет арифметические действия? Непохоже: значение 4 не может быть равным 2, а 5+6 равно 11, а не 5. Давайте посмотрим, что будет, если мы введем символы 1 2 3 4+5 6+7 8+9 10 11 12 и символ перехода на новую строку.

```
1 2 3 4+5 6+7 8+9 10 11 12
= 1
= 4
= 6
= 8
= 10
```

Ну и как? Нет ни 2, ни 3. Почему число 4 в выводе есть, а числа 9 нет (т.е. нет 4+5)? Почему среди результатов есть число 6 и нет числа 13 (т.е. нет 6+7)? Хорошенько присмотритесь: программа выводит каждую третью лексему! Может быть, программа "съедает" часть входной информации без вычислений? Похоже на то. Проанализируем функцию `expression()`.

```
double expression()
{
    double left = term();       // Считываем и вычисляем Терм
    Token t = get_token();      // Получаем следующую лексему
```

```
    while(true) {
        switch(t.kind) {
        case '+':
            left += term();     // Вычисляем Терм и суммируем
            t = get_token();
            break;
        case '-':
            left -= term();     // Вычисляем Терм и вычитаем
            t = get_token();
            break;
        default:
            return left;        // Конец: символов + и - нет;
                                // возвращаем ответ
        }
    }
}
```

Если объект класса **Token**, возвращаемый функцией **get_token()**, не является ни '+', ни '-', выполняется выход из функции. Мы никак не используем этот объект и не храним его в памяти для использования в других функциях. Это неразумно. Отбросить входную информацию, даже не попытавшись выяснить, что она собой представляет, — плохая идея. Беглый анализ показывает, что функции **term()** присущ такой же недостаток. Это объясняет, почему наш калькулятор "съедает" по две лексемы после одной использованной.

Модифицируем функцию **expression()** так, чтобы она не "съедала" лексемы. Куда поместить следующую лексему (**t**), если программа никак не использует ее? Можно рассмотреть много сложных схем, но давайте просто перейдем к очевидному ответу (очевидному, если вы видите эту очевидность): поскольку лексема будет использоваться другой функцией, читающей лексемы из потока ввода, давайте просто вернем лексему обратно в поток ввода! В действительности вернуть символ обратно в поток ввода можно, но это не совсем то, что мы хотим. Мы хотим работать с лексемами, а не возиться с символами. Поэтому хотелось бы, чтобы поток ввода работал с лексемами, а мы бы могли записывать в него уже считанные лексемы.

Давайте предположим, что в нашем распоряжении есть поток лексем — "**Token_stream**" — с именем **ts**. Допустим также, что поток **Token_stream** имеет функцию-член **get()**, возвращающую очередную лексему, и функцию-член **putback(t)**, возвращающую лексему **t** обратно в поток.

Мы реализуем класс **Token_stream** в разделе 6.8, после того как разберемся, как его следует использовать. Имея поток **Token_stream**, можно переписать функцию **expression()** так, чтобы она записывала неиспользованную лексему обратно в поток **Token_stream**.

```
double expression()
{
```

```
    double left = term();       // Считываем и вычисляем Терм
    Token t = ts.get();         // Получаем следующую лексему
     while(true) {
        switch(t.kind) {
        case '+':
            left += term();     // Вычисляем Терм и суммируем
            t = ts.get();
            break;
        case '-':
            left -= term();     // Вычисляем Терм и вычитаем
            t = ts.get();
            break;
        default:
            ts.putback(t);      // Возврат t в поток лексем
            return left;        // Конец: символов + и - нет;
                                // возвращаем ответ
        }
    }
}
```

Кроме того, такие же изменения следует внести и в функцию **term()**.

```
double term()
{
    double left = primary();
    Token t = ts.get();         // Получаем следующую лексему
    while (true)
    {
        switch (t.kind)
        {
        case '*':
            left *= primary();
            t = ts.get();
            break;
        case '/':
        {
            double d = primary();
            if (d == 0)
            {
                error("деление на нуль");
            }
            left /= d;
            t = ts.get();
            break;
        }
        default:
            ts.putback(t);      // Возврат t в поток лексем
            return left;
        }
    }
}
```

В последней функции синтаксического анализатора, `primary()`, достаточно заменить функцию `get_token()` вызовом `ts.get()`; функция `primary()` использует все считанные ею лексемы.

6.7. Испытание второй версии

Итак, мы готовы к испытанию второй версии. Эта вторая версия программы калькулятора (включающая `Token_stream`) находится в файле `calculator01.cpp`. Скомпилируйте ее и запустите. Введите число `2` и символ перехода на новую строку. Нет ответа? Попробуйте ввести еще один символ перехода на новую строку, чтобы убедиться, что компьютер не "завис". По-прежнему нет ответа? Введите число `3` и символ перехода на новую строку. Ответ равен `2`. Попробуйте ввести выражение `2+2` и символ перехода на новую строку. Ответ равен `3`. Вывод на экран выглядит следующим образом:

```
2

3
=2
2+2
=3
```

Гм... Похоже, использование `putback()` в функциях `expression()` и `term()` не решает проблему. Попробуем еще один тест.

```
2 3 4 2+3 2*3
= 2
= 3
= 4
= 5
```

Да! Это правильные ответы! Но последний ответ (`6`) отсутствует. Проблема *предпросмотра* очередной лексемы не решена. Однако на этот раз она заключается не в том, что наш программный код "съедает" символы, а в том, что он не выводит никакой информации, пока не будет введено следующее выражение. Результат вычисления выражения не выводится на экран немедленно; он откладывается до тех пор, пока программа не увидит первую лексему следующего выражения. К сожалению, программа не увидит эту лексему, пока мы не нажмем клавишу <Enter> после следующего выражения. Эта программа на самом деле не настолько плоха, она просто немного медленно реагирует.

Как исправить этот недостаток? Очевидное решение — добавить "команду вывода". Договоримся считать, что после каждого выражения идет точка с запятой, которая завершает выражение и одновременно запускает вывод его результата. Кроме того, добавим в программу "команду выхода",

чтобы позволить программе завершиться штатно, а не использовать для этого неверный ввод. Для этого вполне подойдет символ **q** (первая буква слова "quit" (выход)). Функция **main()** содержит инструкцию

```
while (cin) cout << "=" << expression() << '\n'; // Версия 1
```

Заменим ее запутанным, но более полезным кодом:

```
double val = 0;
while (cin) {
    Token t = ts.get();
    if (t.kind == 'q') break; // 'q' для выхода
    if (t.kind == ';')        // ';' для немедленного вывода
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

Теперь калькулятор действительно можно использовать. Например, мы получим

```
2;
= 2
2+3;
= 5
3+4*5;
= 23
q
```

В данный момент мы имеем хорошую начальную версию калькулятора. Она делает еще не совсем то, что мы хотели, но на ее основе программу можно усовершенствовать. Что очень важно, теперь мы можем устранять недостатки один за другим, сохраняя работоспособность программы.

6.8. Потоки лексем

Прежде чем далее улучшать наш калькулятор, продемонстрируем реализацию класса **Token_Stream**. В конце концов, программа — любая программа — не сможет правильно работать, пока не будут получены корректные данные. Несмотря на то что мы реализуем класс **Token_Stream** в первую очередь, нам не хотелось бы далеко отклоняться от основной темы, пока не сможем получить минимальное решение задачи.

Входной информацией для нашего калькулятора является последовательность лексем, как было показано выше на примере выражения **(1.5+4)*11** (см. раздел 6.3.3). Нам лишь нужна функция, считывающая символы из стандартного потока **cin** и по запросу вводящая в программу следующую лексему. Кроме того, мы видели, что наша программа часто считывает слишком много лексем, поэтому необходимо как-то возвращать

их обратно, чтобы использовать в дальнейшем. Это очень типичная ситуация. Допустим, мы считываем выражение `1.5+4` слева направо. Как без чтения символа `+` можно узнать, что число `1.5` считано полностью? А вдруг мы считываем число `1.55555`? Таким образом, нам нужен поток, который дает лексему при вызове функции `get()` и в который мы сможем возвращать лексему с помощью функции `putback()`. Все сущности в языке C++ имеют тип, поэтому необходимо начать с определения типа `Token_stream`.

Возможно, вы заметили ключевое слово `public:` в определении класса `Token`, приведенном в разделе 6.3.3. В этом разделе для его использования не было очевидных причин. Однако при определении класса `Token_stream` мы обязаны применить его и объяснить его действие. В языке C++ тип, определенный пользователем, часто состоит из двух частей: открытого интерфейса (помеченного как `public:`) и деталей реализации типа (помеченных как `private:`). Идея заключается в том, чтобы отделить то, что необходимо пользователю типа для его использования, от деталей реализации типа, в которые пользователю вникать необязательно.

```
class Token_stream {
public:
     // Пользовательский интерфейс
private:
     // Детали реализации
     // (скрываются от пользователей класса Token_stream)
};
```

Очевидно, что пользователь и разработчик могут быть одним и тем же программистом, но отделение (открытого) интерфейса, предназначенного для пользователей, от (закрытых) деталей реализации, используемых только разработчиками, представляет собой мощное средство структурирования программного кода. Открытый интерфейс должен содержать (только) необходимые пользователю средства, которые обычно представляют собой некоторый набор функций. Закрытая реализация содержит то, что необходимо для реализации этих открытых функций, как правило — данные и функции, связанные с деталями, о которых пользователю незачем знать и которые он не может использовать непосредственно.

Приступим к разработке типа `Token_stream`. Чего пользователь от него ожидает? Очевидно, что нам нужны функции `get()` и `putback()` — именно из-за них мы и ввели понятие потока лексем. Класс `Token_stream` должен создавать объекты класса `Token` из символов, считанных из потока ввода, поэтому нам необходима возможность создавать объекты класса `Token_stream`, способные считывать данные из потока `cin`. Таким образом, простейший вариант класса `Token_stream` выглядит примерно так:

```
class Token_stream {
public:
```

```
    Token_stream();         // Создает объект класса Token_stream,
                            // считывающий данные из потока cin
    Token get();            // Получает объект Token из потока
    void putback(Token t); // Возвращает Token обратно в поток
private:
    // Детали реализации
};
```

Это все, что требуется пользователю для использования класса **Token_stream**. Опытные программисты могут поинтересоваться, почему поток **cin** является единственным возможным источником символов, — просто мы решили вводить символы с клавиатуры. Это решение можно пересмотреть в упражнении, приведенном в главе 7.

Почему мы использовали "длинное" имя **putback()**, а не логичное и краткое имя **put()**? Тем самым мы подчеркнули асимметрию между функциями **get()** и **putback()**: мы возвращаем лексему в поток ввода, а не вставляем ее в поток вывода. Кроме того, функция **putback()** есть в классе **istream**: согласованность имен — полезное свойство. Оно помогает людям запоминать имена функций и избегать ошибок.

Теперь можем создать класс **Token_stream** и использовать его.

```
Token_stream ts;     // Объект класса Token_stream с именем ts
Token t = ts.get(); // Получаем следующий объект Token из ts
// ...
ts.putback(t);      // Возврат объекта Token обратно в ts
```

Это все, что нам нужно, чтобы закончить разработку калькулятора.

6.8.1. Реализация класса Token_stream

Теперь нам необходимо реализовать три функции класса **Token_stream**. Как представить класс **Token_stream**? Иначе говоря, какие данные необходимо хранить в объекте класса **Token_stream**, чтобы он мог выполнить свое задание? Необходима память для лексемы, которая будет возвращена в объект класса **Token_stream**. Для простоты будем считать, что лексемы возвращаются в поток не более чем по одной за один раз. Этого вполне достаточно для нашей программы (а также для очень многих аналогичных программ). Таким образом, нам нужна память для одного объекта класса **Token** и индикатор ее занятости.

```
class Token_stream {
public:
    Token get();            // Получение лексемы (get() определена в §6.8.2)
    void putback(Token t); // Возврат лексемы в поток
private:
    bool full {false};      // Имеется ли лексема в буфере?
    Token buffer;           // Здесь хранится лексема после
                            // возврата с помощью putback()
};
```

Теперь можно определить (написать) две функции-члена. Функция **putback()** проста, так что начнем с нее. Эта функция просто помещает аргумент в буфер **Token_stream**:

```
void Token_stream::putback(Token t)
{
    buffer = t;  // Копируем t в буфер
    full = true; // Буфер заполнен
}
```

Ключевое слово **void** (означающее "ничто") означает, что функция **putback()** не возвращает никакого значения.

Определяя функцию-член вне определения самого класса, мы должны указать, какому классу она принадлежит. Для этого используется обозначение

имя_класса :: *имя_функции-члена*

В данном случае мы определяем функцию-член **putback** класса **Token_stream**.

Почему мы определяем функцию-член вне определения класса? Основной ответ — для ясности: определение класса (в основном) описывает, что класс может делать. Определения функций-членов представляют собой реализации, которые уточняют, как именно класс выполняет то, для чего он предназначен. Мы предпочитаем размещать эти детали там, где они не отвлекают внимание от главного. В идеале каждая логическая единица программы должна помещаться в один экран. Определение класса обычно удовлетворяет этому требованию, если его функции-члены определены в другом месте, а не в определении самого класса.

Если мы хотим гарантировать, что эта функция не будет использована дважды без считывания (с помощью функции **get()**) лексем в промежутке между ее вызовами, можно добавить в функцию соответствующую проверку.

```
void Token_stream::putback(Token t)
{
    if (full) error("putback(): буфер заполнен");
    buffer = t;  // Копируем t в буфер
    full = true; // Буфер заполнен
}
```

Проверка значения **full** проверяет предусловие "В буфере нет ни одного объекта класса **Token**".

Очевидно, что **Token_stream** начинает работу в пустом состоянии, т.е. член **full** должен иметь значение **false** до тех пор, пока впервые не будет вызвана функция **get()**. Мы добиваемся этого путем корректной инициализации члена **full** прямо в определении класса **Token_stream**.

6.8.2. Чтение лексем

Всю реальную работу выполняет функция **get()**. Если в **Token_stream::buffer** нет объекта класса **Token**, функция **get()** должна считать символы из потока **cin** и составить из них объект класса **Token**.

```
Token Token_stream::get()
{
    if (full) {       // Есть готовый объект Token?
        full = false; // Убираем его из буфера и возвращаем
        return buffer;
    }
    char ch;
    cin >> ch;        // Заметим, что оператор >> пропускает
                      // пробельные символы
    switch (ch) {
    case ';':         // Для вывода
    case 'q':         // Для выхода
    case '(': case ')': case '+':
    case '-': case '*': case '/':
        return Token{ch};  // Символ представляет сам себя
    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    {   cin.putback(ch);   // Возвращаем цифру во входной поток
        double val;
        cin >> val;        // Считываем число с плавающей точкой
        return Token{'8',val}; // '8' означает лексему "число"
    }
    default:
        error("Неверная лексема");
    }
}
```

Давайте рассмотрим функцию **get()** подробно. Сначала мы проверяем, нет ли в буфере готового объекта класса **Token**. Если есть, то мы просто возвращаем его.

```
if (full) {       // Есть готовый объект Token?
    full = false; // Убираем его из буфера и возвращаем
    return buffer;
}
```

Только если переменная **full** равна **false** (т.е. в буфере нет лексем), нам придется иметь дело с символами. В таком случае считываем символ и соответствующим образом его обрабатываем. Мы распознаем скобки, операторы и числа. Любой другой символ становится причиной вызова функции **error()**, которая прекращает выполнение программы.

```
default:
    error("Неверная лексема");
```

Функция `error()` описана в разделе 5.6.3 и доступна в заголовочном файле `std_lib_facilities.h`.

Необходимо решить, как представлять разные виды лексем, т.е. выбрать значения для члена `kind` класса `Token`. Для простоты и облегчения отладки мы решили обозначать скобки и операторы соответствующими им символами. В результате обработка скобок и операторов выполняется очень просто:

```
case '(': case ')': case '+':
case '-': case '*': case '/':
    return Token{ch};  // Символ представляет сам себя
```

6.8.3. Считывание чисел

Осталось обработать числа. На самом деле это непросто. Действительно, как узнать значение числа `123`? Хорошо, оно равно `100+20+3`. А что вы скажете о числе `12.34`? Следует ли принимать числа в научной записи, такие как `12.34e5`? Мы могли бы затратить на решение этой задачи часы и дни, но, к счастью, это необязательно. Потоки ввода в языке C++ распознают литералы и сами умеют переводить их в тип `double`. Все, что нам нужно, — как-то заставить поток `cin` сделать это в функции `get()`.

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch);    // Возвращаем цифру во входной поток
    double val;
    cin >> val;         // Считываем число с плавающей точкой
    return Token{'8',val}; // '8' означает лексему "число"
}
```

Мы достаточно произвольно решили, что число в классе `Token` будет представлять символ `'8'`.

Как узнать, что на вход поступило число? Ну, зная по опыту или изучая справочник по языку C++ (например, в приложении A), можно установить, что числовой литерал должен начинаться с цифры или символа `'.'` (десятичной точки). Итак, именно этот факт и следует проверить. Далее мы бы хотели, чтобы поток `cin` прочитал это число, но есть проблема — мы уже считали первый его символ (цифру или десятичную точку), поэтому дальнейшее чтение лексемы приведет к ошибке. Можно попытаться скомбинировать значение первого символа со значением оставшейся части; например, если некто ввел число `123`, мы считываем `1`, а затем из потока `cin` считывается число `23`, и нам остается лишь сложить `100` и `23`. Увы, это подходит только для самых тривиальных случаев. Но, к счастью (и не случайно), поток `cin` работает точно так же, как поток `Token_stream`, в том

смысле, что мы можем вернуть символ в него. В результате, вместо того чтобы выполнять сложные арифметические действия, мы возвращаем первый символ назад в поток `cin` и позволяем ему считать все число.

Пожалуйста, обратите внимание на то, как мы снова и снова избегаем сложностей и вместо этого находим простые решения, часто полагаясь на возможности библиотеки. В этом и заключается смысл программирования: постоянно искать простые решения. Иногда в шутку говорят: "Хороший программист ленив". Это означает, что мы должны быть ленивыми (в хорошем смысле): зачем писать длинную программу, если можно написать короткую?

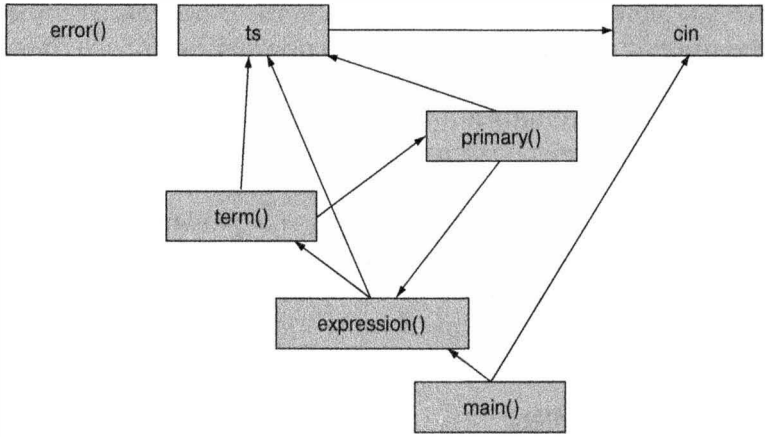
6.9. Структура программы

Как утверждает пословица, за деревьями трудно увидеть лес. Аналогично легко потерять смысл программы, просматривая все ее функции, классы и т.д. Давайте рассмотрим программу, опуская все мелкие детали.

```
#include "std_lib_facilities.h"

class Token        { /* ... */ };
class Token_stream { /* ... */ };

void  Token_stream::putback(Token t) { /* ... */ }
Token Token_stream::get()            { /* ... */ }

Token_stream ts;      // Предоставляет get() и putback()
double expression(); // Объявление для использования
                     // в функции primary()

double primary()    { /* ... */ } // Работа с числами и скобками
double term()       { /* ... */ } // Работа с * и /
double expression() { /* ... */ } // Работа с + и -

int main()          { /* ... */ } // Основной цикл
                                   // и обработка ошибок
```

Порядок следования объявлений важен. Мы не можем использовать имя, пока оно не объявлено, поэтому объект `ts` должен быть объявлен до того, как будет вызвана функция `ts.get()`, а функция `error()` должна быть объявлена до функций синтаксического анализатора, поскольку они ее используют. В графе вызовов имеется интересный цикл: функция `expression()` вызывает функцию `term()`, которая вызывает функцию `primary()`, которая, в свою очередь, вызывает функцию `expression()`.

Эту ситуацию можно проиллюстрировать графически (удалив вызовы функции `error()`, которая вызывается в каждой из функций).



Это значит, что мы не можем просто определить эти три функции: не существует такого порядка их следования, при котором каждая вызываемая функция была бы определена до ее вызова. Таким образом, необходимо объявление, которое не было бы определением. Мы решили объявить "наперед" функцию `expression()`.

Работает ли эта программа? Работает, если придать этому слову определенный смысл. Она компилируется, запускается, правильно вычисляет выражения и выдает осмысленные сообщения об ошибках. Но работает ли она так, как мы от нее ожидаем? Не удивительно, что на самом деле она работает не совсем так, как хотелось бы. Мы испытали первую версию в разделе 6.6 и удалили серьезные ошибки. Однако вторая версия (из раздела 6.7) не намного лучше, хотя в этом нет ничего страшного (это было вполне предсказуемо). Программа вполне успешно выполняет свою основную задачу и позволяет проверить основные идеи. В этом смысле она вполне успешна, но как только вы станете работать с ней, возникнет масса проблем.

ПОПРОБУЙТЕ

Запустите программу, посмотрите, как она работает, и попытайтесь понять, почему она работает именно так.

Задание

Это задание связано с рядом модификаций программы с ошибками, которые превратят довольно бесполезный код в полезную программу.

1. Откройте файл `calculator02buggy.cpp`. Скомпилируйте его. Найдите и исправьте несколько ошибок. Этих ошибок в тексте книги нет. Най-

дите три логические ошибки, специально вставленные в `calculator02buggy.cpp`, и удалите их, чтобы калькулятор давал верные результаты.

2. Измените символ команды выхода с `q` на `x`.
3. Измените символ команды вывода с `;` на `=`.
4. Добавьте в функцию `main()` приветствие.

```
"Добро пожаловать в программу-калькулятор!
Вводите выражения с числами с плавающей точкой."
```

5. Усовершенствуйте это приветствие, указав, какие операторы можно использовать, как выводить результаты на экран и как выйти из программы.

Контрольные вопросы

1. Что означает выражение "Программирование — это понимание"?
2. В главе подробно описан процесс разработки программы-калькулятора. Проведите краткий анализ того, что должен делать калькулятор.
3. Как разбить задачу на небольшие части?
4. Почему следует начинать с небольшой, ограниченной версии программы?
5. Чем плохо "расползание возможностей"?
6. Перечислите три основных этапа разработки программного обеспечения.
7. Что такое прецедент использования?
8. Для чего предназначено тестирование?
9. Следуя материалу главы, опишите разницу между `Термом`, `Выражением`, `Числом` и `Первичным_выражением`.
10. В данной главе входная информация разбивается на компоненты: `Термы`, `Выражения`, `Первичные_выражения` и `Числа`. Сделайте это для арифметического выражения `(17+4)/(5-1)`.
11. Почему в программе нет функции `number()`?
12. Что такое лексема?
13. Что такое грамматика? Что такое правило грамматики?
14. Что такое класс? Для чего мы используем классы?
15. Как указать значение для члена класса по умолчанию?
16. Почему в функции `expression()` в операторе `switch` по умолчанию предусмотрен возврат лексемы в поток?
17. Что такое "предпросмотр"?
18. Что делает функция `putback()` и чем она полезна?

19. Почему в функции `term()` трудно реализовать операцию вычисления остатка (деление по модулю) `%`?
20. Для чего используются два данных-члена класса `Token`?
21. Зачем (иногда) члены класса разделяются на закрытые и открытые?
22. Что произойдет в классе `Token_stream`, если в буфере есть лексема и вызывается функция `get()`?
23. Зачем в инструкцию `switch` в функции `get()` класса `Token_stream` добавлены символы `';'` и `'q'`?
24. Когда следует начинать тестирование программы?
25. Что такое тип, определенный пользователем? Зачем он нужен?
26. Что такое интерфейс пользовательского типа?
27. Почему следует полагаться на библиотечные коды?

Термины

`class`	интерфейс	реализация
`private`	лексема	синтаксический анализатор
`public`	прецедент использования	функции-члены
анализ	проектирование	член класса
грамматика	прототип	
данные-члены	псевдокод	

Упражнения

1. Выполните упражнения из раздела **Попробуйте**, если вы не сделали этого раньше.
2. Добавьте в программу возможность обработки скобок `{}` и `()`, чтобы выражение `{(4+5)*6}/(3+4)` стало корректным.
3. Добавьте оператор вычисления факториала: для его представления используйте восклицательный знак `!`. Например, выражение `7!` означает `7*6*5*4*3*2*1`. Присвойте оператору `!` более высокий приоритет по сравнению с операторами `*` и `/`, т.е. `7*8!` должно означать `7*(8!)`, а не `(7*8)!`. Начните с модификации грамматики, чтобы учесть оператор с более высоким приоритетом. Для согласования со стандартным математическим определением факториала значение `0!` примите равным `1`. *Указание*: калькулятор работает с числами типа `double`, но факториал определен только для целочисленных значений. Поэтому для вычисления `x!` присвойте значение `x` переменной типа `int` и вычисляйте факториал этой переменной.

4. Определите класс **Name_value**, хранящий строку и значение. Повторите упр. 19 из главы 4, используя вместо двух векторов вектор **vector<Name_value>**.
5. Добавьте в "английскую грамматику" из раздела 6.4.1 артикль **the** так, чтобы можно было описать предложения вида "The birds fly but the fish swim".
6. Напишите программу, проверяющую корректность предложений в соответствии с правилами "английской грамматики" из раздела 6.4.1. Будем считать, что каждое предложение заканчивается точкой, ., окруженной пробельными символами. Например, фраза **birds fly but the fish swim .** является предложением, а фразы **but birds fly but the fish swim** (пропущена точка) и **birds fly but the fish swim.** (перед точкой нет пробела) — нет. Для каждого введенного предложения программа должна просто отвечать "Да" или "Нет". *Указание*: не возитесь с лексемами, просто считывайте данные в объект **string** с помощью оператора **>>**.
7. Напишите грамматику для побитовых логических выражений. Побитовое логическое выражение напоминает арифметическое за исключением того, что в нем используются не арифметические, а логические операторы: **!** (отрицание), **~** (дополнение), **&** (и), **|** (или) и **^** (исключающее или). Каждый оператор выполняет операции над всеми битами своих целочисленных операндов (см. раздел 25.5). Операторы **!** и **~** являются префиксными унарными операторами. Оператор **^** имеет более высокий приоритет, чем оператор **|** (так же, как оператор ***** имеет более высокий приоритет, чем оператор **+**), так что выражение **x|y^z** означает **x|(y^z)**, а не **(x|y)^z**. Оператор **&** имеет более высокий приоритет, чем оператор **^**, так что выражение **x^y&z** означает **x^(y&z)**.
8. Повторите упр. 12 из главы 5 (игра "Коровы и быки"), используя четыре буквы, а не четыре цифры.
9. Напишите программу, считывающую цифры и составляющую из них целые числа. Например, число **123** считывается как последовательность символов **1**, **2** и **3**. Программа должна вывести на экран сообщение "**123 — это 1 сотня, 2 десятки и 3 единицы**". Число должно быть выведено как значение типа **int**. Обрабатывайте числа, состоящие из одной цифры, двух, трех и четырех цифр. *Указание*: для того чтобы получить число **5** из символа **'5'**, вычтите из него символ **'0'**, иначе говоря, **'5'-'0'==5**.
10. Перестановка — это упорядоченное подмножество некоторого множества. Например, допустим, что вы хотите подобрать код к сейфу. Существует шестьдесят возможных чисел, а вам необходимо выбрать

три числа для комбинации. В этом случае существует `P(60,3)` перестановок, где количество перестановок определяется по формуле

$$P(a,b)=\frac{a!}{(a-b)!},$$

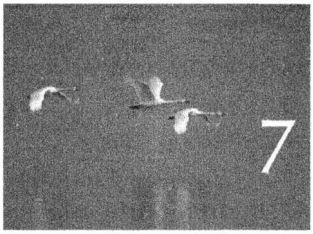
где символ `!` означает факториал. Например, `4!` — это `4*3*2*1`. Сочетания напоминают перестановки, с тем отличием, что в них порядок следования элементов не имеет значения. Например, если вы делаете слоистое мороженое и хотите использовать три разных вкуса из пяти, имеющихся в наличии, вам все равно, когда вы используете ванильное мороженое — в начале или в конце, вы просто хотите использовать его. Формула для вычисления количества сочетаний имеет следующий вид:

$$C(a,b)=\frac{P(a,b)}{b!}.$$

Разработайте программу, запрашивающую у пользователя два числа и что для них необходимо вычислить (количество перестановок или сочетаний), а также выводящую требуемый результат на экран. Сначала выполните анализ перечисленных требований к программе. Точно сформулируйте, что именно должна делать программа. Затем переходите к этапу проектирования. Напишите псевдокод программы и разбейте его на части. Программа должна выполнять проверку ошибок. Убедитесь, что все неправильные входные данные приводят к появлению осмысленных сообщений об ошибках.

Послесловие

Придание смысла входным данным — одна из фундаментальных частей программирования. Каждая программа в той или иной степени сталкивается с этой проблемой. Осмысление чего бы то ни было, непосредственно созданного человеком, относится к одной из труднейших задач. Например, нерешенными задачами остаются многие аспекты распознавания голоса. Простые варианты этой задачи, такие как наш калькулятор, можно решить с помощью грамматики, описывающей входные данные.



7

Завершение программы

Цыплят по осени считают.

Поговорка

Создание программы предполагает последовательное уточнение того, что вы хотите сделать и как вы желаете это выразить. В предыдущей главе мы разработали начальную версию программы, имитирующей работу калькулятора. Теперь мы ее улучшим. Завершение программы, т.е. ее настройка с учетом потребностей пользователей, подразумевает улучшение пользовательского интерфейса, выполнение серьезной работы по устранению ошибок, добавление новых полезных функциональных возможностей и перестройку кода для повышения его ясности и возможности модификаций.

7.1. Введение

Когда программа в первый раз начинает работать нормально, вы, скорее всего, находитесь на полпути к финишу. Для больших программ и программ, неправильная работа которых может привести к тяжелым последствиям, даже "полпути" — слишком оптимистическая оценка. Когда программа "в принципе работает", начинается самое интересное! В этот момент у нас, наконец-то, имеется достаточное количество работающего кода, чтобы приступить к экспериментам с идеями.

В данной главе мы продемонстрируем ход мыслей профессионального программиста, пытающегося улучшить калькулятор из главы 6. Обратите внимание на то, что вопросы о программе и рассматриваемые проблемы сами по себе намного интереснее, чем создаваемый калькулятор. Мы покажем, как эволюционирует реальная программа под влиянием требований и ограничений и как программист может постепенно ее улучшить.

7.2. Ввод и вывод

В начале главы 6 мы решили, что приглашение пользователю ввести данные должно выглядеть следующим образом:

```
Выражение:
```

Кроме того, вывод результатов должен был предваряться словом `Результат:`. Торопясь поскорее запустить программу, мы постоянно забываем об этих деталях. И это совершенно естественно. Мы не можем все время помнить обо всем сразу, поэтому регулярно обнаруживаем, что что-то забыли.

В некоторых задачах первоначальные требования измениться не могут. Как правило, такая жесткая политика ведет к излишне ограниченным программам для решения поставленных задач. Так что давайте рассмотрим, что можно сделать в предположении, что мы можем изменять спецификации программы. Действительно ли мы хотим, чтобы программа выводила

на экран слова **Выражение:** и **Результат:**? Почему мы так решили? Просто "подумать" тут вряд ли поможет. Мы должны проверить разные варианты и выбрать лучший.

В текущей версии для входной строки

```
2+3; 5*7; 2+9;
```

программа выводит следующие результаты:

```
= 5
= 35
= 11
```

Если добавить слова **Выражение:** и **Результат:**, получим следующее:

```
Выражение: 2+3; 5*7; 2+9;
Результат : 5
Выражение: Результат: 35
Выражение: Результат: 11
Выражение:
```

Мы уверены, что кому-то нравится один стиль, а кому-то — другой. В таких ситуациях мы можем предоставить выбор пользователям, но для данной простой задачи это было бы излишним, так что мы должны принять собственное волевое решение. По нашему мнению, слова **Выражение:** и **Результат:** слишком загромождают экран и сбивают с толку. Из-за них сами выражения и результаты занимают меньшую часть выводимого на экран, а поскольку именно они являются предметом нашего внимания, ничто не должно отвлекать нас от них. С другой стороны, если каким-то образом не отделить входную информацию, которую вводит пользователь, и результаты, вычисленные компьютером, получится путаница. Во время первоначальной отладки для индикации результата мы использовали символ **=**. Нам нужно такое же короткое приглашение для ввода. Часто в качестве такого приглашения используется символ **>**.

```
> 2+3;
= 5
> 5*7;
= 35
>
```

Это выглядит намного лучше, и мы можем приступать к небольшим изменениям основного цикла в функции **main()**.

```
double val = 0;
while (cin) {
    cout << "> ";                    // Вывод приглашения
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "= " << val << '\n'; // Вывод результатов
```

```
    else
        ts.putback(t);
    val = expression();
}
```

К сожалению, при вводе нескольких выражений в одной строке вывод выглядит запутанно.

```
> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>
```

Основная проблема заключается в том, что мы не подумали о возможности ввести в одной строке сразу несколько выражений. В действительности хотелось бы получить следующий вариант ввода и вывода:

```
> 2+3; 5*7; 2+9;
= 5
= 35
= 11
>
```

Это выглядит правильно, но, к сожалению, неясно, как этого добиться. Сначала посмотрим на функцию `main()`. Существует ли способ выводить символ `>` тогда и только тогда, когда он не следует за символом `=` немедленно? Мы не можем знать, когда это происходит! Мы должны вывести символ `>` до вызова функции `get()`, но мы не знаем, действительно ли функция `get()` считывает новые символы или просто возвращает объект класса `Token`, созданный из символов, уже считанных с клавиатуры. Иначе говоря, для того чтобы внести это улучшение, нам придется переделать поток `Token_stream`.

Так что пока будем считать, что текущий вариант достаточно хорош. Если мы будем вынуждены модифицировать поток `Token_stream`, то пересмотрим это решение. Неразумно вносить в программу серьезные структурные изменения, чтобы добиться минимальных улучшений, а мы ведь даже еще не протестировали калькулятор как следует.

7.3. Обработка ошибок

Первое, что необходимо сделать, получив "в основном работающую" программу,— попытаться "сломать" ее, т.е. испытать различные входные данные в надежде, что они приведут к неправильной работе программы. Мы говорим "в надежде", потому что основная задача на этом этапе — найти как можно больше ошибок, чтобы исправить их до того, как их обнаружит кто-то другой. Если вы приступите к проверке с

убеждением "Моя программа работает, и я не делаю никаких ошибок!", то пропустите слишком много ошибок и будете очень огорчены, когда все же их обнаружите. Вы должны подвергать сомнению то, что делаете! Правильная позиция формулируется так: "Я «сломаю» ее! Я умнее, чем любая программа, даже моя собственная!" Давайте введем в калькулятор мешанину правильных и неправильных выражений, например следующую.

```
1+2+3+4+5+6+7+8
1-2-3-4
!+2
;;;
(1+3;
(1+);
1*2/3%4+5-6;
();
1+;
+1
1++;
1/0
1/0;
1++2;
-2;
-2;;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

ПОПРОБУЙТЕ

Введите некоторые из этих проблематичных выражений в калькулятор и постарайтесь понять, сколько существует способов вызвать неправильное поведение программы. Сможете ли вызвать ее крах, т.е. обойти обработку ошибок и вызвать машинную ошибку? Мы не думаем, что это у вас получится. А сможете ли вы выйти из программы без осмысленного сообщения об ошибке? Конечно.

Технически этот процесс называется *тестированием* (testing). Существуют люди, занимающиеся испытанием программ профессионально. Тестирование — очень важная часть разработки программного обеспечения, и оно может быть весьма увлекательным занятием. Более подробно тестирование рассматривается в главе 26. Имеется один большой вопрос: "Существует ли способ систематического тестирования программы, позволяющий найти все ошибки?" Универсального ответа на этот вопрос, т.е. ответа, который относился бы ко всем программам, нет. Однако, если отнестись к тестированию серьезно, можно неплохо протестировать многие

программы. При попытках систематического тестирования программы в случае, когда стратегия выбора тестов не является полной, следует выполнить и так называемые "странные" тесты, наподобие следующего:

```
Mary had a little lamb
srtvrqtiewcbet7rewaewre-wqcntrretewru754389652743nvcqnwq;
!@#$%^&*()~:;
```

Однажды при тестировании компилятора я взял за правило подавать на вход компилятора электронные отчеты о его собственных ошибках — заголовки писем, пояснения пользователей и все остальное. Это не было "разумно", поскольку этого "никто никогда не будет делать". Однако в идеале программа должна отлавливать все ошибки, а не только разумные, и в результате вскоре компилятор стал очень устойчивым к "странному вводу".

Первую действительно раздражающую вещь мы обнаружили, когда при тестировании калькулятора выяснилось, что окно закрывается сразу после ввода наподобие

```
+1;
()
!+2
```

Небольшое размышление (или отслеживание выполнения программы) показывает, что проблема заключается в том, что окно закрывается сразу же после вывода сообщения об ошибке. Это происходит потому, что наш механизм удержания окна открытым ожидает ввода символа. Однако во всех трех случаях, упомянутых выше, программа обнаруживала ошибку до того, как считывала все символы, поэтому в строке ввода всегда оставался как минимум один не считанный символ. Программа не могла уберечь этот остающийся от выражения символ от ввода в ответ на приглашение закрыть окно. Этот "остаточный" символ и приводил к преждевременному закрытию окна.

Справиться с этой ошибкой можно, модифицировав функцию **main()** (см. раздел 5.6.3).

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    // keep_window_open():
    cout << "Чтобы закрыть окно, введите символ ~\n";
    char ch;
    for(char ch;cin >> ch;) // Чтение до ввода символа ~
        if (ch=='~') return 1;
    return 1;
}
```

По сути, мы заменили функцию **keep_window_open()** собственным кодом. Заметим, что проблема останется нерешенной, если символ **~** окажется

считанным из входной строки после возникновения ошибки, но это маловероятно.

Обнаружив эту проблему, мы написали вариант функции **keep_window_open()**, в качестве аргумента которой передается строка, с помощью ввода которой пользователь может закрыть окно. Таким образом, более простое решение выглядит так:

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
```

Теперь ввод наподобие

```
+1
!1~~
()
```

приведет к выводу калькулятором соответствующего сообщения об ошибке, а затем приглашения

```
Чтобы выйти, введите ~~
```

и ожиданию, пока пользователь не введет строку **~~**.

Входные данные для калькулятора вводятся с клавиатуры. Это затрудняет тестирование: каждый раз, внося улучшение, мы должны ввести множество контрольных примеров (каждый раз заново!), чтобы убедиться, что программа по-прежнему работает. Было бы лучше где-то хранить контрольные примеры и вызывать их одной командой. Некоторые операционные системы (в частности, Unix) упрощают эту задачу, позволяя потоку **cin** считывать данные из файла без модификации программы, а потоку **cout** — направлять данные в файл. В других случаях мы должны модифицировать программу так, чтобы она использовала файл (см. главу 10).

Рассмотрим примеры.

```
1+2; q
```

и

```
1+2 q
```

Мы хотели бы вывести результат (3) и выйти из программы. Забавно, что строка

```
1+2 q
```

приводит к этому результату, а более корректная строка

```
1+2; q
```

вызывает ошибку "**Ожидается первичное выражение**". Где следует искать эту ошибку? Конечно, в функции **main()**, там, где обрабатываются символы **;** и **q**. Мы добавили инструкции "вывод" и "выход" просто для того, чтобы поскорее получить работающий вариант калькулятора (раздел 6.7), а теперь расплачиваемся за эту поспешность. Рассмотрим еще раз следующий фрагмент:

```
double val = 0;
while (cin) {
    cout << "> ";
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "= " << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

Если обнаруживаем точку с запятой, то переходим к функции **expression()**, не проверяя наличия символа **q**. Эта функция, в первую очередь, вызывает функцию **term()**, которая вызывает функцию **primary()**, которая обнаруживает символ **q**. Он не является **Первичным_выражением**, поэтому мы и получаем сообщение об ошибке. Значит, после проверки на наличие точки с запятой мы должны выполнить проверку наличия символа **q**. Размышляя над этой проблемой, мы также почувствовали необходимость несколько упростить логику, поэтому окончательный вариант функции **main()** выглядит так:

```
int main()
try
{
    while (cin) {
        cout << "> ";
        Token t = ts.get();
        while (t.kind == ';') t=ts.get(); // "Съедает" ';'
        if (t.kind == 'q') {
            keep_window_open();
            return 0;
        }
        ts.putback(t);
        cout << "= " << expression() << '\n';
    }
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
```

```
}
catch (...) {
    cerr << "исключение \n";
    keep_window_open("~~");
    return 2;
}
```

Это повышает надежность обработки ошибок. После этого можно продолжить поиск новых путей улучшения калькулятора.

7.4. Отрицательные числа

Проверив калькулятор, легко убедиться, что он не слишком элегантно обрабатывает отрицательные числа. Например, выражение

```
-1/2
```

является ошибочным. Для того чтобы калькулятор верно вычислил это значение, мы должны были бы написать

```
(0-1)/2
```

Это слишком плохо.

Обычно такие проблемы выявляются на поздних этапах отладки и тестирования. Только тогда можно увидеть, что на самом деле делает программа, и получить информацию, позволяющую уточнить исходные идеи. Планируя проект, целесообразно сэкономить время и извлечь выгоду из наших уроков. Очень часто первая версия поставляется пользователям без необходимых уточнений из-за напряженного расписания и жесткой стратегии управления, которая не позволяет вносить исправления в спецификацию на поздних этапах разработки. Поздние добавления — это кошмар менеджера. В действительности же, когда программа уже достаточно работоспособна, но еще не готова к поставке, еще не поздно вносить дополнения; это самый первый момент, когда можно учесть опыт работы с программой. Реалистичное расписание должно учитывать это обстоятельство.

В данном случае необходимо внести исправления в грамматику, чтобы предусмотреть унарный минус. На первый взгляд, легче всего внести исправления в пункт **Первичное_выражение**. Сейчас он выглядит так:

```
Первичное выражение:
    Число
    "(" Выражение ")"
```

Нам требуется, чтобы этот пункт выглядел примерно таким образом:

```
Первичное выражение:
    Число
    "(" Выражение ")"
```

```
    "-" Первичное выражение
    "+" Первичное выражение
```

Мы добавили унарный плюс, поскольку он есть в языке C++. Если есть унарный минус, то легче реализовать унарный плюс, чем объяснить его бесполезность. Код, реализующий **Первичное_выражение**, принимает следующий вид:

```
double primary()
{
    Token t = ts.get();
    switch (t.kind) {
    case '(': // Обработка правила '(' Выражение ')'
        {
            double d = expression();
            t = ts.get();
            if (t.kind != ')') error("требуется ')'");
            return d;
        }
    case '8':            // '8' представляет число
        return t.value; // возвращаем значение числа
    case '-':
        return - primary();
    case '+':
        return primary();
    default:
        error("требуется первичное выражение");
    }
}
```

Этот код настолько прост, что работает с первого раза.

7.5. Остаток от деления: %

Обдумывая проект калькулятора, мы хотели, чтобы он вычислял остаток от деления — оператор `%`. Однако этот оператор не определен для чисел с плавающей точкой, поэтому мы отказались от этой идеи. Настало время вернуться к ней снова. Это должно быть простым делом.

1. Добавляем `%` как **Token**.
2. Преобразовываем число типа **double** в тип **int**, чтобы впоследствии применить к нему оператор `%`.

Мы знаем, как оператор `%` работает с целочисленными операндами, например:

```
> 2%3;
= 2
> 3%2;
= 1
```

```
> 5%3;
= 2
```

Но как работать с операндами, не являющимися целыми числами?

```
> 6.7%3.3;
```

Каким должен быть результат? Идеального технического ответа на этот вопрос нет. Однако операцию деления по модулю часто определяют и для операндов с плавающей точкой. В частности, `x%y` можно определить как `x%y=x-y*int(x/y)`, так что `6.7%3.3==6.7-3.3*int(6.7/3.3)`, т.е. `0.1`. Это легко сделать с помощью функции стандартной библиотеки `fmod()` из заголовочного файла `<cmath>` (§24.8). Изменим функцию `term()` так, чтобы она включала следующий фрагмент:

```
case '%':
{
    double d = primary();
    if (d == 0) error("%: деление на нуль");
    left = fmod(left,d);
    t = ts.get();
    break;
}
```

В библиотеке `<cmath>` содержится масса стандартных математических функций, таких как `sqrt(x)` (квадратный корень `x`), `abs(x)` (абсолютное значение `x`), `log(x)` (натуральный логарифм `x`) и `pow(x,y)` (`x` в степени `y`).

Можно также просто запретить операцию `%` для аргументов с плавающей точкой. Для этого следует проверить, содержит ли хотя бы один из операндов дробную часть, и, если содержит, вывести соответствующее сообщение об ошибке. Задача обеспечения целочисленных операндов для оператора `%` является вариантом проблемы сужающего преобразования (§3.9.2 и §5.6.4), так что ее можно решить с использованием приведения `narrow_cast`:

```
case '%':
{
    int i1 = narrow_cast<int>(left);
    int i2 = narrow_cast<int>(primary());
    if (i2 == 0) error("%: деление на нуль");
    left = i1%i2;
    t = ts.get();
    break;
}
```

Для простого калькулятора вполне подходят оба решения.

7.6. Приведение кода в порядок

Мы уже внесли несколько изменений в программу. По нашему мнению, все они являются улучшениями, но код начинает постепенно запутываться. Настало время пересмотреть его, чтобы понять, что можно сделать проще и короче, где следует добавить необходимые комментарии и т.д. Другими словами, мы не закончим программу до тех пор, пока она не примет вид, понятный для любого, кто захочет заняться ею. За исключением практически полного отсутствия комментариев исходный текст калькулятора не так уж и плох, но его нужно немного привести в порядок.

7.6.1. Символические константы

Оглядываясь назад, вспомним, что с помощью символа `'8'` мы решили обозначать объекты класса **Token**, содержащие числовое значение. На самом деле совершенно не важно, какой именно символ будет обозначать числовые лексемы; нужно лишь, чтобы он отличался от индикаторов других разновидностей лексем. Однако пока что наш код выглядит довольно странно, и мы должны вставить в него несколько комментариев.

```
case '0':              // Символ '0' обозначает число
    return t.value;    // Возвращаем число
case '-':
    return -primary();
```

Честно говоря, здесь мы также сделали несколько ошибок, напечатав `'0'`, а не `'8'`, поскольку забыли, какое значение выбрали для этой цели. Иначе говоря, непосредственное использование символа `'8'` в коде, предназначенном для обработки объектов класса **Token**, является непродуманным, трудным для запоминания и уязвимым для ошибок; символ `'8'` представляет собой одну из так называемых "магических констант", о которых мы предупреждали в разделе 4.3.1. Так что мы должны ввести символическое имя константы, которая будет представлять число.

```
const char number = '8'; // t.kind==number означает, что t — число
```

Модификатор **const** сообщает компилятору, что мы определили объект, который не будет изменяться; например, выражение **number='0'** должно вызвать сообщение об ошибке. При таком определении переменной **number** нам больше не нужно использовать символ `'8'` явным образом. Фрагмент кода функции **primary()**, приведенный выше, теперь принимает следующий вид:

```
case number:
    return t.value; // Возвращает значение числа
case '-':
    return -primary();
```

Этот фрагмент не требует комментариев. Совершенно необязательно сообщать в комментариях то, что очевидно из самого кода. Повторяющиеся комментарии, объясняющие нечто, часто свидетельствуют о том, что программа требует улучшения.

Аналогично код функции **Token_stream::get()**, распознающий числа, принимает следующий вид:

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{
    cin.putback(ch); // Вернуть цифру во входной поток
    double val;
    cin >> val;      // Считать число с плавающей точкой
    return Token(number,val);
}
```

Можно было бы придумать символические имена для всех лексем, но это излишне. В конце концов, символы **'('** и **'+'** самоочевидны. Анализируя лексемы, легко понять, что лишь символы **';'** для инструкции "вывод" (или "конец выражения") и **'q'** для инструкции "выход" выбраны произвольным образом. Почему не **'p'** или **'e'**? В более крупной программе такая малопонятная и произвольная система обозначения рано или поздно вызовет проблемы, поэтому введем следующие переменные:

```
const char quit = 'q';  // t.kind==quit означает, что t —
                        // лексема выхода
const char print = ';'; // t.kind==print означает, что t —
                        // лексема печати
```

Теперь цикл в функции **main()** можно переписать так:

```
while (cin) {
    cout << "> ";
    Token t = ts.get();
    while (t.kind == print) t=ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << "= " << expression() << '\n';
}
```

Введя символические имена для команд "вывод" и "выход", мы сделали код понятнее. Кроме того, теперь тот, кто будет читать текст функции **main()**, не будет гадать, как кодируются эти инструкции. Например, не удивительно, если мы решим изменить представление инструкции "выход" на символ **'e'** (от слова "exit"). Для этого не потребуется вносить изменения в функцию **main()**.

Теперь в глаза бросаются строки "> " и "= ". Почему мы используем эти "магические" литералы в своем исходном тексте? Как программист, впервые читающий текст функции **main()**, сможет догадаться об их предназначении? Может быть, стоит добавить комментарий? Это может оказаться удачной идеей, но использование символического имени более эффективно.

```
const string prompt = "> ";
const string result = "= "; // Используется для указания на то,
                            // что далее следует результат
```

Если нам в дальнейшем понадобится изменить приглашение или индикатор результата, будет достаточно просто изменить эти константы. Теперь цикл выглядит следующим образом:

```
while (cin) {
    cout << prompt;
    Token t = ts.get();
    while (t.kind ==print) t=ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << result << expression() << '\n';
}
```

7.6.2. Использование функций

Функции должны отражать структуру программы, и их имена должны обеспечивать логическое разделение кода на отдельные части. В этом отношении наша программа до сих пор не вызывала нареканий: функции **expression()**, **term()** и **primary()** непосредственно отражают наше понимание грамматики, а функция **get()** выполняет ввод и распознавание лексем. Тем не менее анализ функции **main()** показывает, что ее можно разделить на две логически разные части.

1. Функция **main()** описывает общую логическую структуру: начало программы, конец программы и обработку "фатальных" ошибок.
2. Функция **main()** выполняет цикл вычислений.

В идеале любая функция должна выполнять единственное логическое действие (см. раздел 4.5.1). Если функция **main()** выполняет оба указанные выше действия, это затеняет структуру программы. Поэтому очевидным решением является вынос цикла вычислений в отдельную функцию **calculate()**.

```
void calculate() // Цикл вычисления выражения
{
    while (cin) {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t=ts.get(); // Удаление вывода
        if (t.kind == quit) return;
        ts.putback(t);
        cout << result << expression() << '\n';
    }
}

int main()
try {
    calculate();
    keep_window_open(); // Удерживает консольное окно открытым
    return 0;
}
catch (runtime_error& e) {
    cerr << e.what() << endl;
    keep_window_open("~~");
    return 1;
}
catch (...) {
    cerr << "исключение \n";
    keep_window_open("~~");
    return 2;
}
```

Этот код намного четче отражает структуру программы, а следовательно, его проще понять.

7.6.3. Размещение кода

Поиск некрасивого кода приводит нас к следующему фрагменту:

```
switch (ch) {
case 'q': case ';': case '%': case '(': case ')':
case '+': case '-': case '*': case '/':
    return Token{ch}; // Каждый символ представляет сам себя
```

Этот код был неплох, пока мы не добавили символы `'q'`, `';'` и `'%'`, но теперь он стал непонятным. Код, который трудно читать, часто скрывает ошибки. И конечно, потенциальные ошибки есть и в этом фрагменте! Для их выявления необходимо разместить каждый раздел `case` в отдельной строке и добавить пару комментариев. Так функция `Token_stream::get()` принимает следующий вид:

```
Token Token_stream::get()
    // Чтение символов из cin и составление Token
{
```

```
    if (full) {             // Проверка наличия Token в буфере
        full = false;
        return buffer;
    }
    char ch;
    cin >> ch;              // Заметим, что оператор >> пропускает
    switch (ch) {           // пробельные символы
    case quit:
    case print:
    case '(':
    case ')':
    case '+':
    case '-':
    case '*':
    case '/':
    case '%':
        return Token{ch}; // Каждый символ представляет сам себя
    case '.':               // Число с плавающей точкой может
                            // начинаться с точки
    // Числовой литерал:
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
    {
        cin.putback(ch);  // Возврат цифры во входной поток
        double val;
        cin >> val;         // Чтение числа с плавающей точкой
        return Token{number,val};
    }
    default:
        error("Неверная лексема");
    }
}
```

Разумеется, можно было бы поместить в отдельной строке раздел **case** для каждой цифры, но это ничуть не делает исходный текст понятнее. Кроме того, в этом случае функция **get()** по размеру могла бы выйти за пределы экрана. В идеале любая функция должна полностью помещаться в один экран; очевидно, что ошибку легче скрыть в коде, который находится за границами экрана. Расположение кода на экране имеет важное значение.

Кроме того, обратите внимание на то, что мы заменили простой символ **'q'** символическим именем **quit**. Это повышает удобочитаемость кода и гарантирует появление сообщения компилятора об ошибке при попытке выбрать для **quit** значение, уже связанное с другой лексемой.

При приведении исходного текста в порядок можно непреднамеренно внести новые ошибки. После наведения порядка в исходном тексте всегда следует проводить повторное тестирование кода. Еще лучше проводить его после внесения каждого улучшения, так что, если что-то пойдет неправильно, вы всегда сможете вспомнить, что именно вы сделали только что. Помните: тестировать надо пораньше и почаще.

7.6.4. Комментарии

При разработке кода мы включили в него несколько комментариев. Хорошие комментарии — важная часть программирования. В рабочей суматохе мы часто забываем об этом. Момент, когда мы возвращаемся к коду для приведения его в порядок, лучше всего подходит для проверки того, что комментарии удовлетворяют следующим требованиям.

1. Корректность (вы могли изменить код, оставив старый комментарий).
2. Достаточность для их читателя (редкое качество).
3. Немногословность (чтобы не отпугнуть читателя).

Подчеркнем важность последнего свойства: все, что необходимо сказать в коде, следует выражать средствами самого языка программирования. Избегайте комментариев, описывающих то, что и так совершенно понятно для тех, кто знает язык программирования. Рассмотрим пример.

```
x = b+c; // Складываем переменные b и c и
         // присваиваем результат переменной x
```

Такие комментарии часто можно встретить в учебниках, но они нужны лишь для того, чтобы объяснить свойства языка, которые еще не известны читателям.

Комментарии нужны для тех вещей, которые плохо выражаются средствами языка программирования. Примером такой ситуации является выражение намерения программиста: код означает лишь то, что программа делает на самом деле, но он ничего не может сказать читателю о действительных намерениях программиста (см. раздел 5.9.1). Посмотрите на исходный текст калькулятора. В нем кое-чего не хватает: функции показывают, как мы обрабатываем выражения и лексемы, но ничего не говорят (помимо самого кода) о том, что именно мы считаем выражением и лексемой. Такие комментарии лучше всего разместить в грамматике или в документации к калькулятору.

```
/*
  Простой калькулятор

  История версий:

  Переработан Бьярне Страуструпом в мае 2007 г.
  Переработан Бьярне Страуструпом в августе 2006 г.
  Переработан Бьярне Страуструпом в августе 2004 г.
  Разработан Бьярне Страуструпом (bs@cs.tamu.edu) весной 2004 г.

  Эта программа реализует основные выражения калькулятора.
```

```
  Ввод осуществляется из потока cin; вывод — в поток cout.

  Грамматика для ввода:

  Инструкция:
      Выражение
      Вывод
      Выход

  Вывод:
      ;

  Выход:
      q

  Выражение:
      Терм
      Выражение + Терм
      Выражение - Терм

  Терм:
      Первичное_выражение
      Терм * Первичное_выражение
      Терм / Первичное_выражение
      Терм % Первичное_выражение

  Первичное_выражение:
      Число
      ( Выражение )
      - Первичное_выражение
      + Первичное_выражение

  Число:
      Литерал_с_плавающей_точкой

  Ввод из потока cin через Token_stream с именем ts.
*/
```

Здесь мы использовали блочный комментарий, который начинается символами **/*** и заканчивается символами ***/**. В реальной программе история версий может содержать сведения о том, какие именно изменения были внесены и какие улучшения были сделаны.

Обратите внимание на то, что эти комментарии не являются кодом. Приведенная грамматика несколько упрощена: сравните правило для **Инструкции** с тем, что на самом деле выполняется в программе (например, в исходном тексте в следующем разделе). Этот комментарий ничего не говорит о цикле в функции **calculate()**, позволяющем выполнять несколько вычислений в рамках одного сеанса работы программы. Мы вернемся к этой проблеме в разделе 7.8.1.

7.7. Восстановление после ошибок

Почему мы прекращаем работу калькулятора, когда обнаруживаем ошибку? В свое время это казалось простым и очевидным решением, но почему? Почему бы не вывести сообщение об ошибке и не продолжить работу? В конце концов, мы часто делаем опечатки, и такие ошибки не означают, что мы решили не продолжать вычисления. Итак, попробуем восстановить работу программы после обнаружения ошибки. Это, по существу означает, что мы должны перехватить исключение и продолжить работу после того, как выполним все необходимые действия по восстановлению корректного состояния калькулятора.

До сих пор все ошибки были представлены исключениями и обрабатывались функцией `main()`. Если мы хотим обеспечить восстановление после ошибок, то функция `calculate()` должна перехватывать исключения и пытаться восстановить нормальную работу, прежде чем приступить к вычислению следующего выражения.

```
void calculate()
{
    while (cin)
    try {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t=ts.get(); // Отбрасывание
        if (t.kind == quit) return;         // команд вывода
        ts.putback(t);
        cout << result << expression() << '\n';
    }
    catch (exception& e) {
        cerr << e.what() << '\n';    // Вывод сообщения об ошибке
        clean_up_mess();
    }
}
```

Мы просто поместили тело цикла `while` в блок `try`, который выводит сообщение об ошибке и восстанавливает нормальное состояние калькулятора. После этого работу можно продолжать по-прежнему.

Что означает выражение "восстанавливает нормальное состояние"? В принципе готовность к выполнению вычислений после исправления ошибки означает, что все данные находятся в корректном и предсказуемом состоянии. В калькуляторе единственными данными за пределами отдельных функций является `Token_stream`. Все, что нам надо сделать, — гарантировать отсутствие лексем, связанных с прекращенными вычислениями и способных помешать будущим вычислениям. Рассмотрим пример.

```
1++2*3; 4+5;
```

Эти входные данные вызывают ошибку, и лексемы `2*3; 4+5` остаются в буферах потоков `Token_stream` и `cin` после того, как второй символ `+` приведет к генерации исключения. У нас есть две возможности.

1. Удалить все лексемы из `Token_stream`.
2. Удалить из `Token_stream` все лексемы, связанные с текущим вычислением.

В первом случае мы отбрасываем из потоков все (включая `4+5;`), а во втором — только `2*3;`, оставляя `4+5;` для последующего вычисления. Оба выбора разумны, и в то же время оба могут кого-то удивить. Обе альтернативы реализуются одинаково просто. Мы предпочли второй вариант, поскольку он упрощает тестирование.

Таким образом, нам нужно выполнить чтение потока до тех пор, пока мы не встретим точку с запятой. На первый взгляд, все очень просто. У нас есть функция `get()`, выполняющая чтение, так что можно написать функцию `clean_up_mess()` следующим образом:

```
void clean_up_mess() // Наивное решение
{
    while (true) {   // Пропускаем все, пока
                     // не обнаружим команду вывода
        Token t = ts.get();
        if (t.kind == print) return;
    }
}
```

К сожалению, эта функция не всегда работает хорошо. Почему? Рассмотрим следующий вариант:

```
1@z; 1+3;
```

Символ `@` приводит нас к разделу `catch` в цикле `while`. Затем для выявления следующей точки с запятой вызывается функция `clean_up_mess()`. Она, в свою очередь, вызывает функцию `get()` и считывает символ `z`. Это приводит к следующей ошибке (поскольку символ `z` не является корректной лексемой), в результате мы оказываемся в обработчике `catch(...)` в функции `main()` и выходим из программы. Теперь у нас нет шансов вычислить лексему `1+3`.

Можно было бы попробовать более сложные блоки `try` и `catch`, но это внесет в программу еще большую путаницу. Ошибки в принципе трудно обрабатывать, а ошибки, возникающие при обработке других ошибок, обрабатывать еще труднее. Поэтому стоит попытаться найти способ удалять из `Token_stream` символы, который не может генерировать исключения. Единственный путь для ввода данных в калькулятор пролегает через функцию `get()`, и он может, как мы только что выяснили, порождать

исключения. Таким образом, необходима новая операция. Очевидное место для ее размещения — класс **Token_stream**.

```
class Token_stream {
public:
    Token get();            // Считывает лексему
    void putback(Token t); // Возвращает лексему в поток
    void ignore(char c);    // Отбрасывает символы до
                            // символа с включительно
private:
    bool full {false};      // Есть ли лексема в буфере?
    Token buffer;           // Хранит лексему, возвращенную
                            // вызовом putback()
};
```

Функция **ignore()** должна быть членом класса **Token_stream**, так как она должна иметь доступ к его буферу. Искомый символ мы решили передавать функции **ignore()** в качестве аргумента — в конце концов, объект класса **Token_stream** не обязан знать, что именно калькулятор рассматривает в качестве необходимого символа для восстановления после ошибки. Мы решили, что этот аргумент должен быть символом, потому что не хотим рисковать, работая с лексемами (мы уже видели, что при этом происходит). Итак, мы получаем следующую функцию:

```
void Token_stream::ignore(char c)
    // Символ с представляет разновидность лексем
{
    // Сначала проверяем буфер:
    if (full && c==buffer.kind) {
        full = false;
        return;
    }
    full = false;

    // Теперь проверяем входные данные:
    char ch = 0;
    while (cin>>ch)
        if (ch==c) return;
}
```

В этом коде сначала происходит проверка буфера. Если в буфере есть символ **c**, прекращаем работу, отбрасывая его; в противном случае необходимо считывать символы из потока **cin**, пока не встретится символ **c**.

Теперь функцию **clean_up_mess()** можно написать следующим образом:

```
void clean_up_mess()
{
    ts.ignore(print);
}
```

Обработка ошибок — всегда достаточно сложное дело. Она требует постоянного экспериментирования и тестирования, поскольку крайне трудно представить заранее, какая ошибка может возникнуть в ходе выполнения программы. Защита программы "от дурака" (т.е. от неправильного использования) всегда представляет собой очень сложную техническую задачу. Начинающие об этом обычно не беспокоятся. Качественная обработка ошибок — один из признаков профессионализма.

7.8. Переменные

Поработав над стилем и обработкой ошибок, можно вернуться к попыткам улучшить функциональные возможности калькулятора. Мы получили вполне работоспособную программу; как же ее улучшить? Первыми в списке пожеланий по развитию калькулятора находятся переменные. Использование переменных позволяет лучше выражать длинные вычисления. Кроме того, для научных вычислений хотелось бы иметь встроенные именованные значения, такие как `pi` и `e`, как в научных калькуляторах.

Переменные и константы — основные расширения нашего калькулятора. Они затронут многие части исходного текста. Такие расширения не следует предпринимать без весомых причин и без достаточного времени, выделяемого на эту работу. Здесь мы добавим переменные и константы, поскольку это дает возможность еще раз проанализировать код и освоить некоторые новые методы программирования.

7.8.1. Переменные и определения

Очевидно, что для работы с переменными и встроенными константами калькулятор должен хранить пары (*имя*, *значение*) так, чтобы мы имели доступ к значению по имени. Можно определить следующий класс `Variable`:

```
class Variable {
public:
    string name;
    double value;
};
```

Член класса `name` используется для идентификации объекта класса `Variable`, а член `value` — для хранения значения, соответствующего члену `name`.

Как хранить объекты класса `Variable` так, чтобы их значение можно было найти или изменить, зная значение `name` объекта? Рассматривая встречавшиеся нам до сих пор инструменты, можно дать только один ответ: в виде вектора объектов класса `Variable`.

```
vector<Variable> var_table;
```

В вектор **var_table** можно записать сколько угодно объектов класса **Variable**, а найти их можно, просматривая элементы вектора один за другим. Теперь можно написать функцию **get_value()**, которая ищет заданную строку **name** и возвращает соответствующее ей значение **value**.

```
double get_value(string s)
    // Возвращает значение переменной с именем s
{
    for (const Variable& v : var_table)
        if (v.name == s) return v.value;
    error("get: неопределенная переменная ", s);
}
```

Этот код достаточно прост: он перебирает объекты класса **Variable** в векторе **var_table** (начиная с первого элемента и продолжая до последнего включительно) и проверяет, совпадает ли их член **name** с аргументом **s**. Если строки **name** и **s** совпадают, функция возвращает значение **value** соответствующего объекта.

Аналогично можно определить функцию **set_value()**, присваивающую новое значение члену **value** объекта класса **Variable**.

```
void set_value(string s, double d)
    // Присваивает объекту s типа Variable значение d
{
    for (Variable& v : var_table)
        if (v.name == s) {
            v.value = d;
            return;
        }
    error("set: неопределенная переменная ", s);
}
```

Теперь можно считывать и записывать переменные, представленные в виде объектов класса **Variable** в векторе **var_table**. Как поместить новый объект класса **Variable** в вектор **var_table**? Что пользователь калькулятора должен написать, чтобы определить новую переменную, а позже присвоить ей значение? Можно сослаться на запись, принятую в языке C++.

```
double var = 7.2;
```

Это сработает, но все переменные в калькулятора и так хранят значения типа **double**, поэтому явно указывать этот тип совершенно не обязательно. Можно было бы написать проще.

```
var = 7.2;
```

Что ж, это возможно, но теперь мы не можем отличить определение новой переменной от синтаксической ошибки.

```
var1 = 7.2; // Определение новой переменной с именем var1
var1 = 3.2; // Определение новой переменной с именем var2
```

Ой! Мы же имели в виду `var2 = 3.2;`, но чуть "опечатались" (не в комментарии). Ну, с этим можно жить... но будем следовать традициям языков программирования, в частности языка C++, в которых объявления переменных (с инициализацией) отличаются от присваивания. Мы можем использовать ключевое слово `double`, но для калькулятора нужно что-нибудь покороче, поэтому — следуя другой старой традиции — выбрали ключевое слово `let`.

```
let var = 7.2;
```

Грамматика принимает следующий вид:

```
Вычисление:
    Инструкция
    Вывод
    Выход
    Вычисление Инструкция

Инструкция:
    Объявление
    Выражение

Объявление:
     "let" Имя "=" Выражение
```

Вычисление — это новая (верхнего уровня) продукция грамматики. Она выражает цикл (в функции `calculate()`), который позволяет выполнять несколько вычислений в ходе одного сеанса работы программы. При обработке выражений и объявлений это правило опирается на правило **Инструкция**. Например, инструкцию можно обработать следующим образом:

```
double statement()
{
    Token t = ts.get();
    switch (t.kind) {
    case let:
        return declaration();
    default:
        ts.putback(t);
        return expression();
    }
}
```

Вместо функции `expression()` в функции `calculate()` теперь можно использовать функцию `statement()`.

```
void calculate()
{
    while (cin)
```

```
    try {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print)
            t=ts.get();              // Отбрасывание команд вывода
        if (t.kind == quit) return; // Выход
        ts.putback(t);
        cout << result << statement() << '\n';
    }
    catch (exception& e) {
          cerr << e.what() << '\n'; // Вывод сообщения об ошибке
           clean_up_mess();
    }
}
```

Теперь мы должны написать функцию **declaration()**. Что она должна делать? Она должна убедиться, что после ключевого слова **let** следует **Имя**, а за ним — символ = и **Выражение**. Именно это утверждает грамматика. Что делать с именем переменной? Мы должны добавить в вектор **var_table** типа **vector<Variable>** объект класса **Variable** с заданными строкой **name** и значением выражения. После этого мы сможем извлекать это значение с помощью функции **get_value()** и изменять его с помощью функции **set_value()**. Однако сначала надо решить, что случится, если мы определим переменную дважды. Рассмотрим пример.

```
let v1 = 7;
let v1 = 8;
```

Мы решили рассматривать повторное определение переменной как ошибку. Обычно это просто опечатка. Вероятно, вместо написанного предполагалось ввести

```
let v1 = 7;
let v2 = 8;
```

Определение объекта класса **Variable** с именем **var** и значением **val** состоит из двух логических частей.

1. Проверка наличия в векторе **var_table** объекта класса **Variable** с именем **var**.
2. Добавление пары (**var**,**val**) в вектор **var_table**.

Мы не должны использовать неинициализированные переменные, поэтому определяем функции **is_declared()** и **define_name()**, представляющие эти две логически отдельные операции.

```
bool is_declared(string var)
    // Есть ли переменная var в векторе var_table?
{
    for (const Variable& v : var_table)
        if (v.name == var) return true;
```

```
    return false;
}

double define_name(string var, double val)
    // Добавляем пару (var,val) в вектор var_table
{
    if (is_declared(var)) error(var," повторное объявление");
    var_table.push_back(Variable(var,val));
    return val;
}
```

Добавить новый объект класса **Variable** в вектор типа **vector<Variable>** легко; эту операцию выполняет функция-член вектора **push_back()**.

```
var_table.push_back(Variable(var,val));
```

Вызов **Variable(var,val)** создает соответствующий объект класса **Variable**, а затем функция **push_back()** добавляет этот объект в конец вектора **var_table**. Имея эти функции, в предположении, что мы умеем обрабатывать **let** и **Имя**, написать функцию **declaration()** не очень трудно.

```
double declaration()
    // Считаем, что мы уже встретили ключевое слово "let"
    // Обрабатываем:  Имя = Выражение
    // Объявление переменной с Именем с начальным значением,
    // заданным Выражением
{
    Token t = ts.get();
    if (t.kind != name)
        error ("в объявлении ожидается имя переменной");
    string var_name = t.name;

    Token t2 = ts.get();
    if (t2.kind != '=')
        error("пропущен символ = в объявлении ", var_name);

    double d = expression();
    define_name(var_name,d);
    return d;
}
```

Обратите внимание, что мы возвращаем значение, хранящееся в новой переменной. Это полезно, когда инициализирующее выражение является нетривиальным, например

```
let v = d/(t2-t1);
```

Это объявление определяет переменную **v** и выводит ее значение. Кроме того, вывод переменной упрощает код функции **calculate()**, поскольку функция **statement()** возвращает значение при каждом вызове. Обычно общие правила позволяют сохранить простоту кода, в то время как частные случаи приводят к его усложнению.

Описанный механизм отслеживания переменных часто называют *таблицей символов* (symbol tables). Его можно радикально упростить с помощью контейнера `map` из стандартной библиотеки (раздел 21.6.1).

7.8.2. Использование имен

Все это очень хорошо, но, к сожалению, не работает. Это уже не должно быть для нас сюрпризом. Первый вариант никогда — ну, почти никогда — не работает. В данном случае мы даже не закончили программу — она просто не компилируется. У нас нет лексемы `'='`, но это легко исправить, добавив дополнительный раздел `case` в функцию `Token_stream::get()` (раздел 7.6.3). Но как представить `let` и `Имя` в виде лексем? Очевидно, для того, чтобы распознавать эти лексемы, необходимо модифицировать функцию `get()`. Как? Вот один из способов.

```
const char name = 'a';              // Лексема Имя
const char let = 'L';               // Лексема let
const string declkey = "let";       // Ключевое слово let

Token Token_stream::get()
{
    if (full) {
        full=false;
        return buffer;
    }
    char ch;
    cin >> ch;
    switch (ch) {
            // Как и ранее
    default:
        if (isalpha(ch)) {
            cin.putback(ch);
            string s;
            cin >> s;
            if (s == declkey)
                return Token(let); // Ключевое слово let
            return Token{name,s};
        }
        error("Неверная лексема");
    }
}
```

В первую очередь, обратите внимание на вызов функции `isalpha(ch)`. Этот вызов отвечает на вопрос "Является ли символ `ch` буквой?"; функция `isalpha()` принадлежит стандартной библиотеке и описана в заголовочном файле `std_lib_facilities.h`. Остальные функции классификации символов описаны в разделе 11.6. Логика распознавания имен совпадает с логикой распознавания чисел: находим первый символ соответствующего

типа (в данном случае — букву), а затем возвращаем его в поток с помощью функции `putback()` и считываем все имя целиком с помощью оператора `>>`.

К сожалению, этот код не компилируется; класс `Token` не может хранить строку, поэтому компилятор отвергает вызов `Token{name,s}`. Чтобы справиться с этой проблемой, нам надо изменить определение класса `Token` так, чтобы он мог хранить как `double`, так и `string` и мог работать с тремя разновидностями инициализаторов.

- Только `kind`; например, `Token{'*'}`.
- `kind` и число; например, `Token{number,4.321}`.
- `kind` и имя; например, `Token{name,"pi"}`.

Мы добиваемся этого, вводя три инициализирующие функции, известные как конструкторы, поскольку они конструируют объекты:

```
class Token {
public:
    char kind;
    double value;
    string name;

    // Инициализирует kind символом ch
    Token(char ch):kind{ch} { }

    // Инициализирует kind и value
    Token(char ch, double val):kind{ch},value{val} { }

    // Инициализирует kind и name
    Token(char ch, string n):kind{ch}, name{n} { }
};
```

Конструкторы добавляют важную степень управляемости и гибкости инициализации. Детальнее о конструкторах мы поговорим в главе 9 (разделы 9.4.2, 9.7).

Для представления лексемы `let` мы выбрали букву `'L'`, а само ключевое слово храним в виде строки. Очевидно, что это ключевое слово легко заменить ключевыми словами `double`, `var`, `#` — чем угодно, просто изменив содержимое строки `declkey`, с которой сравнивается строка `s`.

Попытаемся снова протестировать программу. Если ввести следующие выражения, то легко убедиться, что программа работает:

```
let x = 3.4;
let y = 2;
x + y * 2;
```

Однако стоит самую малость изменить ввод, и становится ясно, что программа пока что не работает так, как надо:

```
let x = 3.4;
let y = 2;
x+y*2;
```

Чем же различаются эти примеры? Давайте разберемся, что происходит.

Проблема в том, что мы небрежно определили лексему **Имя**. Мы даже "забыли" включить правило вывода **Имя** в грамматику (раздел 7.8.1). Какие символы могут бы частью имени? Буквы? Конечно. Цифры? Разумеется, если имя не начинается с них. Символ подчеркивания? Символ +? Ну так что?..

Посмотрим на код еще раз. После первой буквы выполняется чтение объекта класса **string** с помощью оператора **>>**. Он считывает все символы, пока не встретит пробел. Так, например, строка **x+y*2;** является единым именем — даже завершающая точка с запятой считывается как часть имени. Это не то, что требовалось, и совершенно неприемлемо.

Что же надо делать вместо этого? Сначала необходимо точно определить, что может представлять собой имя, а затем соответственно изменить функцию **get()**. Ниже приведена вполне разумная спецификация имени: последовательность букв и цифр, начинающаяся с буквы. Например, все перечисленные ниже строки являются именами:

```
a
ab
a1
Z12
asdsddsfdfdasfdsa434RTHTD12345dfdsa8fsd888fadsf
```

А вот следующие строки именами не являются:

```
1a
as_s
#
as*
a car
```

За исключением отброшенного нами символа подчеркивания это правило совпадает с правилом языка C++. Мы можем реализовать его в разделе **default** в функции **get()**.

```
default:
    if (isalpha(ch)) {
        string s;
        s += ch;
        while(cin.get(ch) &&
              (isalpha(ch) || isdigit(ch))) s+=ch;
        cin.putback(ch);
        if (s == declkey)
            return Token{let};    // Ключевое слово объявления
        return Token{name,s};
    }
    error("Неверная лексема");
```

Вместо непосредственного чтения в объект **s** типа **string** считываем символы и записываем их в переменную **s**, пока считанные символы являются буквами или цифрами. Инструкция **s+=ch** добавляет (дописывает) символ **ch** в конец строки **s**. Любопытная инструкция

```
while (cin.get(ch) && (isalpha(ch) || isdigit(ch)) s+=ch;
```

считывает символ в переменную **ch** (используя функцию-член **get()** потока **cin**) и проверяет, является ли он символом или цифрой. Если да, то она добавляет символ **ch** к строке **s** и повторяет считывание. Функция-член **get()** работает как оператор **>>**, с тем отличием, что она не пропускает пробелы по умолчанию.

7.8.3. Предопределенные имена

Теперь можем легко предопределить некоторые из имен. Например, если представить, что наш калькулятор будет использован для научных вычислений, то нам понадобятся имена **pi** и **e**. В каком месте кода их следует определить? В функции **main()** до вызова функции **calculate()** или в функции **calculate()** до цикла. Мы поместим их определения в функцию **main()**, поскольку они не являются частью каких-либо вычислений.

```
int main()
try {
    // Предопределенные имена
    define_name("pi",3.1415926535);
    define_name("e",2.7182818284);

    calculate();

    keep_window_open(); // Удерживает консольное окно открытым
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
catch (...) {
     cerr << "исключение \n";
     keep_window_open("~~");
     return 2;
}
```

7.8.4. Это все?

Вообще-то нет. Мы внесли так много изменений, что теперь необходимо снова протестировать программу, привести в порядок код и пересмотреть

комментарии. Кроме того, можно было бы сделать больше определений. Например, мы "забыли" об операторе присваивания (см. упр. 2), а наличие этого оператора потребует от нас умения как-то различать переменные и константы (см. упр. 3).

Вначале мы отказались от использования именованных переменных в калькуляторе. Теперь, просматривая код их реализации, мы можем выбрать для себя одну из возможных реакций.

1. Реализация переменных была совсем неплохой; она заняла всего около трех десятков строк кода.
2. Реализация переменных потребовала много работы. Она коснулась каждой функции и внесла новую концепцию в проект калькулятора. Она увеличила размер программы на 45%, а ведь мы еще даже не приступали к реализации оператора присваивания.

Если учесть, что это наша первая программа, имеющая определенную сложность, правильной является вторая реакция. В общем случае это справедливо по отношению к любому предложению, увеличивающему на 50%

размер или сложность программы. В такой ситуации целесообразнее написать новую программу, основанную на предыдущей. В частности, если вы создаете программу, как мы создавали калькулятор, т.е. поэтапно, с тестированием на каждом этапе, то результат окажется куда лучшим, чем при попытке написать всю программу целиком и сразу.

Задание

1. Начав с файла **calculator08buggy.cpp**, доведите программу до успешной компиляции
2. Пройдитесь по всей программе и добавьте необходимые комментарии.
3. В ходе комментирования вы обнаружите ошибки (специально вставленные в код, чтобы вы их нашли). Исправьте их; в тексте книги их нет.
4. Тестирование: подготовьте набор вводных данных и используйте их для тестирования калькулятора. Насколько полон ваш список? Что вы ищете? Включите в список отрицательные числа, нуль, очень большие и очень маленькие числа и "странный" ввод.
5. Проведите тестирование и исправьте все ошибки, которые пропустили при комментировании.
6. Добавьте предопределенное имя **k** со значением **1000**.
7. Предусмотрите возможность вычисления функции **sqrt()**, например **sqrt(2+6.7)**. Естественно, значение **sqrt(x)** является квадратным кор-

нем из числа `x`; например, `sqrt(9)` равно 3. Используйте стандартную библиотечную функцию `sqrt()`, доступную посредством заголовочного файла `std_lib_facilities.h`. Не забудьте обновить комментарии и грамматику.

8. Предусмотрите перехват попыток извлечь квадратный корень из отрицательного числа и вывод на экран соответствующих сообщений об ошибке.
9. Предусмотрите возможность использовать функцию `pow(x,i)`, означающую "умножить `x` на себя `i` раз"; например, `pow(2.5,3)` равно `2.5*2.5*2.5`. Аргумент `i` должен быть целым числом. Проверяйте это условие с помощью методики, которая была использована нами при реализации оператора `%`.
10. Измените "ключевое слово объявления" с `let` на `#`.
11. Измените "ключевое слово выхода" с `q` на `exit`. Для этого понадобится строка для кодирования инструкции `quit`, как использовавшаяся нами строка для инструкции `let` в разделе 7.8.2.

Контрольные вопросы

1. Зачем работать над программой, когда ее первая версия уже доказала свою работоспособность? Перечислите причины.
2. Почему выражение "`1+2; q`", введенное в калькулятор, не приводит к выходу из него после обнаружения ошибки?
3. Почему мы использовали символическую константу с именем `number`?
4. Мы разбили функцию `main()` на две разные функции. Что делает новая функция и зачем мы разделили функцию `main()`?
5. Зачем вообще разделять код на несколько функций? Сформулируйте принципы.
6. Зачем нужны комментарии и как они должны быть организованы?
7. Что делает инструкция `narrow_cast`?
8. Как используются символические константы?
9. Почему важна организация кода?
10. Как мы реализовали оператор `%` (остаток) применительно к числам с плавающей точкой?
11. Что и как делает функция `is_declared()`?
12. Представление `let` во входных данных использует несколько символов. Как получить эту лексему как единое целое в модифицированном коде?

13. Сформулируйте правило, определяющее, какие имена могут быть в калькуляторе, а какие не могут.
14. Чем хороша идея постепенной разработки программ?
15. Когда следует начинать тестирование?
16. Когда следует проводить повторное тестирование?
17. Как вы принимаете решение о том, какие функции следует сделать отдельными?
18. Как вы выбираете имена для переменных и функций? Обоснуйте свой выбор.
19. Зачем нужны комментарии?
20. Что следует писать в комментариях, а что нет?
21. Когда следует считать программу законченной?

Термины

восстановление	обработка ошибок	расползание возможностей
история версий	организация кода	символическая константа
комментирование	поддержка и сопровождение	тестирование

Упражнения

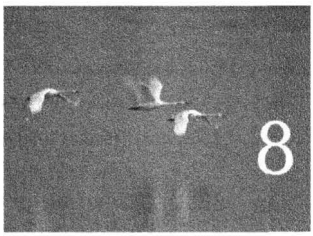
1. Предусмотрите использование символа подчеркивания в именах переменных калькулятора.
2. Реализуйте оператор присваивания `=`, чтобы можно было изменять значение переменной после ее объявления с помощью инструкции `let`. Подумайте, почему эта возможность может быть полезной и к каким проблемам она может привести.
3. Реализуйте именованные константы, которые не могут изменять свои значения. *Указание*: в класс `Variable` необходимо добавить член, позволяющий различать константы и переменные, и проверять его в функции `set_value()`. Если хотите дать пользователю возможность объявлять собственные именованные константы (а не ограничиваться предопределенными константами `pi` и `e`), то необходимо добавить соответствующее обозначение, например `const pi = 3.14;`.
4. Функции `get_value()`, `set_value()`, `is_declared()` и `define_name()` работают с переменной `var_table`. Определите класс `Symbol_table` с членом `var_table` типа `vector<Variable>` и функциями-членами `get()`, `set()`, `is_declared()` и `define()`. Перепишите калькулятор с использованием переменной типа `Symbol_table`.
5. Модифицируйте функцию `Token_stream::get()` так, чтобы, обнаружив символ перехода на следующую строку, она возвращала `Token(print)`.

Это повлечет за собой реализацию просмотра пробельных символов и обработку символа `'\n'`. Для этого можно использовать стандартную библиотечную функцию `isspace(ch)`, возвращающую значение `true`, если символ `ch` является пробельным.

6. Каждая программа должна содержать подсказки для пользователя. Пусть при вводе буквы `H` (в любом регистре) калькулятор выводит на экран инструкцию по эксплуатации.
7. Измените команды `q` и `h` на `quit` и `help` соответственно.
8. Грамматика в разделе 7.6.4 является неполной (мы уже предостерегали вас о том, что не стоит чрезмерно полагаться на комментарии): в ней не определена последовательность инструкций, например `4+4; 5-6;`, и в нее не внесены усовершенствования, описанные в разделе 7.8. Исправьте грамматику. Кроме того, добавьте в первый и во все последующие комментарии программы все, что считаете нужным.
9. Предложите три усовершенствования калькулятора (не упомянутых в главе). Реализуйте одно из них.
10. Модифицируйте калькулятор так, чтобы он работал только с целыми числами; предусмотрите ошибки, возникающие при потере точности и переполнении. *Указание*: воспользуйтесь `narrow_cast` (раздел 7.5).
11. Переработайте две программы, написанные вами при выполнении упражнений к главам 4 и 5. Приведите в порядок их код в соответствии с правилами, описанными в данной главе. Посмотрите, не нашлись ли в процессе наведения порядка какие-либо ошибки в ваших исходных текстах.

Послесловие

Так уж получилось, что на простом примере мы увидели, как работает компилятор. Наш калькулятор анализирует входные данные, разбивая их на лексемы, и распознает их в соответствии с правилами грамматики. Именно этим и занимается компилятор. После анализа входных данных компилятор создает представление (объектный код), который можно выполнить позже. Калькулятор же немедленно вычисляет анализируемые выражения; такие программы в отличие от компиляторов называются интерпретаторами.



8

Технические детали: функции и прочее

Ни один талант не в состоянии преодолеть пристрастия к деталям.

Восьмой закон Леви

В этой и следующей главах мы перейдем от общих рассуждений о программировании к нашему основному инструменту программирования — языку C++. Мы представим технические детали, которые помогут вам получить более широкое и систематическое представление о функциональных возможностях языка C++. Кроме того, эти главы представляют собой обзор многих понятий программирования, введенных ранее, и позволяют исследовать наш инструмент без привлечения новых программных методов и концепций.

8.1. Технические детали

Если бы у нас был выбор, то мы предпочли бы говорить о программировании вообще, а не о свойствах языка программирования. Иначе говоря, намного интереснее изучать, как идеи выражаются в виде кода, чем вникать в технические детали языка программирования, с помощью которого эти идеи воплощаются. Проведем аналогию с естественным языком: ведь никто не станет спорить с тем, что обсуждать стиль и идеи нового романа гораздо увлекательнее, чем изучать грамматику и словарь. Нас намного больше интересуют сами идеи и способы их выражения в виде кода, чем отдельные языковые конструкции.

Однако у нас не всегда есть выбор. Когда вы начинаете программировать, язык программирования можно рассматривать как иностранный, изучать "грамматику и словарь" которого просто необходимо. Именно этим мы и займемся в этой и следующих главах, но читатели должны помнить следующее.

- В первую очередь, мы изучаем программирование.
- Результатом нашей работы являются программы и системы.
- Язык программирования — это лишь средство.

Как ни странно, помнить об этом довольно сложно. Многие программисты не могут устоять перед увлечением мелкими деталями синтаксиса и семантики. В частности, слишком многие ошибочно полагают, что их первый язык программирования — самый лучший. Пожалуйста, не попадайтесь в эту ловушку. Язык C++ во многих отношениях прекрасный язык,

но он не идеален; впрочем, то же самое можно сказать о любом языке программирования.

Большинство понятий, связанных с проектированием и программированием, являются универсальными, и многие из них поддерживаются популярными языками программирования. Это значит, что фундаментальные идеи и методы, изучаемые нами в рамках достаточно продуманного курса программирования, переходят из одного языка в другой. Они могут быть применимы — с разной степенью легкости — ко всем языкам программирования. Однако технические детали языка весьма специфичны. К счастью, языки программирования разрабатываются не в вакууме, поэтому у понятий, которые мы изучаем в нашем курсе, очевидно, есть аналоги в других языках программирования. В частности, язык C++ принадлежит к группе языков, в которую входят такие языки, как C (глава 27), Java и C#, поэтому между ними есть много общего.

Заметьте, что, говоря о технических деталях языка, мы свободно оперируем такими несамоописываемыми именами, как `f`, `g`, `x` и `y`. Мы делаем это, чтобы подчеркнуть техническую природу таких примеров, сделать их очень короткими и не смешивать детали языка с логикой программы. Когда вы видите такие несамоописываемые имена (которые ни в коем случае нельзя использовать в реальном коде), пожалуйста, сосредоточьтесь на технических аспектах кода. Технические примеры обычно содержат код, который просто иллюстрирует правила языка. Если вы скомпилируете и запустите его, то получите множество предупреждений о неиспользуемых переменных, причем некоторые из таких программ вообще не делают никаких осмысленных действий.

Пожалуйста, помните, что эту книгу не следует рассматривать как полное описание синтаксиса и семантики языка C++ (даже по отношению к свойствам, которые мы рассматриваем). Стандарт ISO C++ состоит более чем из 1300 страниц технических описаний; книга *The C++ Programming Language* Страуструпа также содержит свыше 1300 страниц текста, предназначенного для опытных программистов (обе книги содержат описание языка программирования C++ и его стандартной библиотеки). Данное издание не конкурирует с этими книгами ни по охвату материала, ни по полноте его изложения, но соревнуется с ними по понятности текста и по объему времени, которое требуется для его чтения.

8.2. Объявления и определения

Объявление (declaration) — это инструкция, которая вводит имя в область видимости (раздел 8.4),

- устанавливая тип именованной сущности (например, переменной или функции) и

- (необязательно) выполняя инициализацию (например, указывая начальное значение переменной или тело функции).

Вот несколько примеров.

```
int a = 7;              // Переменная типа int
const double cd = 8.7; // Константа с плавающей точкой
                        // двойной точности
double sqrt(double);    // Функция с аргументом типа double,
                        // возвращающая тип double
vector<Token> v;        // Переменная – вектор объектов Token
```

Имя должно быть объявлено до использования в программе на языке C++. Рассмотрим пример.

```
int main()
{
    cout << f(i) << '\n';
}
```

Компилятор выдаст как минимум три сообщения об ошибках, связанных с необъявленными идентификаторами: имена **cout**, **f** и **i** в программе нигде не объявлены. Исправить ошибку, связанную с потоком **cout**, можно, включив в программу заголовочный файл **std_lib_facilities.h**, содержащий его объявление.

```
#include "std_lib_facilities.h" // Содержит объявление cout

int main()
{
    cout << f(i) << '\n';
}
```

Теперь остаются только две ошибки, связанные с неопределенными идентификаторами. При создании реальных программ большинство определений размещают в заголовочных файлах. Именно там определяются интерфейсы полезных функциональных возможностей, которые сами определяются "где-то в другом месте". В принципе, объявление лишь устанавливает, как некая сущность может быть использована; оно определяет интерфейс функции, переменной или класса. Следует помнить об одном очевидном, но невидимом преимуществе такого использования объявлений: мы можем не беспокоиться о том, как определены поток **cout** и его операторы **<<**; мы просто включаем их объявления в программу с помощью директивы **#include**. Мы можем даже не заглядывать в их объявления; из учебников, справочников, примеров программ и других источников нам известно, как используется поток **cout**. Компилятор считывает объявления из заголовочных файлов, необходимых ему для "понимания" кода.

Однако нам по-прежнему необходимо объявить **f** и **i**. И сделать это можно следующим образом:

```
#include "std_lib_facilities.h" // Содержит объявление cout

int f(int);                     // Объявление функции f

int main()
{
    int i = 7;                  // Объявление переменной i
    cout << f(i) << '\n';
}
```

Этот код компилируется без ошибок, поскольку каждое имя было определено, но он не проходит редактирование связей (см. раздел 2.4), поскольку в нем не определена функция `f()`; иначе говоря, мы нигде не указали, что именно делает функция `f()`.

Объявление, которое полностью описывает объявленную сущность, называют *определением* (definition). Рассмотрим пример.

```
int a = 7;
vector<double> v;
double sqrt(double d) { /* ... */ }
```

Каждое определение является объявлением, но только некоторые объявления одновременно являются определениями. Ниже приведены некоторые примеры объявлений, которые не являются определениями; если объявленная сущность использована в исходном тексте, ей должно соответствовать определение, размещенное где-то в другом месте.

```
double sqrt(double); // Здесь функция не имеет тела
extern int a;        // "extern плюс отсутствие инициализатора"
                     // означает, что это не определение
```

Сравнивая определения и объявления, мы придерживаемся общепринятого соглашения, которое устанавливает, что *объявлением* считается только объявление, не являющееся определением, даже если вас немного запутывает такая терминология.

Определение устанавливает, на что именно ссылается имя. В частности, определение переменной выделяет память для этой переменной. Следовательно, ни одну сущность невозможно определить дважды. Рассмотрим пример.

```
double sqrt(double d) {/*...*/} // Определение
double sqrt(double d) {/*...*/} // Ошибка: повторное определение

int a;                          // Определение
int a;                          // Ошибка: повторное определение
```

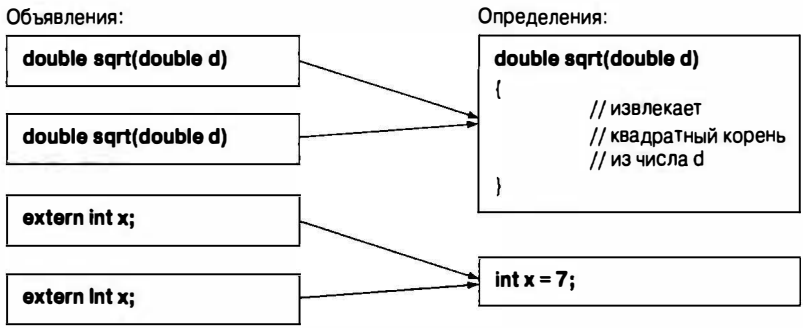
И наоборот, объявление, которое не является одновременно определением, просто сообщает, как можно использовать имя; оно представляет собой интерфейс, но не выделяет памяти и не описывает тело функции.

Следовательно, одно и то же имя можно объявлять несколько раз при условии, что объявления являются согласованными.

```
int x = 7;                          // Определение
extern int x;                       // Объявление
extern int x;                       // Другое объявление

double sqrt(double);                // Объявление
double sqrt(double d) {/*...*/}     // Определение
double sqrt(double);                // Другое объявление sqrt
double sqrt(double);                // Еще одно объявление sqrt

int sqrt(double);                   // Ошибка: несогласованное
                                    // объявление sqrt
```

Почему последнее объявление является ошибкой? Потому что в одной и той же программе не может быть двух функций с именем **sqrt**, принимающих аргумент типа **double** и возвращающих значения разных типов (**int** и **double**).

Ключевое слово **extern**, использованное во втором объявлении переменной **x**, утверждает, что это объявление не является определением. Оно редко бывает нужным. Мы не рекомендуем так поступать, но в принципе такие объявления можно встретить в некоторых исходных текстах, особенно в коде, использующем слишком много глобальных переменных (см. разделы 8.4 и 8.6.2).



Почему в языке C++ предусмотрены как объявления, так и определения? Различие между ними отражает фундаментальное различие между тем, что нам необходимо, чтобы использовать некую сущность (интерфейс), и тем, что нам необходимо, чтобы это нечто делало то, для чего оно предназначено (реализация). Объявление переменной указывает ее тип, но лишь определение создает реальный объект (выделяет память). Объявление функции также устанавливает ее тип (типы аргументов и тип возвращаемого значения), но лишь определение создает тело функции (выполняемые инструкции). Обратите внимание на то, что тело функции хранится в памяти как часть программы, поэтому вполне кор-

ректным будет сказать, что определения функций и переменных выделяют память, а объявления — нет.

Разница между объявлением и определением позволяет разделить программу на части и компилировать их по отдельности. Объявления обеспечивают связь между разными частями программы; при этом не нужно беспокоиться об определениях. Поскольку все объявления должны быть согласованы друг с другом (включая единственное определение), использование имен во всей программе должно быть непротиворечивым. Мы обсудим этот вопрос в разделе 8.3. А здесь мы лишь напомним о синтаксическом анализаторе выражений из главы 6: функция **expression()** вызывает функцию **term()**, которая, в свою очередь, вызывает функцию **primary()**, которая вызывает функцию **expression()**. Поскольку любое имя в программе на языке C++ должно быть объявлено до того, как будет использовано, нет способа ограничиться только определениями этих функций.

```
double expression(); // Это объявление, но не определение

double primary()
{
     // ...
     expression();
     // ...
}

double term()
{
     // ...
     primary();
     // ...
}

double expression()
{
     // ...
     term();
     // ...
}
```

Мы можем расположить эти четыре функции в любом порядке; как бы мы их ни располагали, вызов одной из функций всегда будет предшествовать ее определению. Таким образом, необходимо предварительное объявление. По этой причине мы объявили функцию **expression()** до определения функции **primary()**, и все заработало. Такие циклические вызовы достаточно распространены.

Почему имя должно быть определено до его использования? Не могли бы мы просто потребовать, чтобы компилятор читал программу (как это делаем мы), находил определение и выяснял, какую функцию следует вызвать?

Могли бы, но это приведет к "интересным" техническим проблемам, поэтому мы решили этого не делать. Спецификация языка C++ требует, чтобы определение предшествовало использованию имени (за исключением членов класса; см. раздел 9.4.4). В конце концов, существует обычная практика (не в программировании): когда вы читаете учебники, то ожидаете, что автор определит понятия и обозначения прежде, чем станет их использовать, в противном случае читатели будут вынуждены постоянно догадываться об их смысле. Правило "объявления до использования" упрощает чтение как для людей, так и для компилятора. В программировании существует и вторая причина, по которой это правило имеет большую важность. Программа может состоять из тысяч строк (а то и сотен тысяч), и большинство функций, которые мы хотим вызвать, определены "где-то". Это "где-то" часто является местом, о котором мы просто ничего не знаем. Объявления, которые описывают только способ использования переменной или функции, позволяет нам (и компилятору) не просматривать огромное количество исходных текстов.

8.2.1. Виды объявлений

Программист может объявить множество сущностей в языке C++. Среди них наиболее интересными являются следующие.

- Переменные.
- Константы.
- Функции (см. раздел 8.5).
- Пространства имен (см. раздел 8.7).
- Типы (классы и перечисления; см. главу 9).
- Шаблоны (см. главу 19).

8.2.2. Объявления переменных и констант

Объявление переменной или константы задает ее имя, тип и (необязательно) начальное значение. Рассмотрим пример.

```
int a;                      // Без инициализации
double d = 7;               // Инициализация с помощью =
vector<int> vi(10);         // Инициализация с помощью ()
vector<int> vi2{1,2,3,4}; // Инициализация с помощью {}
```

Полностью грамматику языка можно найти в стандарте ISO C++.

Константы объявляются так же, как переменные, за исключением ключевого слова `const` в качестве части типа и необходимой инициализации.

```
const int x = 7; // Инициализация с помощью =
const int x2{9}; // Инициализация с помощью {}
const int y;     // Ошибка: нет инициализации
```

Причина, по которой константа требует инициализации, очевидна: после объявления константа уже не может изменить свое значение. Как правило, целесообразно инициализировать и переменные; переменная, не имеющая начального значения, способна вызвать недоразумения. Рассмотрим пример.

```
void f(int z)
{
    int x; // Неинициализированная переменная
    // ... здесь нет присваивания переменной x ...
    x = 7; // Присваивание значения переменной x
    // ...
}
```

Этот код выглядит вполне невинно, но что будет, если в первом пропущенном фрагменте, отмеченном многоточием, будет использована переменная **x**? Рассмотрим пример.

```
void f(int z)
{
    int x;   // неинициализированная переменная
    // ... здесь нет присваивания переменной x ...
    if (z>x) {
        // ...
    }
    // ...
    x = 7;   // Присваивание значения переменной x
    // ...
}
```

Поскольку переменная **x** не инициализирована, выполнение оператора **z>x** может привести к неопределенному поведению. Сравнение **z>x** может вести к разным результатам на разных компьютерах и даже на одном и том же компьютере при разных запусках программы. Теоретически оператор **z>x** может даже вызвать прекращение работы программы из-за аппаратной ошибки, но чаще всего ничего не происходит — мы просто получаем непредсказуемые результаты.

Естественно, такое непредсказуемое поведение программы нас не устраивает, но если мы не инициализируем переменные, то в конечном итоге столкнемся с ошибкой. Напомним, что "глупые ошибки" (такие, как использование неинициализированных переменных) происходят из-за спешки или усталости. Как правило, компиляторы пытаются предупредить программистов, но в сложных программах — в которых такие ошибки и появляются чаще всего — они не могут выловить их все. Существуют люди, не привыкшие инициализировать переменные. Часто это происходит потому,

что они учили языки, в которых это не требовалось; вы встретитесь с такими примерами в их программах. Пожалуйста, не усложняйте себе жизнь, забывая инициализировать переменные при их определении.

Мы предпочитаем использовать синтаксис инициализации с фигурными скобками `{}`. Это наиболее обобщенный и наиболее явно указанный "инициализатор". Обычно мы используем его, за исключением очень простой инициализации, где иногда по привычке используется `=`, и пользуемся синтаксисом с круглыми скобками `()`, чтобы указывать количество элементов вектора (см. §17.4.4).

8.2.3. Инициализация по умолчанию

Возможно, вы заметили, что мы часто не инициализируем объекты классов `string`, `vector` и других, например

```
vector<string> v;
string s;
while (cin>>s) v.push_back(s);
```

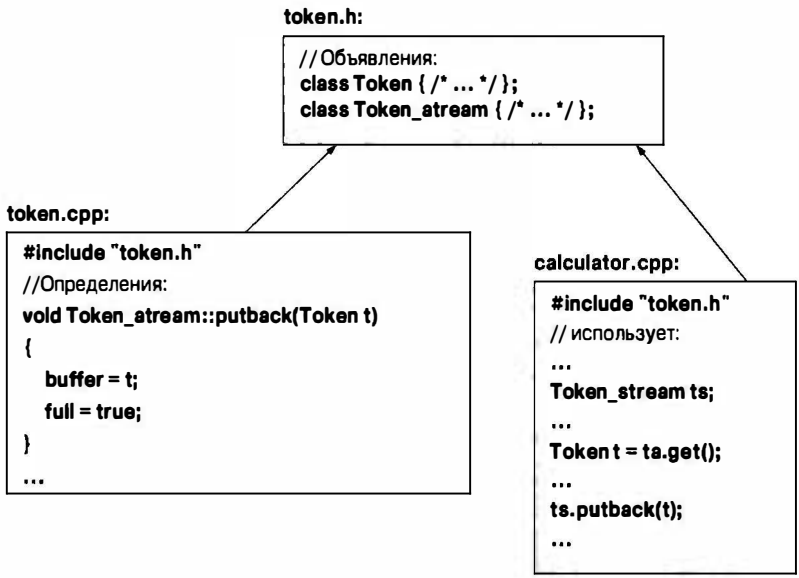
Это не противоречит правилу, в соответствии с которым переменные перед их использованием должны быть проинициализированы. В данном случае, если мы не задаем начальные значения, происходит инициализация строк и векторов по умолчанию. Таким образом, перед входом в цикл вектор `v` пуст (т.е. не содержит элементов), как пуста и строка `s` (имеет значение `""`). Механизм, гарантирующий инициализацию по умолчанию, называется *конструктором по умолчанию* (default constructor).

К сожалению, язык C++ не предусматривает инициализацию по умолчанию для встроенных типов. Лишь глобальные переменные (см. раздел 8.4) по умолчанию инициализируются нулем, но использование глобальных переменных должно быть минимизировано. Наиболее полезные переменные, к которым относятся локальные переменные и члены классов, не инициализируются до тех пор, пока не будет указано их начальное значение (или не задан конструктор по умолчанию). Все, вы предупреждены!

8.3. Заголовочные файлы

Как управлять объявлениями и определениями? В конечном итоге они обязаны быть согласованными, а в реальных программах их могут быть десятки тысяч; программы с сотнями тысяч объявлений тоже не редкость. Как правило, когда вы пишете программу, большинство используемых определений написано не вами. Например, реализации потока `cout` и функции `sqrt()` были написаны много лет назад кем-то другим. Мы просто используем их. Главным средством управления сущностями, определенными где-то в другом месте, в языке C++ являются заголовки. В принципе,

заголовок (header) — это набор объявлений; но так как обычно они записаны в файле, заголовок часто называют *заголовочным файлом* (header file). Такие заголовочные файлы включаются в исходные файлы с помощью директивы **#include**. Например, вы можете решить улучшить организацию исходного кода нашего калькулятора (см. главы 6 и 7), отделив обработку лексем. Для этого можно определить заголовочный файл **token.h**, содержащий объявления, необходимые для использования классов **Token** и **Token_stream**:



Объявления классов **Token** и **Token_stream** находятся в заголовочном файле **token.h**. Их определения находятся в файле **token.cpp**. В языке C++ расширение **.h** обычно используется для заголовочных файлов, а для исходных файлов чаще всего используется расширение **.cpp**. На самом деле в языке C++ расширение файла не имеет значения, но некоторые компиляторы и большинство интегрированных сред разработки программ настаивают на использовании определенных соглашений о расширениях файлов.

В принципе, директива **#include "file.h"** просто копирует объявления из файла **file.h** в ваш файл место, в котором находится эта директива. Например, мы можем написать заголовочный файл **f.h**.

```
// f.h
int f(int);
```

и включить его в файл **user.cpp**

```
// user.cpp
#include "f.h"

int g(int i)
```

```
{
    return f(i);
}
```

При компиляции файла **user.cpp** компилятор выполнит подстановку заголовочного файла и скомпилирует следующий текст:

```
int f(int);
int g(int i)
{
    return f(i);
}
```

Поскольку директива **#include** выполняется компилятором до любой иной обработки файлов, выполняющая ее часть компилятора называется *препроцессором* (preprocessing) (раздел A.17).

Для упрощения проверки согласованности заголовок следует включать как в исходные файлы, использующие объявления, так и в исходные файлы, содержащие определения, соответствующие этим объявлениям. Это позволяет компилятору находить ошибки на самых ранних этапах. Например, представьте себе, что разработчик функции **Token_stream::putback()** сделал ошибки.

```
Token Token_stream::putback(Token t)
{
    buffer.push_back(t);
    return t;
}
```

Этот фрагмент выглядит вполне невинно. К счастью, компилятор находит ошибку, потому что (благодаря директиве **#include**) он видит объявление функции **Token_stream::putback()**. Сравнивая это объявление с соответствующим определением, компилятор выясняет, что функция **putback()** не должна возвращать объект класса **Token**, а переменная **buffer** имеет тип **Token**, а не **vector<Token>**, так что мы не можем использовать функцию **push_back()**. Такие ошибки возникают, когда мы работаем над улучшением кода и вносим изменения, забывая о необходимости согласовывать их с остальной частью программы.

Рассмотрим следующие ошибки:

```
Token t = ts.gett(); // Ошибка: нет члена gett
// ...
ts.putback();        // Ошибка: отсутствует аргумент
```

Компилятор немедленно выдаст ошибку; заголовок **token.h** предоставляет ему всю информацию, необходимую для проверки.

Заголовочный файл **std_lib_facilities.h** содержит объявления стандартных библиотечных средств, таких как **cout**, **vector** и **sqrt()**, а также нескольких простых вспомогательных функций, таких как **error()**,

не являющихся частью стандартной библиотеки. В разделе 12.8 мы продемонстрируем непосредственное использование заголовочных файлов стандартной библиотеки.

Заголовки обычно включаются во многие исходные файлы. Это значит, что заголовок должен содержать только объявления, которые можно дублировать в нескольких файлах (например, объявления функций, классов и числовых констант).

8.4. Область видимости

Область видимости (scope) — это часть исходного текста. Каждое имя объявляется в своей области видимости и является действительным (т.е. находится в области видимости), начиная с точки объявления и заканчивая концом области видимости, в которой оно объявлено. Рассмотрим пример.

```
void f()
{
    g();  // Ошибка: g() не находится (пока) в области видимости
}

void g()
{
    f();  // ОК: Функция f() находится в области видимости
}

void h()
{
    int x = y; // Ошибка: переменная y не находится (пока)
               // в области видимости
    int y = x; // ОК: переменная x находится в области видимости
    g();       // ОК: функция g() находится в области видимости
}
```

Имена, принадлежащие области видимости, видны из вложенных в нее других областей видимости. Например, вызов функции `f()` находится в области видимости функции `g()`, которая "вложена" в глобальную область видимости. Глобальная область видимости не вкладывается ни в какую другую. Правило, утверждающее, что имя должно быть объявлено до того, как будет использовано, по-прежнему действует, поэтому функция `f()` не может вызывать функцию `g()`.

Существует несколько разновидностей областей видимости, которые можно использовать для управления используемыми именами.

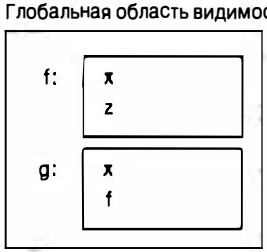
- *Глобальная область видимости* (global scope): область исходного текста, не входящая ни в одну другую область видимости.

- *Пространство имен* (namespace scope): именованная область видимости, вложенная в глобальную область видимости или в другое пространство имен (раздел 8.7).
- *Область видимости класса* (class scope): часть исходного текста, находящаяся в классе (раздел 9.2).
- *Локальная область видимости* (local scope): между фигурными скобками `{...}` блока или в списке аргументов функции.
- *Область видимости инструкции*: например, в цикле `for`.

Основное предназначение области видимости — сохранить локальность имен, чтобы они не пересекались с именами, объявленными в другом месте. Рассмотрим пример.

```
void f(int x)          // Функция f глобальная;
                       // переменная x локальная в функции f
{
    int z = x+7;       // Переменная z локальная
}

int g(int x)           // Функция g глобальная;
                       // переменная x локальная в функции g
{
    int f = x+2;       // Переменная f локальная
    return 2*f;
}
```

Изобразим это графически.



Здесь переменная `x` из функции `f()` отличается от переменной `x` из функции `g()`. Здесь нет никакой неоднозначности, потому что они принадлежат разным областям видимости: переменная `x` из функции `f()` локальна по отношению к функции `f()` и не видна извне, а переменная `x` из функции `g()` локальна по отношению к функции `g()`. Два несовместимых объявления в одной и той же области видимости создают *коллизию имен* (clash). Аналогично переменная `f` объявлена и используется в функции `g()` и (очевидно) не является функцией `f()`.

Рассмотрим логически эквивалентный, но более реальный пример использования локальной области видимости.

```
int max(int a, int b) // Функция max является глобальной,
                      // а переменные a и b — локальными
{
    return (a>=b) ? a : b;
}

int abs(int a)        // Переменная a не имеет никакого
                      // отношения к функции max()
{
    return (a<0) ? -a : a;
}
```

Функции `max()` и `abs()` имеются в стандартной библиотеке, поэтому их не нужно писать самому. Конструкция `?:` называется *арифметической инструкцией if* (arithmetic if) или *условным выражением* (conditional expression). Значение инструкции `(a>=b)?a:b` равно `a`, если `a>=b`, и `b` — в противном случае. Условное выражение позволяет не писать длинный код наподобие следующего:

```
int max(int a, int b) // Функция max является глобальной,
                      // а переменные a и b — локальными
{
    int m;            // Переменная m является локальной
    if (a>=b)
        m = a;
    else
        m = b;
    return m;
}
```

Таким образом, за исключением глобальной области видимости все остальные области видимости обеспечивают локальность имен. В большинстве случаев локальность имени является полезным свойством, поэтому к нему надо стремиться изо всех сил. Когда я объявляю свои переменные, функции и прочее в функциях, классах, пространствах имен и так далее, то не хочу, чтобы они совпадали с вашими именами. Помните: реальные программы содержат *многие* тысячи именованных сущностей. Для возможности поддержки и сопровождения таких программ большинство имен должны быть локальными.

Рассмотрим более крупный технический пример, иллюстрирующий ситуацию, в которой имена выходят за пределы области видимости в конце инструкций и блоков (включая тела функций).

```
// Здесь переменные r, i и v не видны
class My_vector
{
    vector<int> v; // Переменная v находится в области
                   // видимости класса
public:
```

```
    int largest()
    {
        int r = 0; // Переменная r локальна (минимальное
                   // неотрицательное целое число)
        for (int i = 0; i < v.size(); ++i) // Переменная i принадлежит
            r = max(r, abs(v[i]));  // области видимости цикла for

        // Здесь переменная i не видна
        return r;
    }
    // Здесь переменная r не видна
};
// Здесь переменная v не видна

int x; // Глобальные переменные — по возможности избегайте их
int y;

int f()
{
    int x;        // Локальная переменная, скрывающая
                  // глобальную переменную x
    x = 7;        // Локальная переменная x
    {
        int x = y; // Локальная переменная x инициализируется
                   // глобальной переменной y и скрывает
                   // локальную переменную x из предыдущего блока
        ++x;       // Переменная x из предыдущей строки
    }
    ++x;           // Переменная x из первой строки функции f()
    return x;
}
```

Где только можно, избегайте ненужных вложений и сокрытий. Помните девиз "Будь проще!"

Чем больше область видимости имени, тем длиннее и информативнее должно быть ее имя: хуже имен, чем **x**, **y** и **z**, для глобальных переменных не придумаешь. Основная причина, по которой следует избегать глобальных переменных, заключается в том, что трудно понять, какие функции изменяют их значения. В больших программах это практически невозможно. Представьте, что вы пытаетесь отладить программу, и выясняется, что глобальная переменная принимает неожиданное значение. Какая инструкция присвоила ей это значение? Почему? В какой функции? Как это узнать? Функция, присвоившая неправильное значение данной переменной, может находиться в исходном файле, который вы даже никогда не видели! В хорошей программе может быть лишь несколько (скажем, одна или две) глобальных переменных — если они вообще в ней есть. Например, калькулятор, описанный в главах 6 и 7, содержит две глобальные переменные: поток лексем **ts** и таблицу символов **names**.

Обратите внимание на то, что большинство конструкций в языке C++, определяющих пространства имен, являются вложенными.

- Функции в классах: функции-члены (раздел 9.4.2).

```
class C {
public:
    void f();
    void g() // Функция-член может быть определена в классе
    {
        // ...
    }
    // ...
};

void C::f() // Функция-член может быть определена вне класса
{
    // ...
}
```

Это наиболее распространенный и полезный вариант.

- Классы в других классах: члены-классы (они же — вложенные классы).

```
class C {
public:
    struct M {
        // ...
    };
    // ...
};
```

Это допустимо только в сложных классах; помните, что в идеале класс должен быть маленьким и простым.

- Классы в функциях: локальные классы.

```
void f()
{
    class L {
        // ...
    };
    // ...
}
```

Избегайте таких конструкций; если вам нужен локальный класс, скорее всего, ваша функция слишком велика.

- Функции в других функциях: локальные функции (или вложенные функции).

```
void f()
{
    void g() // Некорректно
    {
        // ...
```

```
    }
    // ...
}
```

В языке C++ это не допускается; не пытайтесь так поступать — компилятор вам этого не позволит.

- Блоки в функциях и в других блоках (они же — вложенные блоки).

```
void f(int x, int y)
{
    if (x>y) {
    // ...
    }
    else {
        // ...
        {
            // ...
        }
    // ...
    }
}
```

Вложенные блоки неизбежны, но они свидетельствуют о повышенной сложности программы и легко скрывают ошибки.

В языке C++ существует еще одно средство — пространство имен **name space**, которое используется исключительно для выражения областей видимости (раздел 8.7).

Следите за выравниванием фигурных скобок, обозначающих вложение. При отсутствии выравнивания вложенные конструкции становятся совершенно нечитаемыми:

```
// Опасно уродливый код
struct X {
void f(int x) {
struct Y {
int f() { return 1; } int m; };
int m;
m=x; Y m2;
return f(m2.f()); }
int m; void g(int m) {
if (m) f(m+2); else {
g(m+2); }}
X() { } void m3() {
}

void main() {
X a; a.f(2);}
};
```

Обычно такой неудобочитаемый код скрывает ошибки. Если вы используете интегрированные среды разработки программ, они пытаются авто-

матически выравнивать фигурные скобки (в соответствии со своими настройками). Кроме того, существуют программы для "улучшения форматирования", которые переформатируют исходный код в файле (часто предлагая пользователю выбор из разных вариантов форматов). Однако окончательная ответственность за удобочитаемость кода лежит на его авторе.

8.5. Вызов функции и возврат значения

Функции позволяют нам выражать действия и вычисления. Если мы хотим сделать что-то, заслуживающее имени, то пишем функцию. В языке C++ есть операторы (такие, как + и *), с помощью которых можно вычислить новые значения по операндам, входящим в выражение, и инструкции (такие, как **for** и **if**), позволяющие управлять порядком вычислений. Для того чтобы организовать код из этих примитивов, у нас есть функции.

Для выполнения своего предназначения функции принимают аргументы и, как правило, возвращают результат. В этом разделе мы рассмотрим, как аргументы передаются в функции.

8.5.1. Объявление аргументов и тип возвращаемого значения

Функции в языке C++ используются для именования и представления вычислений и действий. Объявление функции состоит из типа возвращаемого значения, за которым следуют имя функции и список формальных аргументов в круглых скобках, например:

```
double fct(int a, double d); // Объявление функции fct (без тела)
double fct(int a, double d)
    { return a*d; }          // Определение функции fct
```

Определение содержит тело функции (инструкции, выполняемые при ее вызове), в то время как объявление, не являющееся определением, просто завершается точкой с запятой. Формальные аргументы часто называют *параметрами* (parameters). Если вы не хотите, чтобы функция имела аргументы, не указывайте параметры, например:

```
int current_power(); // Эта функция не принимает аргументов
```

Если хотите, чтобы функция не возвращала никаких значений, укажите вместо типа возвращаемого значения ключевое слово **void**, например:

```
void increase_power(int level); // Функция ничего не возвращает
```

Здесь ключевое слово **void** означает "ничего не возвращает".

Параметры можно как именовать, так и не именовать. Главное, чтобы объявления и определения были согласованы друг с другом. Рассмотрим пример.

```
// Поиск строки s в векторе vs;
// vs[hint] может быть подходящим местом для начала поиска
// Возвращает индекс найденного совпадения; -1 означает,
// что совпадениене найдено
int my_find(vector<string> vs,      // Именованные аргументы
            string s, int hint);
int my_find(vector<string>,         // Неименованные аргументы
            string, int);
```

В объявлениях имена формальных аргументов не являются логически обязательными, просто они очень полезны при написании хороших комментариев. С точки зрения компилятора второе объявление функции `my_find()` так же правильно, как и первое: оно содержит всю информацию, необходимую для ее вызова.

Обычно в определении все аргументы имеют имена. Рассмотрим пример.

```
int my_find(vector<string> vs, string s, int hint)
// Поиск строки s в векторе vs, начиная с позиции hint
{
    if (hint < 0 || vs.size() <= hint) hint = 0;
    for (int i = hint; i < vs.size(); ++i) // Поиск, начиная
                                           // с позиции hint
        if (vs[i]==s) return i;
    if (0 < hint) {  // Если строка не была найдена,
                     // повторяем поиск до позиции hint
        for (int i = 0; i < hint; ++i)
            if (vs[i]==s) return i;
    }
    return -1;
}
```

Переменная `hint` немного усложняет код, но она введена на основании предположения, что пользователю может быть примерно известно, где в векторе находится строка. Однако представим себе, что мы уже некоторое время использовали `my_find()`, а затем выяснилось, что пользователи редко применяют переменную `hint`, так что она лишь снижает производительность программы. В таком случае переменная `hint` больше не нужна, но "во внешнем мире" уже имеется масса вызовов функции `my_find()` с аргументом `hint`. Переписывать весь этот внешний код мы не хотим (или не можем, поскольку это чужой код), поэтому мы не хотим изменять объявление функции `my_find()`. Вместо этого мы просто не будем использовать последний аргумент. Поскольку мы его не используем, оставим его безымянным.

```
int my_find(vector<string> vs, string s, int) // 3-й аргумент
{                                             // не используется
    for (int i = 0; i < vs.size(); ++i)
        if (vs[i]==s) return i;
```

```
    return -1;
}
```

Полностью грамматика определений функций изложена в стандарте ISO C++.

8.5.2. Возврат значения

Функция возвращает вычисленное значение с помощью инструкции **return**.

```
T f() // Функция f() возвращает объект класса T
{
     V v;
     // ...
     return v;
}

T x = f();
```

Здесь возвращаемое значение — это именно то значение, которые мы получили бы при инициализации переменной типа **T** значением типа **V**.

```
V v;
// ...
T t(v); // Инициализируем переменную t значением v
```

Таким образом, возврат значения — это разновидность инициализации.

Функция, объявившая о возврате значения, обязана его возвращать. В частности, с этим связана ошибка "провала к концу функции":

```
double my_abs(int x) // Предупреждение: код содержит ошибки
{
    if (x < 0)
        return -x;
    else if (x > 0)
        return x;
} // Ошибка: если x == 0, функция ничего не возвращает
```

В действительности недостаточно умный компилятор может и не заметить, что вы "забыли" о варианте **x==0**[1]. Если функция сложна, компилятор может не разобраться, возвращает ли она значение, так что следует быть осторожным. Это значит, что программист сам должен убедиться, что функция содержит инструкцию **return** или вызов функции **error()** для любого возможного варианта выхода из функции.

---

[1] Хотя, если вы перепишете второе условие как **x>=0**, ряд компиляторов, которые справедливо предупредят вас о возможном отсутствии возврата значения из функции, при таком исправлении окажутся недостаточно умными и все равно предупредят о том, что функция не возвращает значения. — *Примеч. ред.*

По историческим причинам функция **main()** представляет собой исключение из правила. "Провал до дна" функции **main()** эквивалентен инструкции **return 0**, означающей успешное завершение программы.

В функции, не возвращающей никаких значений, инструкцию **return** без указания возвращаемого значения можно использовать для немедленного выхода из функции, например:

```
void print_until_s(vector<string> v, string quit)
{
    for(int s : v) {
        if (v[i] == quit) return;
        cout << v[i] << '\n';
    }
}
```

Как видим, "падение до дна функции" вполне приемлемо в **void**-функциях. Это эквивалентно наличию "на дне" инструкции **return;**.

8.5.3. Передача параметров по значению

Простейший способ передать аргумент функции заключается в передаче копии его значения, использованного в качестве аргумента. Аргумент функции **f()** является локальной переменной, которая инициализируется при каждом ее вызове. Рассмотрим пример.

```
// Передача по значению (функция получает копию
// передаваемого ей значения)
int f(int x)
{
    x = x+1; // Присваивание значения локальной переменной x
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n'; // Вывод: 1
    cout << xx << '\n';    // Вывод: 0; f() не изменяет xx

    int yy = 7;
    cout << f(yy) << '\n'; // Вывод: 8
    cout << yy << '\n';    // Вывод: 7; f() не изменяет yy
}
```

Поскольку в функцию передается копия значения, инструкция **x=x+1** в функции **f()** не изменяет значений переменных **xx** и **yy**, передаваемых ей при двух вызовах. Передачу аргумента по значению можно проиллюстрировать следующим образом.

Передача по значению представляет собой довольно простой механизм, а ее стоимость равна стоимости копирования значения.

8.5.4. Передача параметров по константной ссылке

Передача по значению проста, понятна и эффективна, если передаются небольшие значения, например переменные типа `int`, `double` или `Token` (см. раздел 6.3.2). А что если передаваемое значение велико и представляет собой изображение (зачастую занимающее несколько миллионов битов), большую таблицу чисел (например, несколько тысяч целых чисел) или длинную строку (например, сотни символов)? Тогда копирование оказывается очень затратным механизмом. Конечно, не стоит слишком сильно беспокоиться о стоимости выполняемых операций, но и выполнять ненужную работу также не следует, так как это свидетельствует о плохом выражении идеи, которую мы хотим реализовать. Например, можно написать следующую функцию, выводящую на экран вектор чисел с плавающей точкой:

```
void print(vector<double> v) // Передача по значению; приемлемо?
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i!=v.size()-1) cout << ", ";
    }
    cout << " }\n";
}
```

Функцию `print()` можно применять к векторам любых размеров:

```
void f(int x)
{
     vector<double> vd1(10);       // Небольшой вектор
     vector<double> vd2(1000000); // Большой вектор
     vector<double> vd3(x);        // Вектор неизвестного размера
     // ... заполняем векторы vd1, vd2, vd3 значениями ...
     print(vd1);
     print(vd2);
     print(vd3);
}
```

Этот код работает, но при первом вызове функции **print()** будет скопирован десяток чисел типа **double** (вероятно, 80 байт), при втором — миллионы чисел типа **double** (вероятно, 8 Мбайт), а при третьем количество копируемых чисел заранее неизвестно. Возникает вопрос "Зачем вообще что-то копировать?" Мы же хотим распечатать вектор, а не копировать его элементы. Очевидно, нам нужен способ передачи переменных функциям без их копирования. Например, если вы получили задание составить список книг, находящихся в библиотеке, то совершенно не обязательно покупать копию здания библиотеки и всех книг, в ней содержащихся — достаточно взять адрес библиотеки, пойти туда и просмотреть все книги на месте.

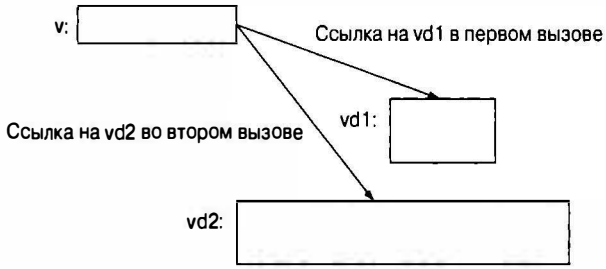
Так и здесь — нам необходим способ передачи функции **print()** "адреса" вектора, а не его копии. "Адрес" вектора называется *ссылкой* (reference) и используется следующим образом:

```
void print(const vector<double>& v) // Передача по
{                                   // константной ссылке
    cout << "{ ";
    for(int i = 0; i < v.size(); ++i) {
       cout << v[i];
       if (i!=v.size()-1) cout << ", ";
    }
    cout << " }\n";
}
```

Символ **&** означает ссылку, а ключевое слово **const** предотвращает случайную модификацию аргумента в функции **print()**. Кроме объявления аргумента, все остальное остается без изменений. Правда, теперь все операции будут производиться не над копией, а над самим аргументом, полученным по ссылке. Такие аргументы называются ссылками, потому что они "ссылаются" на объекты, определенные вне функции. Вызов функции **print()** остается точно таким же, как и раньше.

```
void f(int x)
{
     vector<double> vd1(10);       // Небольшой вектор
     vector<double> vd2(1000000); // Большой вектор
     vector<double> vd3(x);        // Вектор неизвестного размера
     // ... заполняем векторы vd1, vd2, vd3 значениями ...
     print(vd1);
     print(vd2);
     print(vd3);
}
```

Этот механизм можно проиллюстрировать графически.



Константная ссылка обладает полезным свойством: она не позволяет случайно изменить передаваемый объект. Например, если мы сделаем глупую ошибку и попытаемся присвоить элементу вектора, полученному функцией `print()`, какое-то значение, компилятор сразу выдаст сообщение об этом.

```
void print(const vector<double>& v) // Передача по
{                                   // константной ссылке
    // ...
    v[i] = 7; // Ошибка: v — константа (изменяться не может)
    // ...
}
```

Передача аргументов по константной ссылке — очень полезный и распространенный механизм. Вернемся к функции `my_find()` (см. раздел 8.5.1), выполняющей поиск строки в векторе строк. Передача по значению здесь была бы слишком неэффективной.

```
int my_find(vector<string> vs);    // Передача по значению:
            string s);             // копия
```

Если вектор содержит тысячи строк, поиск может потребовать заметного времени даже на быстром компьютере. Мы можем улучшить функцию `my_find()`, передавая аргументы по константной ссылке.

```
// Передача по ссылке: без копирования, доступ только для чтения
int my_find(const vector<string>& vs, const string& s);
```

8.5.5. Передача параметров по ссылке

А что делать, если мы хотим, чтобы функция модифицировала свои аргументы? Иногда это очень нужно. Например, мы можем написать функцию `init()`, которая должна присваивать начальные значения элементам вектора.

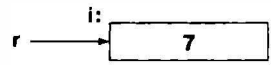
```
void init(vector<double>& v) // Передача по ссылке
{
    for (int i = 0; i < v.size(); ++i) v[i] = i;
}
```

```
void g(int x)
{
    vector<double> vd1(10);       // Небольшой вектор
    vector<double> vd2(1000000); // Большой вектор
    vector<double> vd3(x);        // Вектор неопределенного размера
    init(vd1);
    init(vd2);
    init(vd3);
}
```

Здесь мы хотим, чтобы функция `init()` изменяла вектор, являющийся ее аргументом. Иначе говоря, мы хотим не копировать его (т.е. передавать по значению), не объявлять с помощью константной ссылки (т.е. передавать по константной ссылке), а просто передать обычную ссылку на вектор.

Рассмотрим ссылки более подробно. Ссылка — это конструкция, позволяющая пользователю объявлять новое имя объекта. Например, `int&` — это ссылка на переменную типа `int`. Это позволяет нам написать следующий код:

```
int i = 7;

int& r = i; // r — ссылка на переменную i
r = 9;      // Переменная i становится равной 9
i = 10;
cout << r << ' ' << i << '\n'; // Вывод: 10 10
```



Иначе говоря, любая операция над переменной `r` на самом деле означает операцию над переменной `i`.

Ссылки позволяют уменьшить размер выражений. Рассмотрим следующий пример:

```
vector<vector<double> > v;  // Вектор векторов чисел double
```

Допустим, нам необходимо сослаться на некоторый элемент `v[f(x)][g(y)]` несколько раз. Очевидно, что выражение `v[f(x)][g(y)]` выглядит слишком громоздко, и повторять его несколько раз неудобно. Если бы оно было просто значением, то мы могли бы написать следующий код:

```
double val = v[f(x)][g(y)]; // Значение элемента v[f(x)][g(y)]
```

В таком случае можно было бы повторно использовать переменную `val`. А что если нам нужно и читать элемент `v[f(x)][g(y)]`, и присваивать ему значения `v[f(x)][g(y)]`? В этом случае может пригодиться ссылка.

```
double& var = v[f(x)][g(y)]; // Ссылка на элемент v[f(x)][g(y)]
```

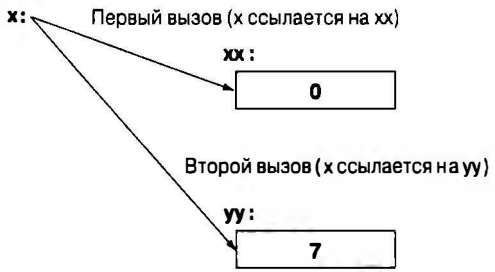
Теперь можем как считывать, так и изменять элемент `v[f(x)][g(y)]` с помощью ссылки `var`:

```
var = var/2+sqrt(var);
```

Это ключевое свойство ссылок — ссылка может служить "аббревиатурой" объекта, и это делает их полезными для использования в качестве аргументов. Рассмотрим пример.

```
// Передача аргумента в функцию по ссылке
int f(int& x)
{
    x = x+1;
    return x;
}

int main()
{
    int xx = 0;
    cout << f(xx) << '\n'; // Вывод: 1
    cout << xx << '\n';    // Вывод: 1; функция f() изменяет
                           // значение xx

    int yy = 7;
    cout << f(yy) << '\n'; // Вывод: 8
    cout << yy << '\n';    // Вывод: 8; функция f() изменяет
                           // значение yy
}
```

Передачу аргументов по ссылке можно проиллюстрировать следующим образом.



Сравните этот пример с соответствующим примером из раздела 8.5.3.

Совершенно очевидно, что передача по ссылке — очень мощный механизм: функции могут непосредственно оперировать любым объектом, передаваемым по ссылке. Например, во многих алгоритмах сортировки перестановка двух значений — весьма важная операция. Используя ссылки, можно написать функцию, меняющую местами два числа типа **double**:

```
void swap(double& d1, double& d2)
{
    double temp = d1; // Копируем значение d1 в переменную temp
    d1 = d2;          // Копируем значение d2 в переменную d1
    d2 = temp;        // Копируем старое значение d1
}                     // в переменную d2
```

```
int main()
{
    double x = 1;
    double y = 2;
    cout << "x == " << x << " y== "
         << y << '\n';              // Вывод: x==1 y==2
    swap(x,y);
    cout << "x == " << x << " y== "
         << y << '\n';              // Вывод: x==2 y==1
}
```

Стандартная библиотека предоставляет функцию **swap()** для любого типа, который можно копировать, поэтому вам не требуется писать функцию **swap()** самостоятельно для каждого типа.

8.5.6. Сравнение механизмов передачи параметров по значению и по ссылке

Когда следует использовать передачу параметров по значению, по ссылке и по константной ссылке? Для начала рассмотрим один технический пример.

```
void f(int a, int& r, const int& cr)
{
    ++a;   // Изменяем локальную переменную a
    ++r;   // Изменяем объект, на который ссылается r
    ++cr;  // Ошибка: cr – константная ссылка
}
```

Если вы хотите изменить значение передаваемого объекта, то должны использовать неконстантную ссылку: передача по значению создаст копию, а передача по константной ссылке предотвратит изменение передаваемого объекта. Так что можно попробовать следующий код:

```
void g(int a, int& r, const int& cr)
{
   ++a;         // Изменяем локальную переменную a
   ++r;         // Изменяем объект, с которым связана ссылка r
   int x = cr; // Считываем объект, с которым связана ссылка cr
}

int main()
{
   int x = 0;
   int y = 0;
   int z = 0;
   g(x,y,z);  // x==0; y==1; z==0
   g(1,2,3);  // Ошибка: аргумент r должен быть переменной OK: поскольку
   g(1,y,3);  // ссылка cr является константной, можно передавать литерал
}
```

Таким образом, если вы хотите изменять значение объекта, передаваемого по ссылке, следует передавать объект. С формальной точки зрения целочисленный литерал `2` — это значение (а точнее, rvalue, т.е. значение в правой части оператора присваивания), а не объект, хранящий значение. Для аргумента `r` функции `f()` требуется lvalue (т.е. значение, стоящее в левой части оператора присваивания).

Обратите внимание на то, что для константной ссылки lvalue не требуется. Она может выполнять преобразования точно так же, как инициализация или передача по значению. При последнем вызове `g(1,y,3)` компилятор зарезервирует переменную типа `int` для аргумента `cr` функции `g()`, так что этот код можно рассматривать следующим образом:

```
g(1,y,3); // Означает: int __compiler_generated = 3;
          // g(1,y,__compiler_generated)
```

Такой объект, создаваемый компилятором, называется *временным объектом* (temporary object).

Итак, ответ на вопрос из первого абзаца раздела можно сформулировать следующим образом.

1. Для передачи очень маленьких объектов следует использовать передачу аргументов по значению.
2. Для передачи больших объектов, которые нельзя изменять, следует использовать передачу аргументов по константной ссылке.
3. Следует возвращать результат, а не модифицировать объект, передаваемый по ссылке.
4. Передачу по ссылке следует использовать только в самых необходимых случаях.

Эти правила позволяют создавать простой, устойчивый к ошибкам и очень эффективный код. Под очень маленькими объектами подразумеваются одна или две переменные типа `int`, одна или две переменные типа `double` или соразмерные им объекты. Если вы видите аргумент, передаваемый по обычной ссылке, то должны считать, что вызываемая функция его модифицирует.

Третье правило отражает ситуацию, когда нам нужна функция, изменяющая значение переменной. Рассмотрим пример.

```
int incr1(int a) { return a+1; }  // Возвращает новое значение
void incr2(int& a) { ++a; }       // Модифицирует объект,
                                  // передаваемый по ссылке

int x = 7;
x = incr1(x);                     // Совершенно очевидно
incr2(x);                         // Совершенно неочевидно
```

Почему же мы все-таки используем передачу аргументов по ссылке? Иногда это оказывается важным.

- При работе с контейнерами (например, векторами) и другими крупными объектами.
- Для функций, изменяющих сразу несколько объектов (в языке C++ функция может возвращать с помощью оператора `return` только одно значение).

Рассмотрим пример.

```
void larger(vector<int>& v1, vector<int>& v2)
    // Делает каждый элемент вектора v1 большим
    // из соответствующих элементов в v1 и v2;
    // аналогично каждый элемент v2 становится
    // меньшим из пары значений
{
    if (v1.size()!= v2.size())
        error("larger(): разные размеры");
    for (int i=0; i < v1.size(); ++i)
        if (v1[i] < v2[i])
            swap(v1[i],v2[i]);
     }

void f()
{
     vector<int> vx;
     vector<int> vy;
     // Считываем vx и vy из входного потока
     larger(vx,vy);
     // ...
}
```

Передача аргументов по ссылке — единственный разумный выбор для функции `larger()`.

Обычно следует избегать функций, модифицирующих несколько объектов одновременно. Теоретически всегда есть альтернативы, например возвращение объекта класса, хранящего несколько значений. Однако все же существует множество готовых программ, в которых функции модифицируют один или несколько аргументов, и этот факт следует учитывать. Например, в языке Fortran — основном языке программирования, используемом для математических вычислений на протяжении более пятидесяти лет, — все аргументы традиционно передаются по ссылке. Многие программисты, работающие над вычислительными программами, копируют проекты на языке Fortran, и вызывают функции, написанные на нем. Такой код часто использует передачу по ссылке или по константной ссылке.

Если передача по ссылке используется только для того, чтобы избежать копирования, следует использовать константную ссылку. Следовательно, если мы видим аргумент, передаваемый по обычной ссылке, это означает, что функция его изменяет. Иначе говоря, если мы видим аргумент, передаваемый по ссылке, не являющейся константной, мы должны считать, что данная функция не только может его модифицировать, но именно так и поступает. В такой ситуации мы обязаны особенно тщательно проверить, действительно ли эта функция делает то, для чего предназначена.

8.5.7. Проверка аргументов и преобразование типов

Передача аргумента представляет собой инициализацию формального аргумента функции фактическим аргументом, указанным при ее вызове. Рассмотрим пример.

```
void f(T x);
f(y);
T x=y; // Инициализация переменной x значением y
       // (см. раздел 8.2.2)
```

Вызов `f(y)` является корректным, если корректна инициализация `T x=y;` и когда обе переменные `x` получают одно и то же значение. Рассмотрим пример.

```
void f(double x);

void g(int y)
{
     f(y);
     double x = y; // Инициализация x значением y
                   // (см. раздел 8.2.2)
}
```

Обратите внимание, что для инициализации переменной `x` значением `y` необходимо преобразовать значение типа `int` в значение типа `double`. То же самое происходит и при вызове функции `f()`. Значение типа `double`, полученное функцией `f()`, совпадает со значением, хранящимся в переменной `x`.

Преобразования часто оказываются полезными, но иногда дают удивительные результаты (раздел 3.9.2). Следовательно, работая с преобразованиями, следует проявлять осторожность. Передачу переменной типа `double` в качестве аргумента функции, ожидающей переменную типа `int`, редко можно оправдать.

```
void ff(int x);

void gg(double y)
```

```
{
    ff(y);      // Как понять, имеет ли это смысл?
    int x = y;  // Как понять, имеет ли это смысл?
}
```

Если вы действительно хотите усечь значение типа **double** до значения типа **int**, то сделайте это явно.

```
void ggg(double x)
{
    int x1 = x;                     // Усечение x
    int x2 = int(x);
    int x3 = static_cast<int>(x); // Явное преобразование (§17.8)

    ff(x1);
    ff(x2);
    ff(x3);

    ff(x);                          // Усечение x
    ff(int(x));
    ff(static_cast<int>(x));        // Явное преобразование (§17.8)
}
```

При этом просматривающий этот код программист сможет увидеть, что вы в действительно думали об этой проблеме.

8.5.8. Реализация вызова функции

Как же на самом деле компилятор выполняет вызов функции? Функции **expression()**, **term()** и **primary()**, описанные в главах 6 и 7, прекрасно подходят для иллюстрации этой концепции за исключением одной детали: они не принимают никаких аргументов, поэтому на их примере невозможно объяснить механизм передачи параметров. Но погодите! Они *обязаны* принимать некую входную информацию; если бы это было не так, то они не могли бы делать ничего полезного. Они принимают неявный аргумент, используя объект **ts** класса **Token_stream** для получения входной информации; объект **ts** является глобальной переменной. Это несколько снижает прозрачность работы программы. Мы можем улучшить эти функции, позволяя им принимать аргумент типа **Token_stream&**. Благодаря этому нам не придется переделывать ни один вызов функции.

Первая функция, **expression()**, совершенно очевидна; она имеет один аргумент (**ts**) и две локальные переменные (**left** и **t**).

```
double expression(Token_stream& ts)
{
    double left = term(ts);
    Token t = ts.get();
    // ...
}
```

Вторая функция **term()** очень похожа на функцию **expression()**, за исключением того, что имеет дополнительную локальную переменную (d), которая используется для хранения результата деления (раздел **case '/'**).

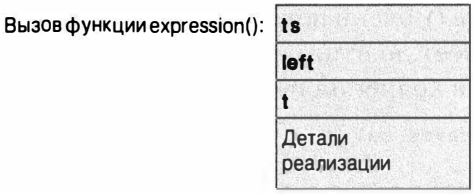
```
double term(Token_stream& ts)
{
    double left = primary(ts);
    Token t = ts.get();
    // ...
        case '/':
        {
            double d = primary(ts);
            // ...
        }
    // ...
}
```

Третья функция **primary()** очень похожа на функцию **term()**, с тем отличием, что у нее нет локальной переменной **left**.

```
double primary(Token_stream& ts)
{
    Token t = ts.get ();
    switch (t.kind) {
    case '(':
        {
            double d = expression(ts);
            // ...
        }
        // ...
    }
}
```

Теперь у этих функций нет скрытых глобальных переменных, и они превосходно подходят для иллюстрации: у них есть аргумент и локальные переменные, и они вызывают друг друга. Возможно, вы захотите освежить память и еще раз посмотреть, как выглядят эти функции в законченном виде, но все их основные свойства, относящиеся к механизму вызова функций, уже перечислены.

При вызове функции реализация языка программирования создает структуру данных, содержащую копии всех ее параметров и локальных переменных. Например, при первом вызове функции **expression()** компилятор создает структуру, напоминающую показанную на рисунке.



"Детали реализации" меняются от компилятора к компилятору, но в принципе к ним относится информация о том, что именно функция должна вернуть вызвавшей ее функции и как вернуть этой вызвавшей функции управление. Такую структуру данных называют *записью активации функции* (function activation record) или просто *записью активации*. Каждая функция имеет собственную запись активации. Обратите внимание на то, что с точки зрения реализации параметр представляет собой всего лишь еще одну локальную переменную.

Теперь функция `expression()` вызывает функцию `term()`, так что компилятор создает запись активации для вызова функции `term()`.

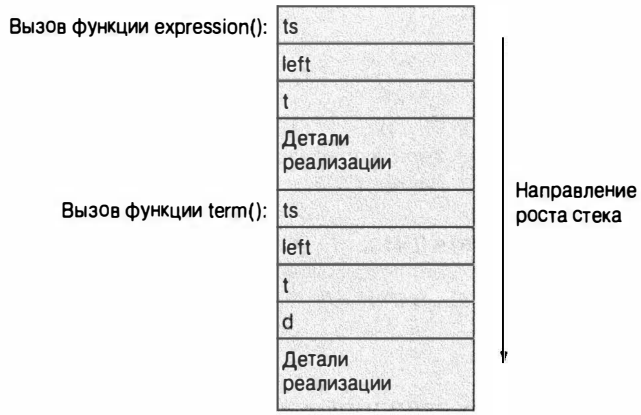
Обратите внимание на то, что функция `term()` имеет дополнительную переменную `d`, которую необходимо хранить в памяти, поэтому при вызове мы резервируем для нее место, даже если в коде она нигде не используется. Все в порядке. Для разумных функций (а только такие функции мы явно или неявно используем в нашей книге) затраты на создание записей активации не зависят от их размера. Локальная переменная `d` будет инициализирована только в том случае, если будет выполнен раздел `case '/'`. Теперь функция `term()` вызывает функцию `primary()`, и мы получаем следующую картину.

Все это становится довольно скучным, но теперь функция `primary()` вызывает функцию `expression()`.

Этот вызов функции `expression()` также имеет собственную запись активации, отличающуюся от записи активации первого вызова функции `expression()`. Хорошо это или плохо, но теперь мы попадаем в очень запутанную ситуацию, поскольку переменные `left` и `t` при двух разных вызовах будут разными. Функция, которая прямо или (как в данном случае) косвенно вызывает себя, называется *рекурсивной* (recursive). Как можно видеть, рекурсивные функции являются естественным следствием метода реализации, который мы используем для вызовов функций и возврата из них (и наоборот).

Итак, каждый раз, когда мы вызываем функцию, *стек записей активации* (stack of activation records), который часто называют просто *стеком* (stack), увеличивается на одну запись. И наоборот, когда функция возвращает управление, ее запись активации больше не используется. Например, когда при последнем вызове функции `expression()` управление возвращается функции `primary()`, стек возвращается в предыдущее состояние.

Когда функция `primary()` возвращает управление функции `term()`, стек возвращается в состояние, показанное ниже.



И так далее. Этот стек, который часто называют *стеком вызовов* (call stack), — структура данных, которая увеличивается и уменьшается с одного конца в соответствии с правилом "последним вошел — первым вышел".

Запомните, что детали реализации стека зависят от реализации языка C++, но в принципе соответствуют схеме, описанной выше. Надо ли вам знать, как реализованы вызовы функции? Разумеется, нет; мы и раньше прекрасно обходились без этого знания, но многие программисты любят использовать термины "запись активации" и "стек вызовов", поэтому лучше понимать, о чем они говорят.

8.5.9. constexpr-функции

Функция представляет собой вычисление, и иногда хотелось бы выполнить его еще во время компиляции. Причиной такого желания обычно является то, что это приводит к огромной экономии этих же вычислений во время выполнения программы. Мы используем функции, чтобы сделать наши расчеты понятными, поэтому, естественно, мы иногда хотим использовать функцию в константном выражении. Мы выражаем наше намерение заставить вычислить значение функции компилятором, объявляя ее как **constexpr**. Такая функция может вычисляться компилятором, если в качестве аргументов функции используются константные выражения, например:

```
constexpr double xscale = 10;    // Масштабные множители
constexpr double yscale = 0.8;
constexpr Point scale(Point p){return{xscale*p.x,yscale*p.y};};
```

Предположим, что **Point** представляет собой простую структуру с членами **x** и **y**, представляющими двумерные координаты. Теперь, когда мы передаем функции **scale()** аргумент **Point**, она возвращает точку с координатами, масштабированными с использованием множителей **xscale** и **yscale**, например:

```
void user(Point p1)
{
    Point p2 {10,10};

    Point p3 = scale(p1); // OK: p3 == {100,8};
                          // вычисление времени выполнения
    Point p4 = scale(p2); // p4 == {100,8}

    constexpr Point p5 = scale(p1); // Ошибка: scale(p1) не
                          // является константным выражением
    constexpr Point p6 = scale(p2); // p6 == {100,8}
    // ...
}
```

Объявленная как **constexpr** функция ведет себя так же, как обычная функция — до тех пор, пока вы не используете ее там, где необходима константа. Тогда она вычисляется во время компиляции при условии, что ее аргументами являются константные выражения (например, **p2**), и приводит к ошибке, если они не являются таковыми (например, **p1**). Чтобы это было возможно, функция, объявленная как **constexpr**, должна быть настолько простой, чтобы компилятор (любой компилятор, соответствующий стандарту) мог ее вычислить. В C++11 это означает, что функция, объявленная как **constexpr**, должна иметь тело, состоящее из одной инструкции **return** (примером может служить функция **scale()**); в C++14 мы также можем написать простой цикл. **constexpr**-функция не может иметь побочные эффекты, т.е. она не может изменять значения переменных вне собственного тела, за исключением тех, которые она присваивает или использует для инициализации.

Вот для простоты пример функции, которая нарушает эти правила:

```
int glob = 9;

constexpr void bad(int& arg) // Ошибка: нет возвращаемого
{                            // значения
    ++arg;     // Ошибка: модифицирует значение переменной вне
               // тела с помощью аргумента
    glob = 7; // Ошибка: модифицирует нелокальную переменную
}
```

Если компилятор не в состоянии определить, что **constexpr**-функция является "достаточно простой" (в соответствии с подробными правилами стандарта), такая функция рассматривается как ошибка.

8.6. Порядок вычислений

Программа выполняется инструкция за инструкцией в соответствии с правилами языка. Когда этот "поток выполнения" достигает определения

переменной, происходит ее создание, т.е. в памяти выделяется память для объекта, и этот объект инициализируется. Когда переменная выходит из области видимости, она уничтожается, т.е. объект, на который она ссылалась, удаляется из памяти, и компилятор может использовать ранее занимаемый им участок памяти для других целей. Рассмотрим пример.

```
string program_name = "silly";
vector<string> v;          // v – глобальная переменная
void f()
{
    string s;              // s – локальная переменная функции f
    while (cin>>s && s!="quit")
    {
        string stripped; // stripped – локальна в цикле
        string not_letters;
        for (int i=0; i < s.size(); ++i) // i имеет область
                                         // видимости инструкции
            if (isalpha(s[i]))
                stripped += s[i];
            else
                not_letters += s[i];
        v.push_back(stripped);
        // ...
    }
    // ...
}
```

Глобальные переменные, такие как **program_name** и **v**, инициализируются до выполнения первой инструкции функции **main()**. Они существуют, пока программа не закончит работу, а потом уничтожаются. Они создаются в порядке следования своих определений (т.е. переменная **program_name** создается до переменной **v**), а уничтожаются в обратном порядке (т.е. переменная **v** уничтожается до переменной **program_name**).

Когда какая-нибудь функция вызывает функцию **f()**, сначала создается переменная **s**; иначе говоря, переменная **s** инициализируется пустой строкой. Она будет существовать, пока функция **f()** не вернет управление вызывающей функции. Каждый раз, когда мы входим в тело цикла **while**, создаются переменные **stripped** и **not_letters**. Поскольку переменная **stripped** определена до переменной **not_letters**, сначала создается переменная **stripped**. Они существуют до выхода из тела цикла. В этот момент они уничтожаются в обратном порядке (иначе говоря, переменная **not_letters** уничтожается до переменной **stripped**) и до того, как произойдет проверка условия выхода из цикла. Так что если до того, как мы обнаружим строку **quit**, мы выполним цикл десять раз, переменные **stripped** и **not_letters** будут созданы и уничтожены десять раз.

Каждый раз, когда мы входим в цикл `for`, создается переменная `i`. Каждый раз, когда мы выходим из цикла `for`, переменная `i` уничтожается до того, как мы достигнем инструкции `v.push_back(stripped);`.

Обратите внимание на то, что компиляторы (и редакторы связей) довольно разумны и им позволено оптимизировать код (что они успешно и делают), лишь бы эта оптимизация приводила к тем же результатам, что и изначальный код. В частности, компиляторы не выделяют и не освобождают память чаще, чем это действительно требуется.

8.6.1. Вычисление выражения

Порядок вычисления подвыражений управляется правилами, которые больше ориентируются на оптимизацию кода, чем на удобство работы программиста. Это неудобно, но в любом случае следует избегать чрезмерно сложных выражений. Помните простое правило: если вы изменяете значение переменной в выражении, то не используйте ее в одном и том же выражении дважды. Рассмотрим примеры.

```
v[i] = ++i;           // Неопределенный порядок вычислений
v[++i] = i;           // Неопределенный порядок вычислений
int x = ++i + ++i;    // Неопределенный порядок вычислений
cout << ++i << ' '
     << i << '\n';    // Неопределенный порядок вычислений
f(++i,++i);           // Неопределенный порядок вычислений
```

К сожалению, не все компиляторы выдают предупреждение о таких ошибках; это плохо, потому что нельзя рассчитывать на то, что результаты будут такими же при выполнении вычислений на другом компьютере, при использовании других компиляторов или при других установках оптимизатора. Компиляторы действительно по-разному обрабатывают этот код; избегайте таких ситуаций.

Обратите внимание на то, что оператор = (присваивание) в выражениях рассматривается просто как еще один оператор, наряду с остальными, поэтому нет никакой гарантии того, что левая часть этого оператора будет вычислена раньше правой части. По этой причине выражение `v[++i] = i` имеет неопределенный результат.

8.6.2. Глобальная инициализация

Глобальные переменные (и переменные из пространств имен; раздел 8.7) в отдельной единице трансляции инициализируются в том порядке, в котором они появляются. Рассмотрим пример.

```
// Файл f1.cpp
int x1 = 1;
int y1 = x1+2; // Переменная y1 становится равной 3
```

Эта инициализация логически происходит до выполнения кода в функции **main()**.

Использование глобальных переменных, за исключением редчайших ситуаций, нецелесообразно. Мы уже говорили, что не существует эффективного способа, позволяющего программисту определить, какие части программы считывают или записывают переменную (раздел 8.4). Другая проблема заключается в том, что порядок инициализации глобальных переменных не определен. Рассмотрим пример.

```
// Файл f2.cpp
extern int y1;
int y2 = y1+2; // Переменная y2 становится равной 2 или 5
```

Такой код нежелателен по нескольким причинам: в нем используются глобальные переменные, которые имеют слишком короткие имена, и сложная инициализация глобальных переменных. Если глобальные переменные в файле **f1.cpp** инициализируются до глобальных переменных в файле **f2.cpp**, то переменная **y2** будет инициализирована числом **5** (как наивно ожидает программист). Однако, если глобальные переменные в файле **f2.cpp** инициализируются до глобальных переменных в файле **f1.cpp**, переменная **y2** будет инициализирована числом **2** (поскольку память, используемая для глобальных переменных, инициализируется нулем до попытки сложной инициализации). Избегайте этого и старайтесь не использовать нетривиальную инициализацию глобальных переменных; любой инициализатор, отличающийся от константного выражения, следует считать сложным.

Но что же делать, если нам действительно нужна глобальная переменная (или константа) со сложной инициализацией? Правдоподобным примером может служить значение по умолчанию для типа **Date**, предназначенного для библиотеки, поддерживающей бизнес-транзакции:

```
const Date default_date(1970,1,1); // По умолчанию — 01.01.1970
```

Откуда мы можем знать, что переменная **default_date** не использовалась до своей инициализации? В принципе мы и не можем этого знать, поэтому не должны писать такое определение. Чаще всего в подобных ситуациях используется вызов функции, возвращающей некое значение. Рассмотрим пример.

```
const Date default_date() // Возвращает дату по умолчанию
{
    return Date(1970,1,1);
}
```

Эта функция создает объект типа **Date** каждый раз, когда вызывается функция **default_date()**. Часто этого вполне достаточно, но если функция **default_date()** вызывается часто, а создание

объекта класса **Date** связано с большими затратами, предпочтительнее было бы сконструировать его только один раз. В таком случае код мог бы выглядеть так:

```
const Date& default_date()
{
    static const Date dd(1970,1,1); // Инициализируем dd
                                    // только при первом вызове
    return dd;
}
```

Статическая локальная переменная инициализируется (создается) только при первом вызове функции, в которой она объявлена. Обратите внимание на то, что мы возвращаем ссылку, чтобы исключить ненужное копирование, а кроме того, эта ссылка — константная, чтобы не дать возможности вызывающей функции случайно изменить возвращаемое значение. Рассуждения о способах передачи аргументов (раздел 8.5.6) относятся и к возвращаемому значению.

8.7. Пространства имен

Для организации кода в рамках функции используются блоки (раздел 8.4). Для организации функций, данных и типов в один тип используется класс (глава 9). Предназначение функций и классов заключается в следующем.

- Они позволяют определить множество сущностей без опасения, что их имена совпадут с другими именами в программе.
- Позволяют именовать то, что мы определили.

Теперь нам не хватает средства для организации классов, функций, данных и типов в виде идентифицируемой и именованной части программы без определения нового типа. Языковый механизм, позволяющий осуществить такую группировку объявлений, называется *пространством имен* (namespace). Например, мы можем создать графическую библиотеку с классами **Color**, **Shape**, **Line**, **Function** и **Text** (глава 13).

```
namespace Graph_lib {
    struct Color            { /* ... */ };
    struct Shape            { /* ... */ };
    struct Line : Shape     { /* ... */ };
    struct Function : Shape { /* ... */ };
    struct Text : Shape     { /* ... */ };
    // ...
    int gui_main()          { /* ... */ }
}
```

Очень вероятно, что кто-то еще захочет использовать эти имена, но теперь это уже не имеет значения. Вы можете определить что-то с именем **Text**, но это что-то уже невозможно спутать с нашим классом, имеющим то же имя. Наш класс называется **Graph_lib::Text**, а ваш класс — просто **Text**. Проблема возникнет только в том случае, если в вашей программе есть класс или пространство имен **Graph_lib**, в которое входит класс **Text**. Имя **Graph_lib** довольно неудачное; мы выбрали его потому, что "красивое и очевидное" имя **Graphics** имеет больше шансов встретиться где-нибудь еще.

Допустим, ваш класс **Text** является частью библиотеки для обработки текстов. Та же логика, которая заставила нас разместить графические средства в пространстве имен **Graph_lib**, подсказывает, что средства для обработки текстов следует поместить в пространстве имен, скажем, с именем **TextLib**.

```
namespace TextLib {
    class Text  { /* ... */ };
    class Glyph { /* ... */ };
    class Line  { /* ... */ };
    // ...
}
```

Если бы мы оба использовали глобальное пространство имен, то столкнулись бы с реальной проблемой. В этом случае действительно возникла бы коллизия между именами классов **Text** и **Line**. И что еще хуже, если бы мы были не создателями, а пользователями библиотеки, то никак не могли бы изменить эти имена и решить проблему. Использование пространств имен позволяет избежать проблем; иначе говоря, наш класс **Text** — это класс **Graph_lib::Text**, а ваш — **TextLib::Text**. Имя, составленное из имени пространства имен (или имени класса) и имени члена с помощью двух двоеточий, **::**, называют *полностью квалифицированным именем* (fully qualified name).

8.7.1. Объявления using и директивы using

Писать полностью определенные имена довольно утомительно. Например, средства стандартной библиотеки языка C++ определены в пространстве имен **std** и могут использоваться примерно так:

```
#include<string>   // Доступ к библиотеке string
#include<iostream> // Доступ к библиотеке iostream

int main()
{
    std::string name;
    std::cout << "Пожалуйста, введите имя\n";
    std::cin  >> name;
```

```
    std::cout << "Привет, " << name << '\n';
}
```

Тысячи раз обращаясь к элементам стандартной библиотеки **string** и **cout**, не хотелось бы каждый раз указывать их полностью квалифицированные имена — **std::string** и **std::cout**. Напрашивается решение: раз и навсегда указать, что под классом **string** мы имеем в виду класс **std::string**, под потоком **cout** — поток **std::cout** и т.д.

```
using std::string; // string означает std::string
using std::cout;   // cout означает std::cout
// ...
```

Эта конструкция называется объявлением **using**. Она эквивалентна обращению "Бьярне", которое относится к Бьярне Страуструпу, при условии, что никаких других Бьярне в комнате нет.

Иногда мы предпочитаем еще большее "сокращение" для использования имен из пространства: "Если ты не в состоянии найти объявление имени в данной области видимости, поищи его в пространстве имен **std**". Чтобы сказать это компилятору, используется директива **using**.

```
using namespace std;  // Открывает доступ к именам
                      // из пространства имен std
```

Обычно используется именно этот стиль.

```
#include<string>      // Доступ к библиотеке string
#include<iostream>    // Доступ к библиотеке iostream
using namespace std;  // Доступ к именам из пространства std

int main()
{
    string name;
    cout << "Пожалуйста, введите имя\n";
    cin  >> name;
    cout << "Привет, " << name << '\n';
}
```

Здесь поток **cin** — это поток **std::cin**, класс **string** — это класс **std::string** и т.д. Пока мы используем заголовочный файл **std_lib_facilities.h**, нам не приходится беспокоиться о стандартных заголовках и пространстве имен **std**.

Обычно желательно избегать использования директив **using** для любых пространств имен, за исключением тех из них, которые широко известны в конкретной области приложения, такого как пространство имен **std**. Проблема, связанная с чрезмерным использованием директивы **using**, заключается в том, что легко запутаться в том, из какого пространства имен пришло то или иное имя, и мы снова рискуем создать коллизию. Явная квалификация с помощью пространств имен и

объявлений using не страдает этой проблемой, так что размещение директивы using в заголовочном файле (в результате чего пользователь не в состоянии ее избежать) — очень плохая привычка. Да, мы разместили директиву using для пространства имен std в заголовочном файле std_lib_facilities.h, просто чтобы упростить обучение. Это позволяет нам написать следующий код:

```
#include "std_lib_facilities.h"
int main()
{
     string name;
     cout << "Пожалуйста, введите имя\n";
     cin  >> name;
     cout << "Привет, " << name << '\n';
}
```

Мы обещаем больше никогда так не делать для любых пространств имен, кроме std.

Задание

1. Создайте три файла: my.h, my.cpp и use.cpp. Заголовочный файл my.h содержит следующий код:

   ```
   extern int foo;
   void print_foo();
   void print(int);
   ```

   Исходный файл my.cpp содержит директивы #include для файлов my.h и std_lib_facilities.h и определения функций print_foo() (для вывода значения переменной foo в поток cout) и print(int i) (для вывода в поток cout значения переменной i).

   Исходный файл use.cpp содержит директиву #include для включения файла my.h, определение функции main(), которая присваивает переменной foo значение 7 и выводит ее на экран с помощью функции print_foo(), а также выводит значение 99 с помощью функции print(). Обратите внимание на то, что файл use.cpp не содержит директиву #include std_lib_facilities.h, поскольку не использует явно ни одну из ее возможностей.

   Скомпилируйте эти файлы и запустите их. В операционной системе Windows, чтобы увидеть результаты вывода на экран, в проект следует включить функции use.cpp и my.cpp и использовать в файле use.cpp код { char cc; cin >> cc; }. *Указание*: для использования cin следует добавить в файл директиву #include <iostream>.

2. Напишите три функции: swap_v(int,int), swap_r(int&,int&) и swap_cr(const int&, const int&). Каждая из них должна иметь тело

```
{ int temp; temp = a, a=b; b=temp; }
```

где `a` и `b` — имена аргументов.

Попробуйте вызвать каждую из этих функций, как показано ниже.

```
int x = 7;
int y = 9;
swap_?(x,y);     // Замените знак ? буквой v, r или cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

Какие функции и вызовы будут скомпилированы и почему? После каждой скомпилированной функции обмена выведите на экран значения аргументов, чтобы убедиться, что они действительно поменялись местами. Если результат вас удивит, обратитесь к разделу 8.6.

3. Напишите программу, использующую единственный файл, содержащий пространства имен `X`, `Y` и `Z`, так, чтобы функция `main()`, приведенная ниже, работала правильно.

```
int main()
{
    X::var = 7;
    X::print();       // Вывод var из пространства имен X
    using namespace Y;
    var = 9;
    print();          // Вывод var из пространства имен Y
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();      // Вывод var из пространства имен Z
    }
    print();          // Вывод var из пространства имен Y
    X::print();       // Вывод var из пространства имен X
}
```

Каждое пространство имен должно содержать определение переменной `var` и функции `print()`, выводящей соответствующую переменную `var` в поток `cout`.

Контрольные вопросы

1. В чем заключается разница между объявлением и определением?
2. Как синтаксически отличить объявление функции от определения функции?
3. Как синтаксически различить объявление переменной от определения переменной?
4. Почему функции из калькулятора в главе 6 нельзя использовать, не объявив их сначала?
5. Чем является инструкция `int a;` — определением или просто объявлением?
6. Почему желательно инициализировать переменные при их объявлении?
7. Из каких элементов состоит объявление функции?
8. Какую пользу приносят отступы в исходном тексте?
9. Для чего используются заголовочные файлы?
10. Что такое область видимости объявления?
11. Перечислите разновидности областей видимости. Приведите пример каждой из них.
12. В чем заключается разница между областью видимости класса и локальной областью видимости?
13. Почему программист должен минимизировать количество глобальных переменных?
14. В чем заключается разница между передачей аргумента по значению и передачей аргумента по ссылке?
15. В чем заключается разница между передачей аргумента по ссылке и передачей аргумента по константной ссылке?
16. Что делает функция `swap()`?
17. Следует ли определять функцию с параметром типа `vector<double>`, передаваемым по значению?
18. Приведите пример неопределенного порядка выполнения вычислений. Какие проблемы создает неопределенный порядок вычислений?
19. Что означают выражения `x&&y` и `x||y`?
20. Соответствуют ли стандарту языка C++ следующие конструкции: функции внутри функций, функции внутри классов, классы внутри классов, классы внутри функций?
21. Что входит в запись активации?
22. Что такое стек вызовов и зачем он нужен?

23. Для чего нужны пространства имен?
24. Чем пространство имен отличается от класса?
25. Объясните смысл объявления `using`.
26. Почему следует избегать директив `using` в заголовочных файлах?
27. Опишите пространство имен `std`.

Термины

`const`	запись активации	определение функции
`constexpr`	инициализатор	параметр
`extern`	локальная область видимости	передача аргументов
`namespace`	необъявленный идентификатор	передача по значению
`return`	область видимости	передача по константной ссылке
аргумент	область видимости инструкции	передача по ссылке
вложенный блок	область видимости класса	предварительное объявление
возвращаемое значение	область видимости пространства имен	рекурсия
глобальная область видимости	объявление	стек вызовов
директива `using`	объявление `using`	технические детали
заголовочный файл	определение	функция

Упражнения

1. Модифицируйте программу-калькулятор из главы 7, чтобы поток ввода стал явным параметром (как показано в разделе 8.5.8). Кроме того, напишите конструктор класса `Token_stream` (раздел 7.8.2) с параметром `istream&` так, чтобы, когда вы поймете, как создавать собственные потоки ввода и вывода (например, подключать файлы), вы смогли бы использовать калькулятор с ними. *Указание*: не пытайтесь копировать `istream`.
2. Напишите функцию `print()`, которая выводит в поток `cout` вектор целых чисел. Передавайте ей два аргумента: строку для "метки" результатов и объект класса `vector`.
3. Создайте вектор чисел Фибоначчи и выведите их на печать с помощью функции из упр. 2. Для того чтобы создать вектор, напишите функцию `fibonacci(x,y,v,n)`, в которой аргументы `x` и `y` имеют тип `int`, аргумент `v` представляет собой пустой `vector<int>`, а аргумент `n` — количество элементов, подлежащих записи в вектор `v`; элемент `v[0]` будет равен `x`, а `v[1]` — `y`. Число Фибоначчи — это элемент последовательности, в которой каждый элемент является суммой двух

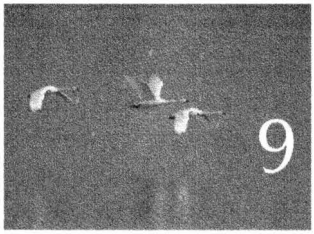
предыдущих. Например, если последовательность начинается с чисел 1 и 2, то мы получаем последовательность 1, 2, 3, 5, 8, 13, 21, ... Функция `fibonacci()` должна генерировать такую последовательность, начинающуюся с чисел `x` и `y`.

4. Переменная типа `int` может хранить целые числа, не превышающие некоторого максимального числа. Вычислите приближение этого максимального числа с помощью функции `fibonacci()`.
5. Напишите две функции, обращающие порядок следования элементов в объекте типа `vector<int>`. Например, вектор 1, 3, 5, 7, 9 становится вектором 9, 7, 5, 3, 1. Первая функция, изменяющая порядок следования элементов на противоположный, должна создавать новый объект класса `vector`, а исходный объект класса `vector` должен оставаться неизменным. Другая функция должна изменять порядок следования элементов в передаваемом ей векторе без использования других векторов. *Указание*: воспользуйтесь функцией `swap`.
6. Напишите варианты функций из упражнения 5 для класса `vector <string>`.
7. Считайте пять имен в вектор `vector<string> name`, затем предложите пользователю указать возраст названных людей и запишите их в вектор `vector<double> age`. Затем выведите на печать пять пар (`name[i],age[i]`). Отсортируйте имена (`sort(name.begin(),name.end())`) и выведите на экран пары (`name[i],age[i]`). Сложность здесь заключается в том, чтобы получить вектор `age`, в котором порядок следования элементов соответствовал бы порядку следования элементов вектора `name`. *Указание*: перед сортировкой вектора `name` создайте его копию и используйте ее для получения упорядоченного вектора `age` после сортировки вектора `name`. Затем выполните упражнение снова, разрешив использование произвольного количества имен.
8. Напишите функцию, которая для двух данных объектов типа `vector <double>`, `price` и `weight`, вычисляет значение, равное сумме всех произведений `price[i]*weight[i]`. Проверьте выполнение условия `weight.size() == price.size()`.
9. Напишите функцию `maxv()`, возвращающую наибольший элемент вектора, передаваемого в качестве аргумента.
10. Напишите функцию, которая находит наименьший и наибольший элементы вектора, являющегося ее аргументом, а также вычисляющую их среднее и медиану. Не используйте глобальные переменные. Результаты можно вернуть либо в виде структуры `struct`, либо с помощью механизма передачи аргументов по ссылке. Какой из этих двух способов следует предпочесть и почему?

11. Усовершенствуйте функцию `print_until_s()` из раздела 8.5.2. Протестируйте ее. Какие наборы данных лучше всего подходят для тестирования? Укажите причины. Затем напишите функцию `print_until_ss()`, которая выводит строки, пока не обнаружит второе вхождение строки аргумента `quit`.
12. Напишите функцию, принимающую аргумент типа `vector<string>` и возвращающую объект типа `vector<int>`, содержащий количество символов в каждой строке. Кроме того, найдите самую короткую и самую длинную строки, а также лексикографически первую и последнюю строки. Сколько отдельных функций вы использовали бы для решения этих задач? Почему?
13. Можно ли объявить константный аргумент функции, который передается не по ссылке (например, `void f(const int);`)? Что это значит? Зачем это нужно? Почему такая конструкция практически не применяется? Испытайте ее; напишите несколько маленьких программ, чтобы увидеть, как она работает.

Послесловие

Большую часть этой (и следующей) главы можно было бы вынести в приложение. Однако в части II, "Ввод и вывод" книги нам потребуются многие средства, описанные здесь. Кроме того, очень скоро мы столкнемся с задачами, для решения которых эти средства были изобретены. При написании простых программ вы неизбежно должны будете решать такие задачи. Поэтому, чтобы сэкономить время и минимизировать недоразумения, необходим систематический подход, а не ряд "случайных" ссылок на справочное руководство и приложения.



Технические детали: классы и прочее

Помните: все перебудем времени.
— Пит Хейн (Piet Hein)

В этой главе мы сосредоточим внимание на основном инструменте программирования: языке C++. Мы опишем технические подробности этого языка, связанные в основном с пользовательскими типами, иначе говоря — с классами и перечислениями. Свойства языка описываются на примере поэтапной разработки типа `Date`. К тому же это позволяет продемонстрировать некоторые полезные приемы разработки классов.

9.1. Типы, определенные пользователем

В языке C++ есть встроенные типы, такие как **char**, **int** и **double** (подробнее они описаны в разделе A.8). Тип называется встроенным, если компилятор знает, как представить объекты такого типа и какие операторы к нему можно применять (такие, как **+** и *****) без уточнений в виде объявлений, которые создает программист в исходном коде.

Типы, не относящиеся к встроенным, называют *пользовательскими типами* или *типами, определенными пользователем* (user-defined types — UDT). Они могут быть частью стандартной библиотеки, доступной в любой реализации языка C++, соответствующей стандарту ISO (например, классы **string**, **vector** и **ostream**, описанные в главе 10), или типами, самостоятельно созданными программистом, как классы **Token** и **Token_stream** (см. разделы 6.5 и 6.6). Как только мы освоим необходимые технические детали, мы создадим графические типы, такие как **Shape**, **Line** и **Text** (речь о них пойдет в главе 13). Стандартные библиотечные типы являются такой же частью языка, как и встроенные типы, но мы все же рассматриваем их как определенные пользователем, поскольку они построены из таких же элементарных конструкций и с помощью тех же приемов, что и типы, разработанные нами; разработчики стандартных библиотек не имеют особых привилегий и средств, которых нет у нас. Как и встроенные типы, большинство типов, определенных пользователем, предоставляют возможность выполнения над ними различных операций. Например, класс **vector** имеет операции **[]** и **size()** (разделы 4.6.1 и B.4.8), класс **ostream** — операцию **<<**, класс **Token_stream** — операцию **get()** (раздел 6.8), а класс **Shape** — операции **add(Point)** и **set_color()** (раздел 14.2).

Зачем мы создаем типы? Компилятор не знает всех типов, которые мы хотим использовать в наших программах. Это невозможно в принципе, поскольку существует слишком много полезных ти-

пов — ни один разработчик языка программирования или компиляторов не может знать обо всех. Каждый день мы изобретаем новые типы. Почему? Для чего пригодны эти типы? Типы нужны, в первую очередь, для непосредственного представления идей в коде. Когда мы пишем программу, ее идеалом является представление наших идей непосредственно в коде так, чтобы мы сами, наши коллеги и компилятор могли понять, что мы написали. Когда мы хотим выполнять арифметические операции над целыми числами, нам отлично подойдет тип `int`; когда хотим работать с текстом, хорошим выбором является класс `string`; когда мы работали с входной информацией для калькулятора, огромную помощь нам оказали классы `Token` и `Token_stream`. Эта помощь имеет два аспекта.

- *Представление.* Тип знает, как представить в объекте необходимые данные.
- *Операции.* Тип знает, какие операции можно применять к объектам.

Многие идеи следуют данному шаблону: "нечто" имеет данные для представления своего текущего значения (которое иногда называют текущим *состоянием*) и набор операций, которые к ним можно применить. Подумайте о компьютерном файле, веб-странице, CD-плеере, чашке кофе, телефоне, телефонном справочнике; все они характеризуются определенными данными и имеют более или менее фиксированный набор операций, которые можно выполнить над ними. В каждом случае результат операции зависит от данных — "текущего состояния" объекта.

Итак, мы хотим выразить "идею" или "понятие" в коде в виде структуры данных и набора функций. Возникает вопрос "Как именно?" Ответ на этот вопрос изложен в данной главе, содержащей технические детали этого процесса в языке C++.

Язык C++ предоставляет две разновидности пользовательских типов: классы и перечисления. Классы носят намного более общий характер и играют более важную роль в программировании, поэтому мы сосредоточим свое внимание, в первую очередь, на них. Класс непосредственно выражает некую концепцию в программе. *Класс* (class) — это (пользовательский) тип, определяющий представление объектов этого класса, их создание, использование и уничтожение (раздел 17.5). Если вы размышляете о чем-то как об отдельной сущности, то, вполне возможно, должны определить класс, представляющий эту "вещь" в вашей программе. Примерами являются вектор, матрица, поток ввода, строка, быстрое преобразование Фурье, клапанный регулятор, рука робота, драйвер устройства, рисунок на экране, диалоговое окно, график, окно, термометр и часы.

В языке C++ (как и в большинстве современных языков) класс является основной строительной конструкцией в крупных программах, которая весьма полезна и для разработки небольших программ, как мы могли убедиться на примере калькулятора (см. главы 6 и 7).

9.2. Классы и члены класса

Класс — это пользовательский тип. Он состоит из встроенных типов, других пользовательских типов и функций. Компоненты, использованные при определении класса, называются его *членами* (members). Класс может содержать несколько членов, а может и не иметь ни одного члена. Рассмотрим пример.

```
class X {
public:
    int m;                  // Член-данные
    int mf(int v) {         // Функция-член
        int old = m; m=v; return old;
    }
};
```

Члены класса могут иметь разные типы. Большинство из них являются либо данными-членами, определяющими представление объекта класса, либо функциями-членами, описывающими операции над такими объектами. Для доступа к членам класса используется синтаксическая конструкция вида *объект.член*, например:

```
X var;              // var – переменная типа X
var.m = 7;          // Присваиваем значение члену m объекта var
int x = var.mf(9); // Вызываем функцию-член mf() объекта var
```

Тип члена определяет, какие операции с ним можно выполнять. Например, можно считывать и записывать член типа `int`, вызывать функцию-член и т.д.

Функция-член, такая как `mf()` класса `X`, не обязана использовать запись `var.m`. Она может использовать обычное имя члена (в данном примере — `m`). Внутри функции-члена имя члена ссылается на член с данным именем в объекте, для которого вызвана функция-член. Таким образом, в вызове `var.mf(9)` имя `m` в определении `mf()` означает `var.m`.

9.3. Интерфейс и реализация

Как правило, класс имеет интерфейс и реализацию. Интерфейс — это часть объявления класса, к которой его пользователь имеет прямой доступ. Реализация — это часть объявления класса, доступ к которой пользователь может получить только косвенно, через интерфейс. Открытый интерфейс идентифицируется меткой `public:`, а реализация — меткой `private:`. Объявление класса можно представить следующим образом:

```
class X {   // Класс имеет имя X
public:
    // Открытые члены:
```

```
    //   - пользовательский интерфейс (доступный всем)
    //     функции
    //     типы
    //     данные (но лучше держать их в разделе private)
private:
    // Закрытые члены:
    //   - детали реализации (используется только членами
    //     данного класса)
    //     функции
    //     типы
    //     данные
};
```

По умолчанию члены класса являются закрытыми. Иначе говоря, фрагмент

```
class X {
    int mf(int);
    // ...
};
```

означает

```
class X {
private:
    int mf(int);
    // ...
};
```

поэтому

```
X x;            // Переменная x типа X
int y = x.mf(); // Ошибка: функция mf закрыта
                // (т.е. недоступна)
```

Пользователь не может непосредственно обращаться к закрытому члену класса. Вместо этого он должен воспользоваться открытой функцией-членом, имеющей доступ к закрытым данным, например:

```
class X {
    int m;
    int mf(int);
public:
    int f(int i) { m=i; return mf(i); }
};

X x;
int y = x.f(2);
```

Мы используем закрытые и открытые члены для представления важного различия между интерфейсом (точка зрения пользователя класса) и деталями реализации (точка зрения разработчика класса). По мере изложения мы опишем эту концепцию более подробно и рассмотрим множество

примеров. Здесь же мы просто укажем, что для обычных структур, которые представляют собой просто данные, это различие не имеет значения. По этой причине имеется сокращенная запись для класса, не имеющего закрытых деталей реализации. Структура **struct** представляет собой **class**, в котором все члены открыты по умолчанию:

```
struct X {
    int m;
    // ...
};
```

эквивалентно

```
class X {
public:
    int m;
    // ...
};
```

Структуры в основном используются для организации данных, члены которых могут принимать любые значения; иначе говоря, мы не можем определить для них никакого осмысленного инварианта (раздел 9.4.3).

9.4. Разработка класса

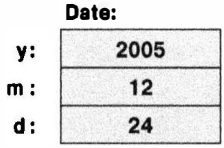
Проиллюстрируем языковые свойства, поддерживающие классы и основные методы их использования, на примере того, как — и почему — простую структуру данных можно преобразовать в класс с закрытыми деталями реализации и операциями. Рассмотрим вполне тривиальную задачу: представить календарную дату (такую, например, как 19 мая 1963 года) в программе. Даты нужны во многих программах (для проведения коммерческих операций, описания погодных данных, календаря, рабочих записей, ведомостей и т.д.). Остается только вопрос, как мы можем их представить.

9.4.1. Структура и функции

Как можно представить дату? На этот вопрос большинство людей отвечают так: "Указать год, месяц и день месяца". Это не единственный и далеко не наилучший ответ, но для наших целей он вполне подходит. Для начала попробуем создать простую структуру.

```
// Простая структура Date (слишком просто?)
struct Date {
     int y; // Год
     int m; // Месяц года
     int d; // День месяца
};

Date today; // Переменная типа Date (именованный объект)
```

Объект типа **Date**, например **today**, может просто состоять из трех чисел типа **int**.



В данном случае нет необходимости скрывать данные, на которых основана структура **Date**, — это предположение будет использовано во всех вариантах этой структуры на протяжении всей главы.

Итак, теперь у нас есть объекты типа **Date**; что с ними можно делать? Все что угодно, в том смысле, что мы можем получить доступ ко всем членам объекта **today** (и другим объектам типа **Date**), а также читать и записывать их по своему усмотрению. Загвоздка заключается в том, что все это не совсем удобно. Все, что мы хотим делать с объектами типа **Date**, можно выразить через чтение и запись их членов. Рассмотрим пример.

```
// Установить текущую дату 20 февраля 2004 года
today.y = 2004;
today.m = 20;
today.d = 2;
```

Этот способ утомителен и способствует появлению ошибок. Вы не заметили в приведенном фрагменте ошибку? Все, что является утомительным, способствует появлению ошибок! Например, ответьте, имеет ли смысл следующий код?

```
Date x;
x.y = -3;
x.m = 13;
x.d = 32;
```

Вероятно, нет, и никто не стал бы писать такую чушь. Или случайно стал бы? А что вы скажете о таком коде?

```
Date y;
y.y = 2000;
y.m = 2;
y.d = 29;
```

Был ли двухтысячный год високосным? Вы уверены?

Итак, нам нужны вспомогательные функции, которые выполняли бы для нас наиболее распространенные операции. В таком случае нам не придется повторять один и тот же код снова и снова, а также снова и снова искать и исправлять одни и те же ошибки. Практически для любого типа самыми распространенными операциями являются инициализация и присваивание. Для типа **Date** к распространенным операциям относится также увеличение значения объекта **Date**. Итак, напишем следующий код:

```
// Вспомогательные функции:
void init_day(Date& dd, int y, int m, int d)
{
    // Проверяет, является ли (y,m,d) правильной датой,
    // и, если является, то инициализирует объект dd
}

void add_day(Date& dd, int n)
{
    // Увеличивает объект dd на n дней
}
```

Попробуем использовать объект типа **Date**.

```
void f()
{
    Date today;
    init_day(today, 12, 24, 2005); // Ой! (В 12-м году не было
                                   // 2005-го дня.)
    add_day(today,1);
}
```

Для начала отметим полезность таких "операций" — здесь они реализованы в виде вспомогательных функций. Проверка корректности даты довольно сложна и утомительна, поэтому, если бы мы не написали соответствующую функцию раз и навсегда, то, скорее всего, где-то пропустили бы проверяющий код и получили бы неправильную программу. Когда мы определяем тип, то хотим выполнять над его объектами какие-то операции. Точное количество и вид этих операций может варьироваться. Варьируется и точный вид реализации этих операций (в виде функций, функций-членов или операторов), но как только мы решаем создавать собственный тип, мы должны спросить себя "Какие операции с этим типом можно выполнять?"

9.4.2. Функции-члены и конструкторы

Мы предусмотрели функцию инициализации для типа **Date**, которая проверяет корректность его объектов. Однако функции проверки приносят мало пользы, если мы не можем их использовать. Например, допустим, что мы определили для типа **Date** оператор вывода **<<** (раздел 9.8):

```
void f()
{
    Date today;
    // ...
    cout << today << '\n';     // Используем объект today
    // ...
    init_day(today,2008,3,30);
    // ...
    Date tomorrow;
```

```
    tomorrow.y = today.y;
    tomorrow.m = today.m;
    tomorrow.d = today.d+1;    // Добавляем единицу к today
    cout << tomorrow << '\n'; // Используем объект tomorrow
}
```

Здесь мы "забыли" инициализировать объект **today**, и до вызова функции **init_day()** этот объект будет иметь неопределенное значение. Кроме того, "кто-то" решил, что вызывать функцию **add_day()** — лишняя потеря времени (или просто не знал о ее существовании), и создал объект **tomorrow** вручную. Это не просто плохой — это очень плохой код. Иногда, а может быть, и в большинстве случаев, эта программа будет работать, но даже самые небольшие изменения приведут к серьезным ошибкам. Например, отсутствие инициализации объекта типа **Date** приведет к выводу на экран "мусора", а прибавление единицы к члену **d** вообще представляет собой мину с часовым механизмом: когда объект **today** окажется последним днем месяца, его увеличение на единицу приведет к появлению неправильной даты. Хуже всего в этом очень плохом коде то, что он не выглядит плохим.

Такие размышления приводят нас к мысли о необходимости функции инициализации, которую нельзя забыть, и об операциях, которые невозможно пропустить. Основным инструментом в этом механизме являются *функции-члены*, т.е. функции, объявленные как члены класса внутри его тела, например:

```
// Простая структура Date,
// гарантирующая инициализацию с помощью конструктора
// и обеспечивающая удобство обозначений
struct Date {
    int y, m, d;               // Год, месяц, день
    Date(int y, int m, int d); // Проверяем корректность даты
                               // и выполняем инициализацию
    void add_day(int n);       // Увеличиваем объект на n дней
};
```

Функция-член, имя которой совпадает с именем класса, является особой. Она называется *конструктором* (constructor) и используется для инициализации (конструирования) объектов класса. Если программист забудет проинициализировать объект класса, имеющего конструктор, требующий предоставления аргумента, то компилятор выдаст сообщение об ошибке. Для такой инициализации существует специальная синтаксическая конструкция.

```
Date my_birthday;          // Ошибка: объект не инициализирован
Date today{12,24,2007};    // Ошибка времени выполнения
Date last{2000,12,31};     // ОК (краткий стиль)
Date next = {2014,2,14};   // ОК (немного более многословно)
Date christmas =
        Date{1976,12,24}; // ОК (многословный стиль)
```

Попытка объявить объект **my_birthday** провалится, поскольку мы не указали требуемое начальное значение. Попытку объявить объект **today** компилятор пропустит, но проверочный код в конструкторе на этапе выполнения программы обнаружит неправильную дату (**{12,24,2007}** — 2007-й день 24-го месяца 12-го года).

Определение объекта **last** содержит в скобках сразу после имени переменной начальное значение — аргументы, требуемые конструктором класса **Date**, в виде списка в фигурных скобках **{}**. Этот стиль инициализации переменных класса, имеющего конструктор с аргументами, является наиболее распространенным. Можно также использовать более многословный стиль, который позволяет явно продемонстрировать создание объекта (в данном случае **Date{1976,12,24}**), а затем использовать его для инициализации с помощью синтаксиса инициализации **=**. Но если вы будете использовать этот стиль, то очень скоро вы просто утомитесь.

Теперь можно попробовать использовать "новоопределенные" переменные.

```
last.add_day(1);
add_day(2);    // Ошибка: какой именно объект типа Date?
```

Обратите внимание на то, что функция-член **add_day()** вызывается для конкретного объекта типа **Date** с помощью записи с точкой (обращение к члену класса). Как определить функцию-член класса, мы покажем в разделе 9.4.4.

В C++98 программисты использовали для списка инициализации круглые скобки, так что вам встретится масса кода наподобие следующего:

```
Date last(2000,12,31);   // OK (старый краткий стиль)
```

Мы предпочитаем использовать для списков инициализации фигурные скобки **{}**, которые более ясно указывают, когда выполняется инициализация (конструирование) объектов, а также потому что такая запись широко распространена. Эта запись может использоваться и для встроенных типов, например:

```
int x{7};  // Ok (современный стиль списка инициализации)
```

Перед списком **{}** может использоваться необязательный символ **=**:

```
Date next = {2014,2,14}; // OK (немного многословнее)
```

Такая комбинация старого и нового стилей некоторыми считается более удобочитаемой.

9.4.3. Сокрытие деталей

Остается одна проблема: что произойдет, если кто-то забудет использовать функцию-член **add_day()**? Что произойдет, если кто-то решит

непосредственно изменить месяц? Оказывается, мы "забыли" предусмотреть возможности для выполнения этой операции.

```
Date birthday{1960,12,31}; // 31 декабря 1960 года
++birthday.d;              // Неправильная дата!
                           // birthday.d == 32

Date today{1970,2,3};
today.m = 14;              // Неправильная дата!
                           // today.m == 14
```

До тех пор, пока представление **Date** доступно для всех, кто-то — вольно или невольно — может сделать ошибку; иначе говоря, сделать нечто, что приведет к созданию неправильной даты. В данном случае мы создали объект типа **Date** со значением, которое не соответствует календарю. Такие неправильные объекты являются минами с часовым механизмом; через какое-то время кто-нибудь, не ведая того, обязательно воспользуется некорректным значением и получит сообщение об ошибке во время выполнения программы или — что еще хуже — получит неверные результаты.

Такие размышления приводят к выводу, что представление типа **Date**, за исключением открытых функций-членов, должно быть недоступным для пользователей. Итак, получаем следующий вариант:

```
// Простой тип Date (управляемый доступ)
class Date {
    int y, m, d;                // Год, месяц, день
public:
    Date(int y, int m, int d); // Проверка и инициализация даты
    void add_day(int n);       // Увеличение Date на n дней
    int month() { return m; }
    int day()   { return d; }
    int year()  { return y; }
};
```

Этот класс можно использовать следующим образом:

```
Date birthday{1970, 12, 30};      // OK
birthday.m = 14;                  // Ошибка: Date::m закрыт
cout << birthday.month() << endl; // Доступ к переменной m
```

Понятие "корректный объект **Date**" — важный частный случай идеи корректного значения. Мы пытаемся разработать наши типы так, чтобы их значения гарантированно были корректными; иначе говоря, скрываем представление, предусматриваем конструктор, создающий только корректные объекты, и разрабатываем все функции-члены так, чтобы они получали объекты в корректном состоянии и оставляли их в таком же. Значение объекта часто называют *состоянием* (state), а корректное значение — *корректным состоянием* объекта.

В качестве альтернативы можно проверять корректность объекта при каждой попытке его использования или просто надеяться на то, что никто никогда не создаст ни одного некорректного значения. Опыт показывает, что такие надежды могут привести к неплохим программам. Однако создание программ, которые иногда выдают ошибочные результаты, а порой вообще приводят к аварийному отказу, не принесет вам профессионального признания. Мы предпочитаем писать программы, корректность которых можно продемонстрировать.

Правило, определяющее смысл корректного значения, называют *инвариантом* (invariant). Инвариант для класса **Date** ("Объект класса **Date** должен представлять дату в прошлом, настоящем или будущем") необычайно трудно сформулировать точно: вспомните о високосных годах, переходе с юлианского на григорианский календарь, часовых поясах и т.п. Однако для простых и реалистичных ситуаций написать класс **Date** — вполне доступная задача. Например, если мы инициализируем журнальные записи, нас не должны беспокоить ни григорианский, ни юлианский календари, ни даже календарь племени майя. Если мы не можем придумать хороший инвариант, то, вероятно, имеют место простые данные. В таких случаях следует использовать обычные структуры **struct**.

9.4.4. Определение функций-членов

До сих пор мы смотрели на класс **Date** с точки зрения разработчика интерфейса и пользователя. Однако рано или поздно нам придется реализовать его функции-члены. Начнем с того, что выделим подмножество класса **Date**, чтобы согласовать его с общепринятым стилем организации открытого интерфейса.

```
// Простой класс Date (детали реализации рассматриваются позже)
class Date {
public:
    Date(int y, int m, int d); // Проверка и инициализация даты
    void add_day(int n);       // Увеличение Date на n дней
    int month();
    // ...
private:
    int y, m, d;               // Год, месяц, день
};
```

Открытый интерфейс разрабатывают в первую очередь, поскольку именно он интересует большинство людей. В принципе, пользователю не обязательно знать детали реализации. На самом же деле люди, как правило, любопытны и хотят знать, насколько разумна реализация класса и какие приемы использовал ее автор, чтобы научиться у него чему-нибудь. И все же, если только реализацию класса создавали не мы, большую часть времени мы будем иметь дело с его открытым интерфейсом. Компилятору

безразличен порядок следования функций и данных, являющихся членами класса; он обрабатывает объявления в любом порядке, в котором мы их укажем.

Определяя члены за пределами класса, мы должны указать, какому классу они принадлежат. Для этого используется обозначение *имя_класса*: :*имя_члена*.

```
Date::Date(int yy, int mm, int dd)// Конструктор
    :y{yy}, m{mm}, d{dd}          // Инициализация членов
{
}

void Date::add_day(int n)
{
    // ...
}

int month()      // Ошибка: мы забыли указать Date::
{
    return m;  // Не функция-член, к переменной m доступа нет
}
```

Запись `:y{yy}, m{mm}, d{dd}` представляет собой инициализацию членов класса и называется списком инициализации (членов). Мы могли бы написать эквивалентный код:

```
Date::Date(int yy, int mm, int dd) // Конструктор
{
    y = yy;
    m = mm;
    d = dd;
}
```

Однако сначала нам следовало бы инициализировать члены их значениями, заданными по умолчанию, и лишь потом присваивать им новые значения. К тому же в этом случае не исключена возможность того, что мы случайно используем член класса до его инициализации. Запись `:y{yy}, m{mm}, d{dd}` точнее отражает наши намерения. Разница между этими фрагментами точно такая же, как между кодом

```
int x; // Сначала определяем переменную x...
// ...
x = 2; // ...потом присваиваем ей значение
```

и кодом

```
int x{2};  // Определяем и инициализируем переменную
```

Функцию-член класса можно также определить прямо в определении класса.

```
// Простой класс Date (детали реализации некоторые
//    программисты предпочитают размещать в конце)
```

```
class Date {
public:
    Date(int yy, int mm, int dd)
        :y{yy}, m{mm}, d{dd}
    {
        // ...
    }

    void add_day(int n)
    {
        // ...
    }

    int month() { return m; }

    // ...
private:
    int y, m, d;  // Год, месяц, день
};
```

Сразу же отметим, что теперь объявление класса стало больше и запутаннее. В данном примере код конструктора и функции `add_day()` могут содержать десятки строк. Это в несколько раз увеличивает размер объявления класса и затрудняет поиск интерфейса среди деталей реализации. Поэтому мы не рекомендуем определять большие функции в объявлении класса.

Однако посмотрите на определение функции `month()`. Оно проще и короче, чем определение `Date::month()`, размещенное за пределами объявления класса. Определения коротких и простых функций можно размещать в объявлении класса.

Обратите также внимание на то, что функция `month()` может обращаться к переменной `m`, несмотря на то, что эта переменная `m` определена позже (ниже) функции `month()`. Член класса может обращаться к другому члену класса независимо от того, в каком месте класса он определен. Правило, утверждающее, что имя переменной должно быть объявлено до ее использования, внутри класса ослабляется.

Определение функции-члена в классе приводит к следующим последствиям.

- Функция становится *встраиваемой* (inline), т.е. компилятор будет пытаться вместо вызова функции генерировать ее код. Это может дать значительное преимущество в производительности для часто вызываемых функций, таких как `month()`.

- При изменении тела встраиваемой функции-члена класса придется перекомпилировать заново все модули, в которых он используется. Если тело функции определено за пределами объявления класса, то потребуется перекомпилировать только само определение класса.

Отсутствие необходимости повторной компиляции при изменении тела функции может оказаться важным преимуществом в больших программах.

- Определение класса становится больше по размеру. Следовательно, будет сложнее находить требуемые члены среди определений функций-членов.

Очевидное правило гласит: не размещайте тела функций-членов в объявлении класса, если вам не нужна повышенная эффективность программы за счет использования небольших встраиваемых функций. Большие функции, скажем, состоящие из пяти и более строк, ничего не выиграют от подстановки. Не следует делать встраиваемыми функции, содержащие более одного-двух выражений.

9.4.5. Ссылка на текущий объект

Рассмотрим простой пример использования класса **Date**.

```
class Date {
    // ...
    int month() { return m; }
    // ...
private:
    int y, m, d; // Год, месяц, день
};

void f(Date d1, Date d2)
{
    cout << d1.month() << ' ' << d2.month() << '\n';
}
```

Откуда функции **Date::month()** известно, что при первом вызове следует вернуть значение переменной **d1.m**, а при втором — **d2.m**? Посмотрите на функцию **Date::month()** еще раз: ее объявление не имеет аргумента! Как же функция **Date::month()** "узнает", для какого объекта она вызывается? Функции-члены класса, такие как **Date::month()**, получают неявный аргумент, позволяющий идентифицировать объект, для которого они вызываются. Таким образом, при первом вызове переменная **m** правильно ссылается на **d1.m**, а при втором — на **d2.m**. Другие варианты использования неявного аргумента описаны в разделе 17.10.

9.4.6. Сообщения об ошибках

Что мы должны делать при обнаружении некорректной даты? В каком месте кода происходит поиск некорректных дат? В разделе 5.6 мы узнали, что ответом на первый вопрос служит "сгенерировать исключение", а самым очевидным местом для этого является соз-

дание объекта класса **Date**. Если мы создали правильные объекты класса **Date** и все функции-члены написаны правильно, то мы никогда не получим объект класса **Date** с неверным значением. Поэтому следует предотвратить создание неправильных объектов класса **Date**.

```
// Простой класс Date (предотвращаем неверные даты)
class Date {
public:
    class Invalid { };          // Используется для исключения
    Date(int y, int m, int d); // Проверка и инициализация даты
    // ...
private:
    int y, m, d;      // Год, месяц, день
    bool is_valid(); // Если дата правильная, возвращает true
};
```

Мы поместили проверку корректности даты в отдельную функцию **is_valid()**, потому что с логической точки зрения эта проверка отличается от инициализации, а также потому, что нам может потребоваться несколько конструкторов. Как видите, закрытыми могут быть не только данные, но и функции:

```
Date::Date(int yy, int mm, int dd)
    : y{yy}, m{mm}, d{dd}        // Инициализация данных-членов
{
    if (!is_valid()) throw Invalid{}; // Проверка корректности
}
bool Date::is_valid()    // Возвращает true для корректной даты
{
    if (m<1 || 12<m) return false;
    // ...
}
```

При таком определении класса **Date** можно написать следующий код:

```
void f(int x, int y)
try {
    Date dxy{2009,x,y};
    cout << dxy << '\n';     // Объявление << см. в разделе 9.8
    dxy.add_day(2);
}
catch(Date::Invalid) {
    error("неверная дата"); // Функция error() определена
                            // в разделе 5.6.3
}
```

Теперь мы знаем, что оператор **<<** и функция **add_day()** всегда будут работать с корректными объектами класса **Date**.

До завершения разработки класса **Date** в разделе 9.7 опишем некоторые возможности языка, которые потребуются нам для того, чтобы сделать эту работу хорошо: перечисления и перегрузку операторов.

9.5. Перечисления

Перечисление **enum** (enumeration) — это очень простой пользовательский тип, который задает множество значений (элементов перечисления) в виде символических констант:

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

"Тело" перечисления представляет собой просто список его элементов. Ключевое слово **class** в **enum class** означает, что перечисления находятся в области видимости перечисления, т.е., чтобы обратиться к элементу **jan**, мы должны написать **Month::jan**.

Каждому элементу перечисления можно задать определенное значение, как это сделано выше с элементом **jan**, или предоставить компилятору самому выбрать подходящее значение. Если положиться на компилятор, то он присвоит каждому элементу перечисления число, на единицу превышающее значение предыдущего. Таким образом, наше определение перечисления **Month** присваивает каждому месяцу последовательные значения, начиная с единицы. Это эквивалентно тому, как если бы мы написали

```
enum class Month {
    jan=1, feb=2, mar=3, apr=4,  may=5,  jun=6,
    jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
};
```

Однако это утомительно и открывает много возможностей для ошибок. На самом деле, набирая этот текст для книги, мы сделали две опечатки, пока не получили правильный вариант; все же выполнение простой, повторяющейся, "механической" работы лучше предоставить компилятору. Компилятор такие задачи решает лучше, чем люди, и при этом не устает.

Если не инициализировать первый элемент перечисления, то отсчет начнется с нуля:

```
enum class Day {
    monday, tuesday, wednesday,
    thursday, friday, saturday, sunday
};
```

Здесь **monday==0** и **sunday==6**. На практике лучше всего выбирать начальное значение счетчика равным нулю.

Перечисление **Month** можно использовать следующим образом:

```
Month m = Month::feb;

Month m2 = feb;       // Ошибка: feb не в области видимости
m = 7;                // Ошибка: нельзя присвоить значение
                      // int переменной Month
```

```
int n = m;              // Ошибка: нельзя присвоить значение
                        // Month переменной int
Month mm = Month(7); // (Непроверяемое) преобразование значения
                        // int в тип Month
```

Month представляет собой отдельный тип, отличный от "базового" типа **int**. Каждое значение типа **Month** имеет эквивалентное целочисленное значение, но большинство значений типа **int** не имеют эквивалентного значения типа **Month**. Например, мы преднамеренно написали неправильную инициализацию

```
Month bad = 9999;       // Ошибка: целое число невозможно
                        // преобразовать в объект типа Month
```

Если вы настаиваете на использовании записи **Month(9999)**, то компилятор с вами согласится. Во многих ситуациях язык C++ не пытается останавливать программиста от потенциально опасных действий, если программист явно на этом настаивает; в конце концов, программисту виднее! Заметим, что вы не можете использовать запись **Month{9999}**, поскольку такая запись допускает только те значения, которые могут использоваться в инициализации **Month**; значения **int** к таковым не относятся.

К сожалению, мы не можем определить конструктор для перечисления, который проверял бы значения инициализаторов, но написать простую функцию для проверки не составляет труда.

```
Month int_to_month(int x)
{
    if (x < int(Month::jan) || int(Month::dec) < x)
            error("неправильный месяц");
    return Month(x);
}
```

Мы использовали запись **int(Month::jan)** для получения целочисленного представления **Month::jan**. Теперь можно написать следующий код:

```
void f(int m)
{
    Month mm = int_to_month(m);
    // ...
}
```

Для чего нужны перечисления? В основном перечисления полезны, когда нам нужно множество связанных друг с другом именованных целочисленных констант. Как правило, с помощью перечислений представляют наборы альтернатив (**up**, **down**; **yes**, **no**, **maybe**; **on**, **off**; **n**, **ne**, **e**, **se**, **s**, **sw**, **w**, **nw**) или отличительных признаков (**red**, **blue**, **green**, **yellow**, **maroon**, **crimson**, **black**).

9.5.1. "Простые" перечисления

В дополнение к перечислениям **enum class**, известным как *перечисления с областью видимости* (scoped enumeration), имеются "простые" перечисления, которые отличаются от рассмотренных тем, что неявно "экспортируют" свои элементы в область видимости перечисления и допускают неявное преобразование в тип **int**, например:

```
enum Month { // Обратите внимание на отсутствие "class"
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;          // OK: feb в области видимости
Month m2 = Month::feb; // OK: можно и так
m = 7;                  // Ошибка: нельзя присваивать значение
                        // int переменной типа Month
int n = m;              // OK: можно присваивать значение Month
                        // переменной типа int
Month mm = Month(7);    // Преобразование значения int в
                        // тип Month (непроверяемое)
```

Очевидно, что "простые" перечисления являются менее строгими, чем **enum class**. Их элементы могут "загрязнять" область видимости, в которой определено перечисление. Это может быть удобно, но иногда приводит к сюрпризам. Например, если вы пытаетесь использовать такое перечисление **Month** вместе с механизмами форматирования **iostream** (раздел 11.2.1), то обнаружите коллизию, заключающуюся в том, что **dec** используется и для обозначения декабря, и для вывода десятичных чисел.

Аналогично преобразование значения перечисления в **int** может быть удобно (оно спасает нас от необходимости выполнения явного преобразования в тип **int**), но иногда приводит к сюрпризам, например:

```
void my_code(Month m)
{
    if (m == 17)      do_something();       // 17-й месяц?
    if (m == monday) do_something_else(); // Сравнение месяца с
                                          // понедельником?
}
```

Если **Month** является **enum class**, ни одно из условий не будет скомпилировано. Если **monday** является элементом "простого" перечисления **enum**, а не **enum class**, сравнение месяца с понедельником будет выполнено, но, скорее всего, с нежелательным результатом.

Следует предпочитать более простые и безопасные перечисления **enum class**, но в старом коде вы встретите множество "простых" перечислений **enum**: **enum class** являются новинкой C++11.

9.6. Перегрузка операторов

Для класса или перечисления можно определить практически все операторы, существующие в языке C++. Этот процесс называют *перегрузкой операторов* (operator overloading). Он применяется, когда требуется сохранить привычные обозначения для разрабатываемого нами типа. Рассмотрим пример.

```
enum class Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

Month operator++(Month& m)                  // Префиксный инкремент
{
    m = (m==Dec)?Jan:Month(int(m)+1); // Циклический переход
    return m;
}
```

Конструкция `?:` представляет собой оператор "арифметического если": переменная `m` становится равной `Jan`, если `(m==Dec)`, и `Month(int(m)+1)` в противном случае. Это довольно элегантный способ выразить цикличность календаря. Тип `Month` теперь можно использовать следующим образом:

```
Month m = Sep;
++m; // m становится равным Oct
++m; // m становится равным Nov
++m; // m становится равным Dec
++m; // m становится равным Jan (циклический переход)
```

Можно не соглашаться с тем, что применение инкремента перечисления `Month` является широко распространенным способом, заслуживающим реализации в виде отдельного оператора. Однако что вы скажете об операторе вывода? Его можно определить следующим образом:

```
vector<string> month_tbl;

ostream& operator<<(ostream& os, Month m)
{
    return os << month_tbl[int(m)];
}
```

Здесь предполагается, что объект `month_tbl` был инициализирован где-то в другом месте, так что, например, `month_tbl[Mar]` представляет собой строку `"March"` или какое-то другое подходящее название месяца (см. раздел 10.11.3).

Разрабатывая собственный тип, можно перегрузить для него практически любой оператор, предусмотренный в языке C++, например `+`, `-`, `*`, `/`, `%`, `[]`, `()`, `^`, `!`, `&`, `<`, `<=`, `>` или `>=`. Невозможно определить собственный оператор; конечно, можно представить, что програм-

мист захочет иметь операторы ** или $=, но язык C++ этого не допускает. Операторы можно определить только для установленного количества операндов; например, можно определить унарный оператор –, но невозможно перегрузить как унарный оператор <= ("меньше или равно"). Аналогично можно перегрузить бинарный оператор +, но нельзя перегрузить как бинарный оператор ! ("нет"). То есть язык позволяет использовать для определенных программистом типов существующий синтаксис, но не позволяет этот синтаксис расширять.

Перегруженный оператор должен иметь хотя бы один операнд с пользовательским типом.

```
int operator+(int,int); // Ошибка: нельзя перегрузить
                        // встроенный оператор +
Vector operator+ (const Vector&, const Vector &); // OK
Vector operator+=(const Vector&, int);            // OK
```

В общем случае мы не рекомендуем определять операторы для типа, если вы не уверены полностью, что это значительно улучшит ваш код. Кроме того, операторы следует определять, сохраняя их общепринятый смысл: оператор + должен обозначать сложение; бинарный оператор * — умножение; оператор [] — доступ; оператор () — вызов функции и т.д. Это просто совет, а не правило языка, но это хороший совет: общепринятое использование операторов, такое как символ + для сложения, значительно облегчает понимание программы. В конце концов, этот совет является результатом сотен лет опыта использования математических обозначений. И наоборот, малопонятные операторы и необычное использование операторов могут запутать программу и стать источником ошибок. Более на эту тему мы распространяться не будем. Просто в следующих главах применим перегрузку операторов в соответствующих местах.

Заметим, что наиболее интересными для перегрузки операторами являются не операторы +, –, *, и /, как зачастую предполагают многие начинающие программисты, а =, ==, !=, <, [] и ().

9.7. Интерфейсы классов

Ранее мы указывали, что открытый интерфейс и реализация класса должны быть отделены друг от друга. Поскольку в языке C++ остается возможность использовать простые структуры **struct**, некоторые профессионалы могут не согласиться с этим утверждением. Однако как разработать хороший интерфейс? Чем хороший интерфейс отличается от плохого? Частично на эти вопросы можно ответить только с помощью примеров, но существует несколько общих принципов, которые поддерживаются в языке C++.

- Интерфейс должен быть полным.
- Интерфейс должен быть минимальным.
- Класс должен иметь конструкторы.

- Класс должен поддерживать копирование (или явно запрещать его) (см. раздел 14.2.4).
- Следует предусмотреть тщательную проверку типов аргументов.
- Необходимо идентифицировать немодифицирующие функции-члены (см. раздел 9.7.4).
- Деструктор должен освобождать все ресурсы (см. раздел 17.5).

См. также раздел 5.5, в котором описано, как выявлять ошибки и сообщать о них во время выполнения программы.

Первые два принципа можно подытожить так: "Интерфейс должен быть как можно более малым, но не менее необходимого". Интерфейс должен быть малым, потому что его легче изучить и запомнить, а программист, занимающийся реализацией класса, не будет терять время на реализацию излишних или редко используемых функций. Кроме того, небольшой интерфейс означает, что, если что-то пойдет не так, как задумано, в поисках причин потребуется проверить лишь несколько функций. В среднем чем больше открытых функций, тем труднее найти ошибку. Пожалуйста, не усложняйте себе жизнь, создавая классы с открытыми данными! Но, разумеется, интерфейс должен быть полным, в противном случае он будет бесполезным. Нам не нужен интерфейс, который не позволяет нам делать то, что действительно необходимо.

Перейдем к изучению менее абстрактных и более реалистичных понятий, поддерживаемых в языке C++.

9.7.1. Типы аргументов

Определяя конструктор класса **Date** в разделе 9.4.3, мы использовали в качестве аргументов три переменные типа **int**. Это привело к проблемам.

```
Date d1{4,5,2005};  // Год 4, день 2005
Date d2{2005,4,5};  // 5 апреля или 4 мая?
```

Первая проблема (недопустимый день месяца) легко решается путем проверки в конструкторе. Однако вторую проблему (путаницу между месяцем и днем месяца) невозможно выявить с помощью кода, написанного пользователем. Она возникает из-за того, что существуют разные соглашения о записи дат; например, 4/5 в США означает 5 апреля, а в Англии — 4 мая. Поскольку эту проблему невозможно устранить с помощью вычислений, мы должны придумать что-то еще. Очевидным решением является использование типа **Month**.

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

// Простой класс Date (с использованием типа Month)
class Date {
public:
    Date(int y, Month m, int d); // Проверка корректности даты
                                 // и инициализация
    // ...
private:
    int y;   // Год
    Month m;
    int d;   // День
};
```

При использовании типа **Month** компилятор выдаст ошибку, если мы поменяем местами месяц и день. Кроме того, перечисление **Month** позволяет использовать символические имена. Такие имена, как правило, легче читать и записывать, чем работать с числами, а значит, меньше и риск допустить ошибку.

```
Date dx1{1998,4,3};           // Ошибка: 2-й аргумент не имеет
                              // тип Month
Date dx2{1998,4,Month::mar};  // Ошибка: 2-й аргумент не имеет
                              // тип Month
Date dx2{4,Month::mar,1998};  // Ошибка времени выполнения:
                              // день 1998
Date dx2{Month::mar,4,1998};  // Ошибка: 2-й аргумент не имеет
                              // тип Month
Date dx3{1998,Month::mar,30}; // ОК
```

Такой подход решает много проблем. Обратите внимание на квалификацию элемента перечисления **mar** именем перечисления: **Month::mar**. Мы не пишем **Month.mar**, так как **Month** не является объектом (**Month** — это тип), а **mar** не является его данным-членом (это символическая константа — элемент перечисления). Обозначение **::** используется после имени класса, перечисления или пространства имен (раздел 8.7), а **.** (точка) — после имени объекта.

Когда есть выбор, выявлять ошибки следует во время компиляции, а не во время выполнения программы. Мы предпочитаем, чтобы ошибки вылавливал компилятор, а не искать, в каком месте кода возникла проблема. Кроме того, для выявления ошибок на этапе компиляции не требуется писать и выполнять специальный проверочный код.

А нельзя ли подобным образом выявить путаницу между днем месяца и годом? Можно, но решение этой проблемы будет не таким простым и элегантным, как в случае типа **Month**; в конце концов, возможно, мы действительно имели в виду именно четвертый год. Даже если мы

ограничимся современной эпохой, в перечисление придется включать слишком много лет.

Вероятно, лучшее, что мы можем сделать (без точного значения о предназначении класса **Date**), — это написать код наподобие следующего:

```
class Year {                          // Год в диапазоне [min,max)
    static const int min = 1800;
    static const int max = 2200;
public:
    class Invalid { };
    Year(int x):y{x} {if (x<min || max<=x) throw Invalid{}; }
    int year() { return y; }
private:
    int y;
};

class Date {
public:
    Date(Year y, Month m, int d); // Проверка корректности даты
                                  // и инициализация
    // ...
private:
    Year y;
    Month m;
    int d;  // День
};
```

Теперь можно воспользоваться новым классом.

```
Date dx1 {Year{1998},4,3};            // Ошибка: второй
                                      // аргумент — не Month
Date dx2 {Year{1998},4,Month::mar};   // Ошибка: второй
                                      // аргумент — не Month
Date dx2 {4,Month::mar,Year{1998}};   // Ошибка: первый
                                      // аргумент — не Year
Date dx2 {Month::mar,4,Year{1998}};   // Ошибка: второй аргумент — не Month
Date dx3 {Year{1998},Month::mar,30}; // ОК
```

Следующая фатальная и неожиданная ошибка выявится только на этапе выполнения программы.

```
Date dx2{Year{4},Month::mar,1998}; // Ошибка времени выполнения:
                                   // Year::Invalid
```

Стоило ли выполнять дополнительную работу и вводить обозначения для лет? Естественно, это зависит от того, какие задачи вы собираетесь решать с помощью типа **Date**, но в данном случае мы сомневаемся в этом и не хотели бы создавать и использовать отдельный класс **Year**.

Когда мы программируем, то всегда спрашиваем сами себя о том, достаточно ли хорошим получилось данное приложение. Как правило, мы не можем позволить себе роскошь очень долго искать

идеальное решение, если уже нашли достаточно хорошее. Втягиваясь в поиски наилучшего решения, мы настолько запутаем программу, что она станет хуже, чем первоначальный вариант. Как сказал Вольтер, "Лучшее — враг хорошего".

Обратите внимание на слова **static const** в определениях переменных **min** и **max**. Они позволяют нам определить символические константы для целых типов в классах. Использование модификатора **static** по отношению к члену класса гарантирует, что в программе существует только одна копия его значения, а не по одной копии на каждый объект данного класса. В данном случае, поскольку инициализатор является константным выражением, мы могли бы использовать **constexpr** вместо **const**.

9.7.2. Копирование

Мы всегда должны создавать объекты, иначе говоря — всегда предусматривать инициализацию и конструкторы. Вероятно, это самые важные члены класса: для того чтобы написать их, необходимо решить, как инициализировать объект и что значит корректность его значений (т.е. определить инвариант). Даже просто размышления об инициализации помогут вам избежать ошибок.

Затем необходимо решить, можно ли копировать объекты и как это делать?

Для класса **Date** или перечисления **Month** ответ очевиден: копирование необходимо, и смысл *копирования* тривиален: просто копируются все члены класса. Фактически это и предусмотрено по умолчанию. Если не указано ничего другого, компьютер сделает именно это. Например, если **Date** используется для инициализации или стоит в правой части оператора присваивания, то все его члены будут скопированы.

```
Date holiday {1978, Month::jul, 4};    // Инициализация
Date d2 = holiday;
Date d3 = Date{1978, Month::jul, 4};
holiday = Date{1978, Month::dec, 24}; // Присваивание
d3 = holiday;
```

Этот код работает, как и ожидалось. Запись **Date{1978,Month::dec,24}** означает создание соответствующего неименованного объекта **Date**, который затем можно соответствующим образом использовать, например:

```
cout << Date{1978, Month::dec, 24};
```

В данном случае конструктор класса действует почти как литерал типа класса. Это часто удобнее, чем сначала создавать переменную или константу, а затем использовать ее лишь один раз.

А если нас не устраивает копирование по умолчанию? В таком случае мы можем либо определить собственное копирование (см. раздел 18.3), либо удалить копирующий конструктор и копирующее присваивание (см. раздел 14.2.4).

9.7.3. Конструкторы по умолчанию

Неинициализированные переменные могут быть источником серьезных ошибок. Для того чтобы решить эту проблему, в языке C++ предусмотрено понятие конструктора, гарантирующее, что каждый объект класса будет инициализирован. Например, мы объявили конструктор **Date::Date(int,Month,int)**, чтобы гарантировать, что каждый объект класса **Date** будет правильно проинициализирован. В данном случае это значит, что программист должен предоставить три аргумента соответствующих типов. Рассмотрим пример.

```
Date d0;                  // Ошибка: нет инициализатора
Date d1{};                // Ошибка: пустой инициализатор
Date d2{1998};            // Ошибка: слишком мало аргументов
Date d3{1,2,3,4};         // Ошибка: слишком много аргументов
Date d4{1,"jan",2};       // Ошибка: неверный тип аргумента
Date d5{1,Month::jan,2}; // ОК: использован конструктор
                          //     с тремя аргументами
Date d6{d5};              // ОК: использован
                          //     копирующий конструктор
```

Обратите внимание, что несмотря на то, что мы определили конструктор для класса **Date**, мы по-прежнему можем копировать объекты класса **Date**.

Многие классы имеют вполне разумные значения по умолчанию; иначе говоря, для них существует очевидный ответ на вопрос "Какое значение следует использовать, если инициализация не выполнена?" Рассмотрим пример.

```
string s1;          // Значение по умолчанию: пустая строка ""
vector<string> v1; // Значение по умолчанию: вектор без элементов
```

Все это выглядит вполне разумно и работает в соответствии с указанными комментариями. Это достигается за счет того, что классы **vector** и **string** имеют *конструкторы по умолчанию*, которые неявно выполняют желательную инициализацию.

Для типа **T** обозначение **T{}** представляет собой запись для значения по умолчанию, определенного конструктором, заданным по умолчанию, так что можно написать следующий код:

```
string s1 = string{};                  // Значение по умолчанию:
                                       // пустая строка ""
vector<string> v1 = vector<string>{}; // Значение по умолчанию:
                                       // вектор без элементов
```

Однако мы предпочитаем эквивалентный более краткий стиль.

```
string s1;          // Значение по умолчанию: пустая строка ""
vector<string> v1; // Значение по умолчанию: вектор без элементов
```

Для встроенных типов, таких как **int** и **double**, конструктор по умолчанию подразумевает значение 0, так что запись **int{}** — это просто усложненное представление нуля, а **double{}** — длинный способ записать число **0.0**.

Использование конструктора, заданного по умолчанию, — это не просто вопрос стиля. Представьте себе, что у нас могут быть неинициализированные объекты классов **string** и **vector**.

```
string s;
for (int i=0; i<s.size(); ++i) // Ошибка: цикл выполняется
                               // неопределенное количество раз!
    s[i] = toupper(s[i]);      // Ошибка: изменяется содержимое
                               // случайной ячейки памяти!
vector<string> v;
v.push_back("bad");            // Ошибка: запись по
                               // случайному адресу!
```

Если значения переменных **s** и **v** действительно не определены, то неизвестно, сколько элементов они содержат или (при общепринятом способе реализации; см. раздел 17.5) не ясно, где эти элементы должны храниться. В результате будут использованы случайные адреса, а это худшая разновидность ошибки! В принципе, без конструктора мы не можем установить инвариант, поскольку не можем гарантировать, что значения переменных будут корректными (см. раздел 9.4.3). Мы вынуждены настаивать на инициализации таких переменных, так что код следует переписать следующим образом:

```
string s1 = "";
vector<string> v1{};
```

Однако этот код не кажется нам таким уж хорошим. Для объекта класса **string** строка **""** является очевидным обозначением пустой строки, а для объекта класса **vector** легко догадаться, что **{}** означает вектор без элементов. Однако для многих типов не так легко найти разумную запись значения по умолчанию. В таких случаях лучше было бы определить конструктор, создающий объект без использования явной инициализации. Такие конструкторы не имеют аргументов и называются *конструкторами по умолчанию*.

Для дат не существует очевидного значения по умолчанию[1]. По этой причине мы до сих пор не определяли для класса **Date** конструктор по умолчанию, но сейчас сделаем это (просто чтобы показать, что мы можем это сделать).

[1] Зачастую такой "датой по умолчанию" становится текущая дата. — *Примеч. ред.*

```
class Date {
public:
     // ...
     Date(); // Конструктор по умолчанию
     // ...
private:
     int y;
     Month m;
     int d;
};
```

Теперь мы должны выбрать дату по умолчанию. Для этого вполне подходит первый день XXI столетия.

```
Date::Date()
    :y{2001}, m{Month::jan}, d{1}
{
}
```

Значения по умолчанию для членов можно указать не в конструкторе, а в самих членах:

```
class Date {
public:
    // ...
    Date();        // Конструктор по умолчанию
    Date(year, Month, day);
    Date(int y);   // 1 января года y
    // ...
private:
    int   y {2001};
    Month m {Month::jan};
    int   d {1};
};
```

При этом значения по умолчанию становятся доступными каждому конструктору:

```
Date::Date(int y)                    // 1 января года y
:y{yy}
{
    if (!is_valid()) throw Invalid{}; // Проверка корректности
}
```

Поскольку `Date(int)` не инициализирует явно месяц (`m`) и день (`d`), неявно используются указанные инициализаторы (`Month::jan` и `1`). Инициализатор для члена класса, определенный как часть объявления класса, называется *инициализатором в классе* (in-class initializer).

Если не хотите встраивать значение, заданное по умолчанию, в код конструктора, то можете использовать константу (или переменную). Для того чтобы избежать использования глобальных переменных и связанных

с ними проблем инициализации, можно использовать прием, описанный в разделе 8.6.2.

```
const Date& default_date()
{
     static Date dd{2001,Month::jan,1};
     return dd;
}
```

Ключевое слово **static** здесь использовано для того, чтобы переменная **dd** создавалась только один раз, а не каждый раз при очередном вызове функции **default_date()**. Инициализация этой переменной происходит при первом вызове функции **default_date()**. С помощью функции **default_date()** легко определить конструктор по умолчанию для класса **Date**.

```
Date::Date()
    :y{default_date().year()},
     m{default_date().month()},
     d{default_date().day()}
{
}
```

Обратите внимание, что конструктор по умолчанию не обязан проверять значение, заданное по умолчанию; это уже сделано конструктором, создавшим объект, возвращаемый функцией **default_date()**. Имея конструктор по умолчанию для класса **Date**, мы можем создавать векторы объектов класса **Date** без перечисления значений элементов:

```
vector<Date> birthdays(10);
```

Без конструктора по умолчанию мы были бы вынуждены указывать значение по умолчанию явно.

```
vector<Date> birthdays(10,default_date()); // Десять объектов
                                           // Date по умолчанию
vector<Date> birthdays2 = {                // Десять объектов
                                           // Date по умолчанию
    default_date(), default_date(), default_date(),
    default_date(), default_date(), default_date(),
    default_date(), default_date(), default_date(),
    default_date()
};
```

При указании количества элементов в векторе мы используем круглые, а не фигурные скобки, чтобы избежать неоднозначности в случае **vector<int>** (§18.2).

9.7.4. Константные функции-члены

Некоторые переменные предназначены для того, чтобы изменяться (собственно, на то они и "переменные"), а некоторые — нет; иначе говоря,

существуют переменные, представляющие неизменные значения. Обычно их называют *константами*, и для их описания используется ключевое слово **const**, например:

```
void some_function(Date& d, const Date& start_of_term)
{
    int a = d.day();              // OK
    int b = start_of_term.day(); // Должно быть OK (почему?)
    d.add_day(3);                 // Отлично
    start_of_term.add_day(3);     // Ошибка
}
```

Здесь подразумевается, что переменная **d** — изменяема, а переменная **start_of_term** — нет; другими словами, функция **some_function()** не может изменить переменную **start_of_term**. Откуда компилятору это известно? Дело в том, что мы сообщили ему об этом, объявив переменную **start_of_term** константой (**const**). Тогда почему же с помощью функции **day()** нельзя прочитать переменную **day** из объекта **start_of_term**? В соответствии с предыдущим определением класса **Date** вызов **start_of_term.day()** считается ошибкой, поскольку компилятор не знает, что функция **day()** не изменяет свой объект класса **Date**. Об этом в программе нигде не сказано, поэтому компилятор предполагает, что функция **day()** может модифицировать свой объект класса **Date**, и выдает сообщение об ошибке.

Решить эту проблему можно, разделив операции над классом на модифицирующие и немодифицирующие. Это не только поможет понять суть класса, но и имеет очень важное практическое значение: операции, которые не модифицируют объект, можно применять к константным объектам. Рассмотрим пример.

```
class Date {
public:
    // ...
    int day() const;          // Константный член: модифицировать
                              // объект не может
    Month month() const;      // Константный член: модифицировать
                              // объект не может
    int year() const;         // Константный член: модифицировать
                              // объект не может

    void add_day(int n);      // Неконстантный член: может
                              // модифицировать объект
    void add_month(int n);    // Неконстантный член: может
                              // модифицировать объект
    void add_year(int n);     // Неконстантный член: может
                              // модифицировать объект
private:
    int y;      // Год
    Month m;    // Месяц
    int d;      // День месяца
```

```
};

Date d {2000, Month::jan, 20};
const Date cd {2001, Month::feb, 21};

cout << d.day() << " - "
     << cd.day() << '\n';  // OK
d.add_day(1);              // OK
cd.add_day(1);             // Ошибка: cd - константный объект
```

Ключевое слово **const** в объявлении функции-члена сразу после списка аргументов означает, что эту функцию-член можно вызывать для константных объектов. Как только мы объявили функцию-член константной, компилятор строго следит за нашим обещанием не модифицировать объект:

```
int Date::day() const
{
    ++d; // Ошибка: попытка изменить объект
         // в константной функции-члене
    return d;
}
```

Естественно, мы не собираемся мошенничать. В основном компилятор обеспечивает защиту от несчастных случаев, что очень полезно при разработке сложных программ.

9.7.5. Члены и вспомогательные функции

Разрабатывая минимальный (хотя и полный) интерфейс, мы вынуждены оставлять за бортом много полезных операций. Функцию, которая может быть просто, элегантно и эффективно реализована как самостоятельная функция (т.е. не функция-член), следует реализовывать за пределами класса. При этом функция не сможет повредить данные, хранящиеся в объекте класса. Предотвращение доступа к данным является важным фактором, поскольку обычные методы поиска ошибок "вращаются вокруг типичных подозрительных мест"; иначе говоря, если с классом что-то не так, мы в первую очередь проверяем функции, имеющие прямой доступ к его представлению: определенно, одна из них является причиной ошибки. Если таких функций десяток, работать будет намного проще, чем если их пятьдесят.

Пятьдесят функций для класса **Date**! Возможно, вы думаете, что мы шутим. Вовсе нет: несколько лет назад я делал обзор нескольких коммерческих библиотек для работы с календарем и обнаружил в них множество функций вроде **next_Sunday()**, **next_workday()** и т.д. Пятьдесят — это совсем не невероятное число для класса, разработанного для удобства пользователей, а не для удобства его проектирования, реализации и сопровождения.

Отметим также, что если меняется представление, то достаточно переписать только те функции, которые имеют к нему прямой доступ. Это вторая важная практическая причина для минимизации интерфейса. Разрабатывая класс **Date**, мы могли бы решить, что дату лучше представлять в виде целого числа дней, прошедших с 1 января 1900 года, а не в виде тройки (год, месяц, день). В этом случае нам пришлось бы изменить только функции-члены.

Вот несколько примеров *вспомогательных функций* (helper functions).

```
Date next_Sunday(const Date& d)
{
    // Имеет доступ к объекту d, используя d.day(),
    // d.month() и d.year(); создает и
    // возвращает новый объект Date
}

Date next_weekday(const Date& d) { /* ... */ }

bool leapyear(int y) { /* ... */ }

bool operator == (const Date& a, const Date& b)
{
    return a.year()  == b.year()
        && a.month() == b.month()
        && a.day()   == b.day();
}

bool operator != (const Date& a, const Date& b)
{
     return !(a==b);
}
```

Вспомогательные функции также называют *функциями-помощниками*, *дополнительными функциями* и многими иными терминами. Различие между ними и другими функциями, не являющимися членами класса, не более чем логическое; иначе говоря, вспомогательная функция представляет собой концепцию проектирования, а не концепцию языка программирования. Вспомогательные функции часто получают в качестве аргументов объекты класса, для которого они играют вспомогательную роль (хотя существуют и исключения; таковой является приведенная выше функция **leapyear()**). Часто для идентификации вспомогательных функций используются пространства имен (см. раздел 8.7):

```
namespace Chrono {
    enum class Month { /* ... */ };
    class Date { /* ... */ };
    bool is_date(int y, Month m, int d); // true для
                                         // корректной даты
    Date next_Sunday(const Date& d)  { /* ... */ }
```

```
    Date next_weekday(const Date& d) { /* ... */ }
    bool leapyear(int y) { /* ... */ }   // См. упр. 10
    bool operator==(const Date& a, const Date& b) {
            /* ... */ }
    // ...
}
```

Обратите внимание на функции == и !=. Это типичные вспомогательные функции. Для многих классов функции == и != имеют очевидный смысл, но, поскольку это не распространяется на все классы, компилятор не может создать их вместо программиста, как создает копирующий конструктор или копирующее присваивание.

Заметим также, что мы ввели вспомогательную функцию `is_date()`, которая заменяет функцию `Date::check()`, поскольку проверка корректности даты в целом не зависит от представления класса `Date`. Например, нам не нужно знать, как представлены объекты класса `Date` для того, чтобы узнать, что дата "30 января 2008 года" является корректной, а "30 февраля 2008 года" — нет. Возможно, существуют аспекты даты, которые зависят от ее представления (например, корректна ли дата "30 января 1066 года"), но (при необходимости) об этом может позаботиться конструктор `Date`.

9.8. Класс `Date`

Итак, соединим все идеи и понятия вместе и посмотрим, как может выглядеть класс `Date`. Там, где тело функции содержит лишь комментарий `...`, фактическая реализация слишком сложна (пожалуйста, не пытайтесь пока что ее написать). Сначала разместим объявления в заголовочном файле `Chrono.h`.

```
// Файл Chrono.h

namespace Chrono {
enum class Month {
    jan=1, feb, mar, apr, may, jun,
    jul, aug, sep, oct, nov, dec
};

class Date {
public:
    class Invalid { };              // Для генерации исключений
    Date(int y, Month m, int d); // Проверка корректности
                                    // и инициализация
    Date();                         // Конструктор по умолчанию
    // Копирование по умолчанию нас устраивает

    // Немодифицирующие операции:
    int day() const { return d; }
    Month month() const { return m; }
```

```
    int year() const { return y; }

    // Модифицирующие операции:
    void add_day(int n);
    void add_month(int n);
    void add_year(int n);
private:
    int y;
    Month m;
    int d;
};

bool is_date(int y, Month m, int d); // true для корректной даты
bool leapyear(int y);                // true для високосного года
bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);
ostream& operator<<(ostream& os, const Date& d);
istream& operator>>(istream& is, Date& dd);
} // Chrono
```

Определения находятся в файле `Chrono.cpp`.

```
// Файл Chrono.cpp
#include "Chrono.h"
namespace Chrono {
// Определения функций-членов:
Date::Date(int yy, Month mm, int dd)
    : y{yy}, m{mm}, d{dd}
{
    if (!is_date(yy,mm,dd)) throw Invalid{};
}

const Date& default_date()
{
    static Date dd {2001,Month::jan,1}; // Начало XXI века
    return dd;
}

Date::Date()
    :y{default_date().year()},
     m{default_date().month()},
     d{default_date().day()}
{
}

void Date:: add_day(int n)
{
    // ...
}

void Date::add_month(int n)
{
```

```
    // ...
}

void Date::add_year(int n)
{
    if (m==Month::feb && d==29 &&
        !leapyear(y+n)) { // В невисокосный год
        m = Month::mar;   // 29 февраля превращается в 1 марта
        d = 1;
    }
    y+=n;
}

// Вспомогательные функции:
bool is_date(int y, Month m, int d)
{
    // Полагаем y корректным
    if (d<=0) return false; // d должно быть положительным
    if (m<Month::jan || Month::dec<m) return false;

    int days_in_month = 31; // В месяце не более 31 дня
    switch (m) {
    case Month::feb:     // Длина февраля бывает разной
        days_in_month = (leapyear(y))?29:28;
        break;
    case Month::apr: case Month::jun:
    case Month::sep: case Month::nov:
        days_in_month = 30; // В этих месяцах 30 дней
        break;
    }
    if (days_in_month<d) return false;
    return true;
}

bool leapyear(int y)
{
    // См. упр. 10
}

bool operator==(const Date& a, const Date& b)
{
    return a.year()  == b.year()
        && a.month() == b.month()
        && a.day()   == b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}
```

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
              << ',' << int(d.month())
              << ',' << d.day() << ')';
}

istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1!= '(' || ch2!=',' ||
        ch3!=',' || ch4!=')') {       // Ошибка формата
        is.clear(ios_base::failbit); // Устанавливаем бит ошибки
        return is;
    }
    dd = Date(y, Month(m),d);         // Обновляем dd
    return is;
}

enum class Day {
    sunday, monday, tuesday, wednesday,
    thursday, friday, saturday
};

Day day_of_week(const Date& d)
{
    // ...
}

Date next_Sunday(const Date& d)
{
    // ...
}

Date next_weekday(const Date& d)
{
    // ...
}
} // Chrono
```

Функции, реализующие операции >> и << для класса **Date**, будут подробно рассмотрены в разделах 10.8 и 10.9.

Задание

Это задание следует выполнить с каждой из версий класса `Date`, перечисленных ниже. Для каждой версии определите объект класса `Date` с именем `today`, инициализированный датой 19 мая 1963 года. Затем определите объект класса `Date` с именем `tomorrow` и присвойте ему значение, скопировав в него объект `today` и увеличив его день на единицу с помощью функции `add_day()`. Выведите на печать объекты `today` и `tomorrow`, используя оператор `<<`, определенный в разделе 9.8.

Проверка корректности даты может быть очень простой. Вы можете не обращать внимания на наличие високосных лет. Просто не допускайте, чтобы месяц выходил за пределы диапазона [1,12], а день месяца — за пределы диапазона [1,31]. Проверьте каждую версию хотя бы на одной некорректной дате, например (2009,13,–5).

1. Версия из раздела 9.4.1.
2. Версия из раздела 9.4.2.
3. Версия из раздела 9.4.3.
4. Версия из раздела 9.7.1.
5. Версия из раздела 9.7.4.

Контрольные вопросы

1. Какие две части класса описаны в главе?
2. В чем заключается разница между интерфейсом и реализацией класса?
3. Какие ограничения и проблемы, связанные со структурой `Date`, описаны в этой главе?
4. Почему в классе `Date` используется конструктор, а не функция `init_day()`?
5. Что такое инвариант? Приведите примеры.
6. Когда функцию следует разместить в определении класса, а когда она должна быть определена за его пределами? Почему?
7. Когда следует применять перегрузку оператора? Перечислите операторы, которые вы могли бы захотеть перегрузить (укажите причины такого желания).
8. Почему открытый интерфейс класса должен быть минимальным?
9. Что изменится, если к объявлению функции-члена добавить ключевое слово `const`?
10. Почему вспомогательные функции лучше всего размещать за пределами класса?

Термины

`class`
`const`
`enum`
`struct`
вспомогательная функция
встраивание
встроенные типы
деструктор
инвариант
инициализатор в классе
интерфейс
конструктор
корректное состояние
перечисление
пользовательские типы
представление
реализация
структура
элемент перечисления

Упражнения

1. Перечислите разумные операции для реальных объектов, указанных в разделе 9.1 (например, для тостера).
2. Разработайте и реализуйте класс **`Name_pairs`**, содержащий пару (имя, возраст), где имя — объект класса **`string`**, а возраст — переменная типа **`double`**. Представьте эти члены класса в виде объектов классов **`vector<string>`** (с именем **`name`**) и **`vector<double>`** (с именем **`age`**). Предусмотрите операцию ввода **`read_names()`**, считывающую ряд имен. Предусмотрите операцию **`read_ages()`**, предлагающую пользователю ввести возраст для каждого имени. Предусмотрите операцию **`print()`**, которая выводит на экран пары (**`name[i]`**,**`age[i]`**) (по одной в строке) в порядке, определенном вектором **`name`**. Предусмотрите операцию **`sort()`**, упорядочивающую вектор **`name`** в алфавитном порядке и соответствующим образом реорганизующую вектор **`age`**. Реализуйте все "операции" как функции-члены. Протестируйте этот класс (конечно, тестировать надо как можно раньше и чаще).
3. Замените функцию **`Name_pair::print()`** (глобальным) оператором **`operator<<`** и определите операции == и != для объектов класса **`Name_pair`**.
4. Посмотрите на "головоломный" пример из раздела 8.4. Корректно расставьте в нем отступы и поясните смысл каждой конструкции. Заметим, что этот код не делает никаких осмысленных операций; это просто "запудривание мозгов".
5. Для выполнения этого и нескольких следующих упражнений необходимо разработать и реализовать класс **`Book`**, который является частью программного обеспечения библиотеки. Класс **`Book`** должен иметь члены для хранения кода ISBN, названия, фамилии автора и даты регистрации авторских прав. Кроме того, он должен хранить данные о том, выдана книга на руки или нет. Создайте функции, возвращающие эти данные. Создайте функции, проверяющие, выдана ли книга на руки. Предусмотрите простую проверку данных, которые вводятся в объект класса **`Book`**; например, код ISBN допускается только в форме

`n-n-n-x`, где `n` — целое число; `x` — цифра или буква. Храните код ISBN как `string`.

6. Добавьте операторы в класс `Book`. Пусть оператор `==` проверяет, совпадают ли коды ISBN у двух книг. Пусть оператор `!=` также сравнивает коды ISBN, а оператор `<<` выводит в поток название, фамилию автора и код ISBN в отдельных строках.
7. Создайте перечисление для класса `Book` с именем `Genre`. Предусмотрите типы для фантастики, прозы, периодических изданий, биографий и детской литературы. Отнесите каждую книгу к определенному жанру `Genre` и внесите соответствующие изменения в конструктор класса `Book` и его функции-члены.
8. Создайте класс `Patron` для библиотеки. Этот класс должен содержать имя пользователя, номер библиотечной карточки, а также размер членского взноса. Предусмотрите функции, имеющие доступ к этим членам, а также функцию, устанавливающую размер членского взноса. Предусмотрите вспомогательный метод, возвращающий булево значение (`bool`), указывающее, заплатил ли пользователь членские взносы.
9. Создайте класс `Library`. Включите в него векторы классов `Book` и `Patron`. Включите также структуру `Transaction` и предусмотрите в ней члены классов `Book`, `Patron` и класс `Date` из текста данной главы. Создайте вектор объектов класса `Transaction`. Создайте функции, добавляющие записи о книгах и клиентах библиотеки, а также о состоянии книг (выдана ли книга читателю). Если пользователь взял книгу, библиотека должна быть уверена, что пользователь является ее клиентом, а книга принадлежит ее фондам. Если эти условия не выполняются, выдайте сообщение об ошибке. Проверьте также, нет ли у пользователя задолженности по уплате членских взносов. Если задолженность есть, выдайте сообщение об ошибке. Если нет, создайте объект класса `Transaction` и поместите его в вектор объектов класса `Transaction`. Напишите также функцию, возвращающую вектор, содержащий имена всех клиентов, имеющих задолженность.
10. Реализуйте функцию `leapyear()` из раздела 9.8.
11. Разработайте и реализуйте набор полезных вспомогательных функций для класса `Date`, включая такие функции, как `next_workday()` (в предположении, что любой день, кроме субботы и воскресенья, является рабочим) и `week_of_year()` (в предположении, что первая неделя начинается 1 января, а первый день недели — воскресенье).
12. Измените представление класса `Date` на хранение количества дней, прошедших с 1 января 1970 года (так называемый нулевой день), в переменной типа `long int`, и соответствующим образом переработайте функции из раздела 9.8. Предусмотрите отбрасывание всех дат,

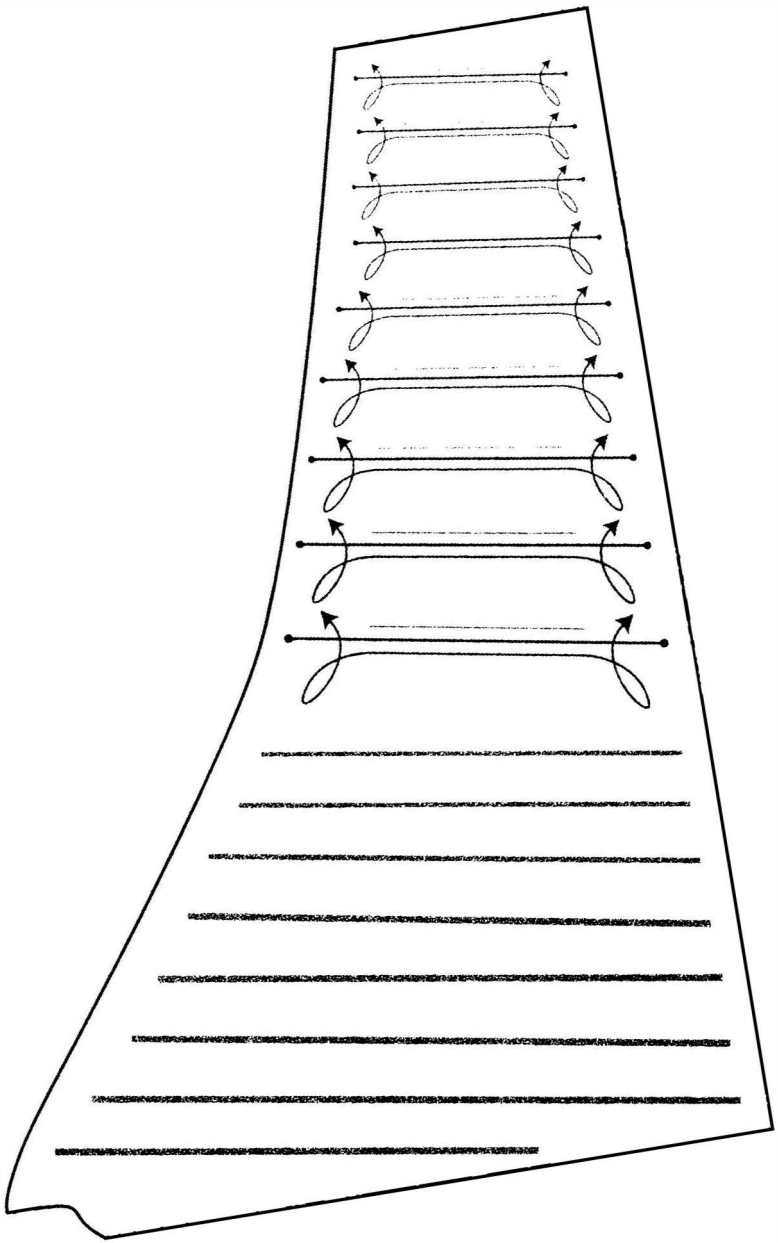
которые находятся за представимыми данным образом пределами (например, смело отбрасывайте все даты, предшествующие нулевому дню, т.е. не допускайте отрицательных дней).

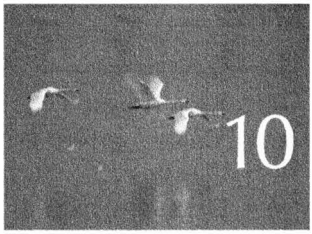
13. Разработайте и реализуйте класс для представления рациональных чисел `Rational`. Рациональное число состоит из двух частей: числителя и знаменателя, например 5/6 (пять шестых, или .83333). Предусмотрите операторы присваивания, сложения, вычитания, умножения, деления и проверки равенства. Кроме того, предусмотрите преобразование в тип `double`. Зачем может понадобиться такой класс `Rational`?
14. Разработайте и реализуйте класс `Money` для вычислений, связанных с долларами и центами, точность которых определяется по правилу округления 4/5 (0,5 цента округляется вверх; все, что меньше 0,5, округляется вниз). Денежные суммы должны представляться в центах с помощью переменной типа `long`, но ввод и вывод должны использовать доллары и центы, например $123.45. Не беспокойтесь о суммах, выходящих за пределы диапазона типа `long int`.
15. Доработайте класс `Money`, добавив в него валюту (как аргумент конструктора). Разрешите использовать инициализаторы с плавающей точкой, если эти значения точно представимы с помощью типа `long int`. Не допускайте некорректных операций. Например, выражение `Money*Money` не имеет смысла, а `USD1.23+DKK5.00` имеет смысл, только если существует таблица преобразования, определяющая обменный курс между американскими долларами (USD) и датскими кронами (DKK).
16. Определите оператор ввода `>>`, который считывает денежные суммы с указанием валют (например, `USD1.23` или `DKK5.00`) в переменную типа `Money`. Определите также соответствующий оператор вывода `<<`.
17. Приведите пример вычислений, когда класс `Rational` позволяет получить математически лучшие результаты, чем класс `Money`.
18. Приведите пример вычислений, в котором класс `Rational` позволяет получить математически лучшие результаты, чем тип `double`.

Послесловие

Существует множество пользовательских типов. Их гораздо больше, чем показано здесь. Пользовательские типы, особенно классы, образуют сердце языка C++ и являются ключом ко многим наиболее эффективным методам проектирования. Большая часть оставшихся глав книги посвящена проектированию и использованию классов. Класс (или набор классов) — это механизм, позволяющий выразить наши концепции в виде кода. В этой главе мы изложили в основном языковые аспекты классов; в последующих главах мы сосредоточимся на том, как элегантно выразить в виде классов полезные идеи.

Часть II
Ввод и вывод





Потоки ввода и вывода

Наука — это знания о том,
как не дать себя одурачить.
— Ричард Фейнман (Richard P. Feynman)

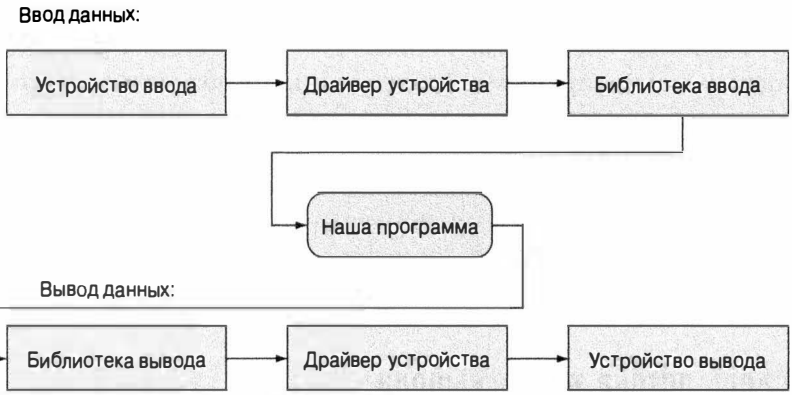
В этой и следующей главах описываются стандартные средства ввода и вывода в языке C++: потоки ввода-вывода. Мы расскажем, как читать и записывать файлы, как обрабатывать ошибки, а также применять операторы ввода-вывода к пользовательским типам. В центре внимания данной главы находится базовая модель: как читать и записывать отдельные значения, а также как открывать, читать и записывать целые файлы. В заключительном примере приводится большой фрагмент кода, иллюстрирующий эти аспекты программирования. Детальное описание этих вопросов вы найдете в следующей главе.

10.1. Ввод и вывод

Вычисления без данных бессмысленны. Для выполнения интересующих нас вычислений мы должны ввести в программу данные и получить результаты. В разделе 4.1 мы уже упоминали о чрезвычайном разнообразии источников данных и адресатов для вывода. Если мы не будем учитывать этот факт, то будем писать программы, получающие входные данные только из одного конкретного источника и выдающие результаты только на одно конкретное устройство вывода. В определенных приложениях, например цифровых фотоаппаратах или сенсорах топливного инжектора, это может быть приемлемым (а иногда даже необходимым), но при решении задач более общего характера нам необходимо отделять способы, с помощью которых программа читает и записывает данные, от реальных устройств ввода и вывода. Если бы мы были вынуждены непосредственно обращаться к устройствам разных видов, то каждый раз, когда на рынке появляется новый экран или диск, мы должны были бы изменять свою программу или ограничивать пользователей лишь теми экранами и дисками, которые нам нравятся. Разумеется, это абсурд.

Большинство современных операционных систем поручают управление устройствами ввода-вывода специализированным драйверам, а затем программы обращаются к ним с помощью средств библиотеки ввода-вывода, обеспечивающих максимально единообразную связь с разными источниками и адресатами данных. В общем случае драйверы устройств глубоко внедрены в операционную систему и недоступны для большинства пользователей, а библиотечные средства ввода-вывода обеспечивают абстракцию ввода-вывода, так что программист не должен думать об устройствах и их драйверах.

При использовании такой модели вся входная и выходная информация может рассматриваться как потоки байтов (символов), обрабатываемые



средствами библиотеки ввода-вывода. Более сложные разновидности ввода-вывода требуют специализированных подходов и выходят за рамки данной книги. Наша работа как программистов, создающих приложения, сводится к следующему.

1. Настроить потоки ввода-вывода для получения данных из устройств ввода и вывода данных на устройства вывода.
2. Читать и записывать потоки.

Практические детали передачи символов с устройства и на устройство находятся в компетенции библиотеки ввода-вывода и драйверов устройств. В этой и следующей главах мы увидим, как работать с системой ввода-вывода, состоящей из потоков форматированных данных, с помощью стандартной библиотеки языка C++.

С точки зрения программиста, существует много разных видов ввода и вывода. Одна из возможных классификаций имеет следующий вид.

- Потоки (многих) единиц данных (как правило, связанных с файлами, сетевыми соединениями, записывающими устройствами или дисплеями).
- Взаимодействие с пользователем с помощью клавиатуры.
- Взаимодействие с пользователем с помощью графического интерфейса (вывод объектов, обработка щелчков мыши и т.д.).

Эта классификация не является единственно возможной, а различия между тремя видами ввода-вывода не столь отчетливы, как может показаться. Например, если выходной поток символов представляет собой HTTP-документ, предназначенный для браузера, то результат выглядит очень похожим на взаимодействие с пользователем с помощью графических элементов. И наоборот, результаты взаимодействия с помощью пользовательского графического интерфейса можно представить в программе

в виде последовательности символов. Однако такая классификация соответствует нашим средствам: первые две разновидности ввода-вывода обеспечиваются стандартными библиотечными потоками ввода-вывода и непосредственно поддерживаются большинством операционных систем. Начиная с главы 1 мы использовали библиотеку **iostream** и будем работать с ней в данной и следующей главах. Графический вывод и взаимодействие с пользователем посредством графического интерфейса обеспечиваются разнообразными библиотеками. Этот вид ввода-вывода мы рассмотрим в главах 12–16.

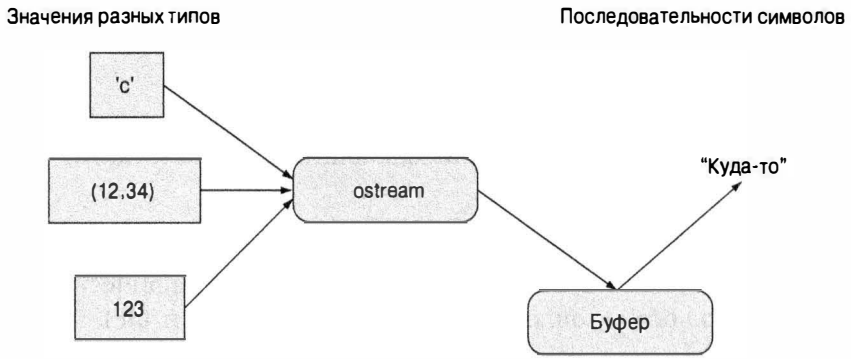
10.2. Модель потока ввода-вывода

Стандартная библиотека языка C++ содержит определение типа **istream** для потоков ввода и типа **ostream** — для потоков вывода. В наших программах мы использовали стандартный поток **istream** с именем **cin** и стандартный поток **ostream** с именем **cout**, так что мы уже знаем основы применения этой части стандартной библиотеки (которую часто называют библиотекой **iostream**).

Поток **ostream** делает следующее.

- Превращает значения разных типов в последовательности символов.
- Посылает эти символы "куда-то" (например, на консоль, в файл, основную память или на другой компьютер).

Поток **ostream** можно представить следующим образом.



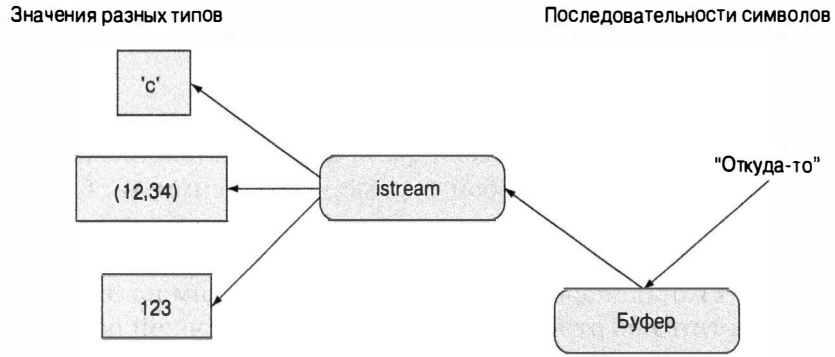
Буфер — это структура данных, которую поток **ostream** использует внутренне для хранения в ходе взаимодействия с операционной системой информации, полученной от вас. Задержка между записью в поток **ostream** и появлением символов в пункте назначения обычно объясняется тем, что эти символы находятся в буфере. Буферизация важна для производи-

тельности программы, а производительность программы важна при обработке больших объемов данных.

Поток **istream** делает следующее.

- Превращает последовательности символов в значения разных типов.
- Получает эти символы "откуда-то" (например, с консоли, из файла, из основной памяти или от другого компьютера).

Поток **istream** можно представить следующим образом.

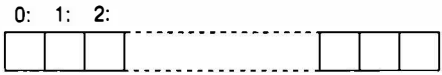


Как и поток **ostream**, для взаимодействия с операционной системой поток **istream** использует буфер. При этом буферизация может оказаться визуально заметной для пользователя. Когда вы используете поток **istream**, связанный с клавиатурой, все, что вы введете, останется в буфере, пока вы не нажмете клавишу <Enter> (возврат каретки/переход на новую строку), так что если вы передумали, то, пока не нажали клавишу <Enter>, можете стереть символы с помощью клавиши <Backspace>.

Одно из основных применений вывода — организация данных для чтения, доступного людям. Вспомните о сообщениях электронной почты, академических статьях, веб-страницах, счетах, деловых отчетах, списках контактов, оглавлениях, показаниях датчиков состояния устройств и т.д. Потоки **ostream** предоставляют много возможностей для форматирования текста по вкусу пользователей. Аналогично большая часть входной информации записывается людьми или форматируется так, чтоб люди могли ее прочитать. Потоки **istream** обеспечивают возможности для чтения данных, созданных потоками **ostream**. Вопросы, связанные с форматированием, будут рассмотрены в разделе 11.2, а ввод информации, отличной от символов, — в разделе 11.3.2. В основном сложность, связанная с вводом данных, обусловлена обработкой ошибок. Для того чтобы привести более реалистичные примеры, начнем с обсуждения того, как модель потоков ввода-вывода связана с файлами данных.

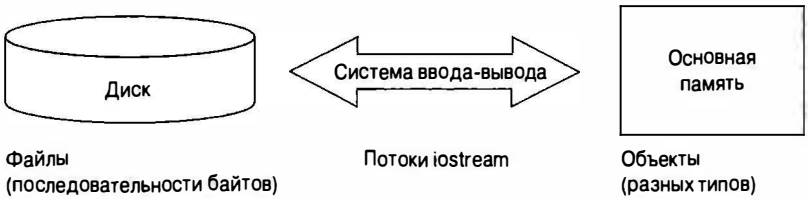
10.3. Файлы

Обычно у нас имеется намного больше данных, чем способна вместить основная память нашего компьютера, поэтому большая часть информации хранится на дисках или других средствах хранения данных высокой емкости. Такие устройства также предотвращают исчезновение данных при выключении компьютера — такие данные являются персистентными. На самом нижнем уровне файл просто представляет собой последовательность байтов, пронумерованных начиная с нуля.



Файл имеет формат; иначе говоря, он имеет набор правил, определяющих смысл байтов. Например, если файл является текстовым, то первые четыре байта представляют собой первые четыре символа. С другой стороны, если файл хранит бинарное представление целых чисел, то первые четыре байта используются для бинарного представления первого целого числа (раздел 11.3.2). Формат по отношению к файлам на диске играет ту же роль, что и типы по отношению к объектам в основной памяти. Мы можем приписать битам, записанным в файле, определенный смысл тогда и только тогда, когда известен его формат (разделы 11.2 и 11.3).

При работе с файлами поток **`ostream`** преобразует объекты, хранящиеся в основной памяти, в потоки байтов и записывает их на диск. Поток **`istream`** действует наоборот; иначе говоря, он считывает поток байтов с диска и составляет из них объект.



Чаще всего мы предполагаем, что байты на диске являются символами из обычного набора символов. Это не всегда так, но, поскольку другие представления обработать несложно, мы, как правило, будем придерживаться этого предположения. Кроме того, будем считать, что все файлы находятся на дисках (т.е. на вращающихся магнитных устройствах хранения данных). И опять-таки, это не всегда так (вспомните о флэш-памяти), но на данном уровне программирования фактическое устройство хранения не имеет значения. Это одно из главных преимуществ абстракций файла и потока.

Для того чтобы прочитать файл, мы должны

1. знать его имя;
2. открыть его (для чтения);
3. считать символы;
4. закрыть файл (хотя это обычно выполняется неявно).

Для того чтобы записать файл, мы должны

1. именовать его;
2. открыть файл (для записи) или создать новый файл с таким именем;
3. записать наши объекты;
4. закрыть файл (хотя это обычно выполняется неявно).

Мы уже знаем основы чтения и записи, поскольку во всех рассмотренных нами ситуациях поток **ostream**, связанный с файлом, ведет себя точно так же, как поток **cout**, а поток **istream**, связанный с файлом, ведет себя точно так же, как объект **cin**. Операции, характерные только для файлов, мы рассмотрим позднее (в разделе 11.3.3), а пока посмотрим, как открыть файлы, и сосредоточим свое внимание на операциях и приемах, которые можно применить ко всем потокам **ostream** и **istream**.

10.4. Открытие файла

Если вы хотите считать данные из файла или записать их в файл, откройте поток конкретно для этого файла. Поток **ifstream** — это поток **istream** для чтения из файла, поток **ofstream** — это поток **ostream** для записи в файл, а поток **fstream** представляет собой поток **iostream**, который можно использовать как для чтения, так и для записи. Перед использованием файловый поток должен быть связан с файлом, например:

```
cout << "Пожалуйста, введите имя файла: ";
string iname;
cin >> iname;
ifstream ist(iname);  // ist - входной поток для файла iname
if (!ist) error("Невозможно открыть входной файл ",name);
```

Определение потока **ifstream** с именем, заданным строкой **name**, открывает файл с этим именем для чтения. Проверка **!ist** позволяет выяснить, был ли файл открыт корректно. После этого можно считывать данные из файла точно так же, как из любого другого потока **istream**. Например, в предположении, что оператор ввода **>>** определен для типа **Point**, можно написать следующий фрагмент программы:

```
vector<Point> points;
for (Point p; ist>>p; )
    points.push_back(p);
```

Вывод в файлы выполняется аналогично, с помощью потоков **ofstream**. Рассмотрим пример.

```
cout << "Введите имя файла для вывода: ";
string oname;
cin >> oname;
ofstream ost {oname};  // ost — выходной поток для файла oname
if (!ost) error("Невозможно открыть выходной файл ",oname);
```

Определение потока **ofstream** с именем, заданным строкой **name**, открывает файл с этим именем для записи. Проверка **!ost** позволяет выяснить, был ли файл успешно открыт. После этого можно записывать данные в файл точно так же, как и в любой другой поток **ostream**, например:

```
for (int p : points)
    ost << '(' << p.x << ',' << p.y << ")\n";
```

Когда файловый поток выходит из области видимости, связанный с ним файл закрывается. Когда файл закрывается, связанный с ним буфер сбрасывается; иначе говоря, символы из буфера записываются в файл.

Как правило, файлы в программе лучше всего открывать как можно раньше, до выполнения каких-либо серьезных вычислений. В конце концов, было бы слишком расточительным выполнить большую часть работы и обнаружить, что вы не можете ее завершить, потому что вам некуда записать результаты.

В идеале открытие файла выполняется неявно как часть процесса создания потоков **ostream** и **istream**, а его закрытие основывается на области видимости потока, например:

```
void fill_from_file(vector<Point>& points, string& name)
{
    ifstream ist {name};   // Открытие файла для чтения
    if (!ist) error("невозможно открыть входной файл ",name);
    // ... использование ist ...
    // Неявное закрытие файла при выходе из функции
}
```

Операции **open()** и **close()** (раздел Б.7.1) могут быть выполнены и явно. Однако ориентация на область видимости минимизирует шансы, что кто-то попытается использовать файловый поток до того, как файл будет связан с потоком, или после его закрытия, например:

```
ifstream ifs;
// ...
ifs >> foo;          // Неудачно: нет связанного с потоком файла
// ...
ifs.open(name,       // Открываем файл name ...
    ios_base::in); // ... для чтения
// ...
ifs.close();         // Закрываем файл
```

```
// ...
ifs >> bar;        // Неудачно: связанный с потоком файл закрыт
// ...
```

В реальной программе возникающие проблемы, как правило, намного серьезнее. К счастью, мы не можем открыть файловый поток во второй раз, предварительно его не закрыв, например:

```
fstream fs;
fs.open("foo", ios_base::in) ; // Открываем файл для ввода
// Пропущена функция close()
fs.open("foo", ios_base::out); // Неудачно: поток ifs уже открыт
if (!fs) error("Невозможно");
```

Не забывайте проверять поток после его открытия.

Почему допускается явное использование функций `open()` и `close()`? Дело в том, что иногда время жизни соединения потока с файлом не совпадает с его областью видимости. Однако это событие происходит так редко, что о нем можно не беспокоиться. Следует также отметить, что такой код можно встретить у программистов, которые используют стили языков и библиотек, не имеющих идиомы области видимости, используемой потоками `iostream` (и остальной частью стандартной библиотеки C++).

Как будет показано в главе 11, о файлах можно сказать намного больше, но сейчас нам достаточно того, что их можно использовать как в качестве источников данных, так и для вывода информации. Это позволяет нам писать программы, которые были бы нереалистичными, если предположить, что всю входную информацию пользователь должен был бы вводить непосредственно с клавиатуры. С точки зрения программиста большое преимущество файла заключается в том, что мы можем многократно читать его в процессе отладки, пока программа не начнет работать правильно.

10.5. Чтение и запись файла

Посмотрим, как можно было бы считать результаты некоторых измерений из файла и представить их в памяти. Допустим, в файле записана температура воздуха, измеренная на метеостанции.

```
0 60.7
1 60.6
2 60.3
3 59.22
...
```

Этот файл содержит последовательность пар (час, температура). Часы пронумерованы от 0 до 23, а температура измерена по шкале Фаренгейта. Дальнейшее форматирование не предусмотрено; иначе говоря, файл не содержит никаких заголовков (например, информации о том, где снято

показание температуры), единиц измерений, знаков пунктуации (например, скобок вокруг каждой пары значений) или признака конца файла. Это простейший вариант.

Можно представить считываемую температуру в виде типа **Reading**.

```
struct Reading {         // Данные о температуре
    int    hour;         // Часы после полуночи [0,23]
    double temperature; // По Фаренгейту
};
```

При этом данные можно считывать следующим образом:

```
vector<Reading> temps; // Здесь хранится считанная информация
int hour;
double temperature;
while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) error("Некорректное время");
    temps.push_back(Reading{hour,temperature});
}
```

Это типичный цикл ввода. Поток **istream** с именем **ist** может быть входным файловым потоком (**ifstream**), как в предыдущем разделе, стандартным потоком ввода (**cin**) или любым другим потоком **istream**. Для кода, подобного приведенному выше, не имеет значения, откуда поток **istream** получает данные. Все, что требуется знать нашей программе, — это то, что поток **ist** представляет собой **istream** и что данные имеют ожидаемый формат. Следующий раздел посвящен интересному вопросу: как выявлять ошибки во входных данных и что можно сделать после выявления ошибки форматирования.

Записать данные в файл обычно проще, чем считать их оттуда. Как и ранее, после инициализации потока мы не обязаны знать, что именно он собой представляет. В частности, мы можем использовать выходной файловый поток (**ofstream**) из предыдущего раздела наравне с любым другим потоком **ostream**. При выводе мы, например, могли бы пожелать, чтобы каждая пара значений была заключена в скобки.

```
for(int i=0; i < temps.size(); ++i)
    ost << '(' << temps[i].hour << ','
        << temps[i].temperature << ")\n";
```

В результате мы получаем программу, которая читает исходные данные из файла и создает новый файл в формате (час, температура).

Поскольку файловые потоки автоматически закрывают свои файлы при выходе из области видимости, полная программа принимает следующий вид.

```
#include "std_lib_facilities.h"

struct Reading {         // Данные о температуре
```

```
    int    hour;        // Час после полуночи [0,23]
    double temperature; // По Фаренгейту
};

int main()
{
    cout << "Введите имя входного файла: ";
    string iname;
    cin >> iname;
    ifstream ist {iname};  // ist читает данные из файла iname
    if (!ist) error("Невозможно открыть входной файл ",iname);

    string oname;
    cout << "Введите имя выходного файла: ";
    cin >> oname;
    ofstream ost {oname};  // ost записывает данные в файл oname
    if (!ost) error("Невозможно открыть выходной файл ",oname);

    vector<Reading> temps; // Хранилище данных
    int hour;
    double temperature;
    while (ist >> hour >> temperature) {
        if (hour < 0 || 23 <hour) error("Некорректное время");
        temps.push_back(Reading{hour,temperature});
    }
    for (int i=0; i<temps.size(); ++i)
        ost << '(' << temps[i].hour << ','
            << temps[i].temperature << ")\n";
}
```

10.6. Обработка ошибок ввода-вывода

Вводя данные, мы должны предвидеть ошибки и обрабатывать их. Какими бывают ошибки? Как их обрабатывать? Ошибки возникают из-за того, что их совершают люди (неправильно поняли инструкцию, сделали опечатку, по клавиатуре прошлась кошка и т.д.), из-за того, что файлы не соответствуют спецификациям, из-за того, что программисты имеют неправильное представление об ожидаемых данных, и т.д. Возможности для совершения ошибок при вводе данных ничем не ограничены! Однако поток `istream` сводит их все к четырем возможным классам, которые называют *состояниями потока* (stream state).

Состояния потока	
`good()`	Операции выполнены успешно
`eof()`	Достигнут конец ввода (конец файла)
`fail()`	Произошло нечто неожиданное (например, мы ждем цифру, а получаем букву)
`bad()`	Произошло нечто неожиданное и серьезное (например, ошибка чтения диска)

К сожалению, различия между состояниями **fail()** и **bad()** определены неточно и зависят от точки зрения программистов, создающих операции ввода-вывода для новых типов. Однако основная идея проста: если операция ввода обнаруживает простую ошибку форматирования, она переводит поток в состояние **fail()**, предполагая, что вы (пользователь операции ввода) способны ее исправить. Если же произошло нечто совершенно ужасное, например неправильное чтение с диска, то операция ввода переводит поток в состояние **bad()**, предполагая, что вам ничего не остается делать, кроме как отказаться от попыток считать данные из потока. Поток в состоянии **bad()** находится также и в состоянии **fail()**. Это приводит нас к следующей общей логике:

```
int i = 0;
cin >> i;
if (!cin) { // Мы окажемся здесь (только) при сбое ввода
    if (cin.bad()) error("cin испорчен"); // Поток поврежден!
    if (cin.eof()) {
        // Входных данных больше нет
        // Именно так обычно завершается ввод данных
    }
    if (cin.fail()) { // С потоком случилась неприятность
        cin.clear();  // Приготовиться к дальнейшему вводу
        // Исправление ситуации
    }
}
```

Выражение **!cin** можно прочитать как "поток **cin** в плохом состоянии" или "с потоком **cin** что-то случилось", или "поток **cin** не находится в состоянии **good()**". Это выражение противоположно по смыслу выражению "операция успешно завершена". Обратите внимание на инструкцию **cin.clear()** при обработке состояния **fail()**. Если поток находится в состоянии **fail()**, мы, вероятно, сможем его восстановить. Для этого надо явно вывести поток из состояния **fail()**, и можно продолжить просматривать последовательность символов, находящихся в этом потоке; функция **clear()** гарантирует, что после выполнения вызова **cin.clear()** поток **cin** перейдет в состояние **good()**.

Рассмотрим пример использования состояния потока. Давайте подумаем, каким образом считать последовательность целых чисел, которые могут завершаться символом * или признаком конца файла (<Ctrl+Z> в Windows или <Ctrl+D> в Unix) в вектор:

```
1 2 3 4 5 *
```

Ввести их можно с помощью функции наподобие следующей:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
    // Чтение целых чисел из ist в v до достижения
    // конца файла или символа завершения
```

```
{
    for (int i; ist >> i; ) v.push_back(i);
    if (ist.eof()) return;              // Отлично: конец файла

    if (ist.bad()) error("ist поврежден"); // Поток поврежден!
    if (ist.fail()) { // Сброс состояния и сообщение об ошибке
        ist.clear();  // Сброс состояния потока, чтобы можно
                      // было получить признак завершения
        char c;
        ist >> c;     // Чтение символа (если повезет -
                      // символа завершения)
        if (c != terminator) {    // Неожиданный символ
            ist.unget();          // Вернуть его в поток
            ist.clear(ios_base::failbit); // Вернуть состояние
        }                                 // fail()
    }
}
```

Обратите внимание на то, что, даже если мы не находим признак конца файла, мы все равно выходим из функции. В конце концов, мы могли собрать какие-то данные, а функция, вызвавшая функцию `fill_vector()`, может восстановить поток из состояния `fail()`. Поскольку мы сначала сбросили состояние потока, чтобы проверить символ завершения, то, если мы не обнаруживаем таковой, мы должны вернуть поток обратно в состояние `fail()`. Для этого выполняется вызов `ist.clear(ios_base::failbit)`. Обратите внимание на потенциально опасное использование функции `clear()`: на самом деле вызов функции `clear()` с аргументом *устанавливает* переданные в нем флаги (биты) состояния потока `iostream`, сбрасывая (только) не указанные. Переводя поток в состояние `fail()`, мы указываем, что обнаружили ошибку форматирования, а не нечто более серьезное. Мы возвращаем символ в поток `ist`, используя функцию `unget()`; функция, вызывающая функцию `fill_vector()`, может использовать его по своему усмотрению. Функция `unget()` представляет собой более короткий вариант функции `putback()` (разделы 6.8.2, Б.7.3), который основывается на предположении, что поток помнит, какой символ был последним, и поэтому его не обязательно указывать явно.

Если вы вызвали функцию `fill_vector()` и хотите знать, что вызвало прекращение ввода, можете проверить состояния потока `fail()` и `eof()`. Кроме того, можно перехватить исключение `runtime_error`, сгенерированное функцией `error()`, но понятно, что вряд ли вы что-то получите из потока `istream` в состоянии `bad()`. Большинство вызывающих функций не

предусматривает сложной обработки ошибок. По этой причине практически во всех случаях единственное, что мы можем сделать, обнаружив состояние `bad()`, — это сгенерировать исключение. Для того чтобы облегчить себе жизнь, мы можем поручить потоку `istream` сделать это вместо нас.

```
// Теперь поток ist генерирует исключение,
// если попадает в состояние bad
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

Эти обозначения могут показаться странными, но результат простой: если поток **ist** окажется в состоянии **bad()**, он сгенерирует стандартное библиотечное исключение **ios_base::failure**. Вызвать функцию **exceptions()** необходимо только один раз. Все это позволяет упростить циклы ввода, игнорируя состояние **bad()**:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
    // Чтение целых чисел из ist в v до достижения
    // конца файла или символа завершения
{
    for (int i; ist >> i; ) v.push_back(i);
    if (ist.eof()) return;              // Отлично: конец файла

    // не good(), не bad(), не eof(); значит, fail()
    ist.clear();  // Сброс состояния потока, чтобы можно
                  // было получить признак завершения
    char c;
    ist >> c;     // Чтение символа (если повезет -
                  // символа завершения)
    if (c != terminator) {    // Неожиданный символ
        ist.unget();          // Вернуть его в поток
        ist.clear(ios_base::failbit); // Вернуть состояние
    }                                 // fail()
}
```

Класс **ios_base**, который появляется здесь, представляет собой часть потока **iostream**, где хранятся константы, такие как **badbit**, исключения, такие как **failure**, и другие полезные вещи. Для обращения к нему необходим оператор **::**, например **ios_base::badbit** (раздел Б.7.2). Мы не планируем подробно описывать библиотеку **iostream**; для этого понадобился бы отдельный курс лекций. Например, потоки **iostream** могут обрабатывать разные наборы символов, реализовывать разные стратегии буферизации, а также содержат средства форматирования представлений денежных средств на разных языках (однажды мы даже получили сообщение об ошибке, связанной с форматированием представления украинской валюты). Все, что вам необходимо знать о потоках **iostream**, можно найти в книгах *The C++ Programming Language* Страуструпа (Stroustrup) и *Standard C++ IOStreams and Locales* Лангера (Langer).

Поток **ostream** можно тестировать на наличие тех же состояний, что и у потока **istream**: **good()**, **fail()**, **eof()** и **bad()**. Однако в программах, которые мы описываем в этой книге, ошибки при выводе встречаются намного реже, чем при вводе, поэтому мы редко их проверяем. Если существенна вероятность того, что устройство вывода недоступно, переполнено или сломано, в программе следует предусмотреть проверку состояния потока

вывода после каждой операции вывода, так, как мы делали выше по отношению к операции ввода.

10.7. Считывание отдельного значения

Итак, мы знаем, как считать последовательность значений, завершающихся признаком конца файла или символом завершения ввода. Впоследствии мы рассмотрим еще несколько примеров, а сейчас обсудим популярную идею о том, чтобы запрашивать значение несколько раз, пока не будет введен его приемлемый вариант. Это позволит нам изучить несколько распространенных проектных решений. Мы обсудим эти альтернативы на примерах нескольких решений простой проблемы — как получить от пользователя приемлемое значение. Начнем с первой попытки решения — очевидного, но запутанного, а затем станем его постепенно совершенствовать. Наше основное предположение заключается в том, что мы имеем дело с интерактивным вводом, в ходе которого человек набирает на клавиатуре входные данные и читает сообщения, поступающие от программы. Давайте предложим пользователю ввести целое число от 1 до 10 (включительно).

```
cout << "Введите целое число от 1 до 10:\n";
int n = 0;
while (cin>>n) {                    // Читаем
    if (1<=n && n<=10) break;       // Проверяем диапазон
    cout << "Извините, " << n
         << " выходит за пределы [1,10]; попробуйте еще\n";
}
// ... используем полученное значение n ...
```

Этот код довольно уродлив, но почти работоспособен. Если вы не любите использовать оператор **break** (раздел А.6), то можете объединить считывание и проверку диапазона.

```
cout << "Введите целое число от 1 до 10:\n";
int n = 0;
while(cin>>n && !(1<=n&&n<=10)) // Читаем и проверяем диапазон
    cout << "Извините, " << n
         << " выходит за пределы [1,10]; попробуйте еще\n";
// ... используем полученное значение n ...
```

Но это изменения, которые носят лишь "косметический" характер. И почему мы утверждаем, что этот код работоспособен "почти"? Дело в том, что он будет работать, если пользователь аккуратно вводит целые числа. Если же пользователь небрежно наберет букву **t** вместо цифры **6** (на большинстве клавиатур буква **t** расположена прямо под цифрой **6**), то программа выйдет из цикла, не изменив значения переменной **n**, поэтому это число окажется за пределами допустимого диапазона. Такой код нельзя назвать качественным. Шутник (или усердный тести-

ровщик) может также ввести с клавиатуры признак конца файла (нажав комбинацию клавиш <Ctrl+Z> в системе Windows или <Ctrl+D> в системе Unix). И программа снова выйдет из цикла со значением `n`, лежащим за пределами допустимого диапазона. Иначе говоря, для того чтобы обеспечить надежный ввод, мы должны решить три проблемы.

1. Что делать, если пользователь вводит число, находящееся за пределами допустимого диапазона?
2. Что делать, если пользователь не вводит никакого числа (признак конца файла)?
3. Что делать, если пользователь вводит неправильные данные (в данном случае — нечто, отличное от целого числа)?

Что же делать во всех этих ситуациях? При создании программ часто возникает вопрос "Чего мы хотим на самом деле?" В данном случае для каждой из трех ошибок у нас есть три альтернативы.

1. Решить проблему в коде при вводе данных.
2. Сгенерировать исключение, чтобы проблему решил кто-то другой (возможно, прекратив выполнение программы).
3. Игнорировать проблему.

Это очень распространенные при обработке ошибок альтернативы, а наш пример — хороший пример рассуждения о том, что делать с ошибками.

Заманчиво сказать, что третья альтернатива, т.е. игнорировать проблему, не является приемлемой ни в коем случае, но это было бы преувеличением. Если я пишу простую программу для собственного использования, то могу делать все, что захочу, даже забыть о проверке ошибок, которые могут привести к ужасным результатам. Однако если я пишу программу, которую буду использовать через несколько часов после ее создания, то было бы глупо оставлять такие ошибки. Если же я планирую передавать свою программу другим людям, то не стану оставлять такие дыры в системе проверки ошибок. Пожалуйста, обратите внимание на то, что местоимение "я" здесь использовано намеренно; местоимение "мы" могло бы ввести в заблуждение. Мы не считаем третью альтернативу приемлемой, даже если в проекте участвуют только два человека.

Выбор между первой и второй альтернативами является настоящим, а не формальным; иначе говоря, в программе могут быть веские причины выбрать любую из них. Сначала отметим, что в большинстве программ нет локального и элегантного способа обработать ситуацию, когда пользователь не ввел данные, сидя за клавиатурой: после того как поток ввода был закрыт, нет большого смысла предлагать пользователю ввести число. Мы могли бы заново открыть поток `cin` (используя функцию `cin.clear()`),

но пользователь вряд ли закрыл этот поток непреднамеренно (как можно случайно нажать комбинацию клавиш <Ctrl+Z>?). Если программа ждет ввода целого числа, а обнаруживает конец файла, то часть программы, пытающаяся прочитать это число, должна прекратить свои попытки и надеяться, что какая-то другая часть программы справится с этой проблемой; иначе говоря, наш код, требующий ввода от пользователя, должен сгенерировать исключение. Это значит, что происходит не выбор между генерацией исключений и локальным решением проблемы, а выбор проблем (если таковые имеют место), которые следует обрабатывать локально.

10.7.1. Разделение задачи на управляемые части

Попробуем локально решать как проблемы, связанные с выходом за пределы допустимого диапазона, так и проблемы ввода данных некорректного типа.

```
cout << "Введите целое число от 1 до 10:\n";
int n = 0;

while (true)
{
    cin >> n;
    if (cin) {    // Введено целое число; проверяем его
        if (1 <= n && n <= 10) break;
        cout << "Извините, " << n
             << "выходит за пределы [1,10]; попробуйте еще\n";
    }
    else if(cin.fail()){ // Обнаружено нечто, что не является
                         // целым числом
        cin.clear();     // Возвращаем поток в состояние good();
                         // мы хотим взглянуть на символы
        cout << "Это не число; попробуйте еще раз\n";
        for(char ch; cin>>ch && !isdigit(ch);) // Отбрасываем
            /* Ничего не делаем */;            // не цифры
        if (!cin) error("Ввода нет");   // цифры не обнаружены
        cin.unget();                   // возвращаем цифру, чтобы
                                       // можно было считать число
    } else {
        error("Ввода нет"); // Состояние eof или bad:
    }                       // прекращаем работу
}
// если мы добрались до этой точки, значит, число n лежит
// в диапазоне [1,10]
```

Этот код запутан и многословен. На самом деле мы бы не рекомендовали людям писать такие программы всякий раз, когда они ждут от пользователя ввода целого числа. С другой стороны, мы должны предусматривать потенциальные ошибки, поскольку людям свойственно

ошибаться. Так что же делать? Причина запутанности кода в том, что в нем перемешано сразу несколько задач.

- Считывание значения.
- Предложение ввести значение.
- Вывод сообщений об ошибках.
- Пропуск "плохих" входных символов.
- Проверка диапазона входных чисел.

Для того чтобы сделать код яснее, часто достаточно просто логически разделить задачи среди нескольких функций. Например, мы можем выделить код, восстанавливающий ввод после обнаружения "плохого" (т.е. неожиданного) символа.

```
void skip_to_int()
{
    if (cin.fail()) { // Обнаружено нечто, не являющееся
                      // целым числом
        cin.clear();  // Возвращаем поток в состояние good();
        for(char ch; cin>>ch;) { // Отбрасываем не цифры
            if (isdigit(ch) || ch == '-') {
                cin.unget();  // Возвращаем цифру в поток, чтобы
                              // можно было считать число
                return;
            }
        }
    }
    error("Ввода нет");          // Состояние eof или bad
}
```

Имея такую вспомогательную функцию `skip_to_int()`, можно написать следующий код:

```
cout << "Введите целое число от 1 до 10:\n";
int n = 0;
while (true) {
    if (cin>>n) { // Введено целое число; проверяем его
        if (1<=n && n<=10) break;
        cout << "Извините, " << n
             << "выходит за пределы [1,10]; попробуйте еще\n";
    }
    else {
        cout << "Извините, это не число; попробуйте еще раз\n";
        skip_to_int();
    }
}
// Если мы добрались до этой точки, значит, число n лежит
// в диапазоне [1,10]
```

Этот код лучше, но все равно остается слишком длинным и запутанным для того, чтобы много раз применять его в программе.

Какую операцию мы бы хотели иметь на самом деле? Один из разумных ответов звучит так: "Нам нужны две функции: одна должна считывать любое число типа `int`, а другая — целое число из заданного диапазона".

```
int get_int();                    // Считывает из cin число int

int get_int(int low, int high); // Считывает из cin число int
                                  // в диапазоне [low,high]
```

Если бы у нас были эти функции, то по крайней мере мы могли бы использовать их просто и правильно. Их несложно написать.

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Это не число; попробуйте еще раз\n";
        skip_to_int();
    }
}
```

В основном функция `get_int()` упорно считывает данные, пока не найдет цифры, которые можно интерпретировать как целое число. Чтобы прекратить работу функции `get_int()`, следует ввести целое число или признак конца файла (во втором случае функция `get_int()` сгенерирует исключение).

Используя такую общую функцию `get_int()`, можно написать проверку выхода за пределы диапазона `get_int()`:

```
int get_int(int low, int high)
{
    cout << "Введите целое число из от "
         << low << " до " << high << " включительно:\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << "Извините, " << n
             << "выходит за пределы интервала ["<< low
             << ':' << high << "]; попробуйте еще\n";
    }
}
```

Этот вариант функции `get_int()` работает так же упорно, как и остальные. Функция продолжает получать из `cin` целые числа, пока не получит число, лежащее в указанных пределах.

Теперь можно написать код для ввода целых чисел наподобие следующего.

```
int n = get_int(1,10);
cout << "n: " << n << endl;

int m = get_int(2,300);
cout << "m: " << m << endl;
```

Не забудьте только предусмотреть перехват исключения, если не хотите получить сообщение об ошибке в (возможно, редкой) ситуации, когда функция `get_int()` на самом деле не может прочесть ни одного требуемого числа.

10.7.2. Отделение диалога от функции

Функции `get_int()` по-прежнему смешивают ввод данных с выводом сообщений, адресованных пользователю. Для простых программ это вполне допустимо, но в большой программе мы можем пожелать, чтобы сообщения были разными. Мы можем захотеть вызывать `get_int()` так, как показано в следующем фрагменте исходного текста:

```
int strength = get_int(1,10,"Введите силу",
                       "Вне диапазона, попробуйте еще");
cout << "Сила: " << strength << '\n';

int altitude = get_int(0,50000,
                       "Введите высоту в футах",
                       "Вне диапазона, повторите ввод");
cout << "Высота: " << altitude << " футов над уровнем моря\n";
```

Реализовать такую функцию можно так:

```
int get_int(int low, int high,
            const string& greeting,
            const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << sorry << ": [" << low << ':' << high << "]\n";
    }
}
```

Довольно трудно составлять сообщения из заготовок, поэтому необходимо выработать стиль сообщений. Часто это оказывается полезным и позволяет составлять действительно гибкие сообщения, необходимые для поддержки многих естественных языков (например, арабского, бенгальского, китайского, датского, английского и французского), но эта задача не для новичков.

Обратите внимание на то, что наше решение все еще не завершено: функция `get_int()` без указания диапазона все еще осталась "болтушкой".

Более тонкий аспект этой проблемы заключается в том, что вспомогательные функции, используемые в разных частях программы, не должны содержать "вшитых" сообщений. Далее, библиотечные функции, которые по своей сути предназначены для использования во многих программах, вообще не должны выдавать никаких сообщений пользователю, — в конце концов, автор библиотеки может даже не предполагать, что программа, в которой используется его библиотека, будет выполняться на машине под наблюдением со стороны человека. Это одна из причин, по которым наша функция `error()` не выводит никаких сообщений об ошибках (см. раздел 5.6.3); в общем случае мы не можем знать, куда их выводить.

10.8. Операторы вывода, определенные пользователем

Определение оператора вывода << для заданного типа, как правило, представляет собой тривиальную задачу. Основная проблема при его разработке заключается в том, что разные люди могут предпочитать разные представления результатов, поэтому трудно прийти к общему соглашению о каком-то едином формате. Однако даже при отсутствии единого формата, который удовлетворял бы всех пользователей, часто целесообразно предусмотреть оператор << для пользовательского типа. В ходе отладки и на первых этапах проектирования нам нужна хотя бы тривиальная запись объектов указанного типа. Позднее нам может понадобиться более сложный оператор вывода <<, позволяющий пользователю получать форматированную информацию. Кроме того, если желаемое представление выходной информации отличается от стандартного представления, обеспечиваемого оператором <<, мы можем просто обойти этот оператор и записывать отдельные части объектов пользовательского типа так, как мы хотим.

Рассмотрим простой оператор вывода для типа `Date` из раздела 9.8, который просто печатает год, месяц и день, разделенные запятыми.

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
              << ',' << int(d.month())
              << ',' << d.day() << ')';
}
```

Таким образом, дата "30 августа 2004 года" будет представлена как `(2004,8,30)`. Такое простое представление элементов в виде списка типично для типов, содержащих небольшое количество членов, хотя мы могли бы реализовать более сложную идею или точнее учесть специфические потребности.

В разделе 9.6 мы упоминали о том, что оператор, определенный пользователем, выполняется с помощью вызова соответствующей функции. Рассмотрим пример. Если в программе определен оператор вывода << для типа **Date**, то инструкция

```
cout << d1;
```

где объект **d1** имеет тип **Date**, представляет собой вызов функции

```
operator<<(cout,d1);
```

Обратите внимание на то, что первый аргумент **ostream&** функции **operator<<()** одновременно является ее возвращаемым значением. Это позволяет создавать "цепочки" операторов вывода. Например, мы могли бы вывести сразу две даты.

```
cout << d1 << d2;
```

В этом случае сначала был бы выполнен первый оператор <<, а затем второй.

```
cout << d1 << d2; // Означает operator<<(cout,d1) << d2;
                  // или operator<<(operator<<(cout,d1),d2);
```

Иначе говоря, сначала происходит первый вывод объекта **d1** в поток **cout**, а затем вывод объекта **d2** в поток вывода, являющийся результатом выполнения первого оператора. Фактически мы можем использовать любой из указанных трех вариантов вывода объектов **d1** и **d2**. Однако один из этих вариантов намного проще остальных.

10.9. Операторы ввода, определенные пользователем

Определение оператора ввода >> для заданного типа и формат ввода обычно тесно связаны с обработкой ошибок. Следовательно, эта задача может оказаться довольной сложной.

Рассмотрим простой оператор ввода для типа **Date** из раздела 9.8, который считывает даты, записанные ранее с помощью оператора <<, определенного выше.

```
istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') {
        // Ошибка формата
        is.clear(ios_base::failbit);
        return is;
    }
```

```
    dd = Date{y,Date::Month(m),d}; // Обновляем объект dd
    return is;
}
```

Этот оператор >> считывает такие тройки, как (2004,8,20), и пытается создать объект типа Date из полученных трех чисел. Как правило, выполнить ввод данных намного труднее, чем их вывод. Просто при вводе данных намного больше возможностей для появления ошибок, чем при выводе.

Если данный оператор >> не находит трех чисел, заданных в формате (*целое*, *целое*, *целое*), то поток ввода перейдет в одно из состояний fail, eof или bad, а целевой объект типа Date останется неизмененным. Для установки состояния потока istream используется функция-член clear(). Очевидно, что флаг ios_base::failbit переводит поток в состояние fail(). В идеале при сбое во время чтения следует оставлять объект класса Date без изменений; это дает более ясный код. В идеале также хотелось бы, чтобы функция operator>>() отбрасывала любые символы, которые она не использует, но в данном случае это было бы слишком трудно сделать: мы должны были бы прочитать слишком много символов, пока не обнаружится ошибка формата. В качестве примера рассмотрим тройку (2004, 8, 30}. Только когда мы увидим закрывающую фигурную скобку, }, обнаружится ошибка формата, и нам потребуется вернуть в поток слишком много символов, в то время как функция unget() позволяет вернуть только один символ. Если функция operator>>() считывает неправильный объект класса Date, например (2004,8,32), конструктор класса Date генерирует исключение, которое приводит к прекращению выполнения оператора operator>>().

10.10. Стандартный цикл ввода

В разделе 10.5 мы видели, как считываются и записываются файлы. Однако тогда мы еще не рассматривали обработку ошибок (см. раздел 10.6) и считали, что файл считывается от начала до конца. Это разумное предположение, поскольку корректность файла мы часто проверяем отдельно. Тем не менее мы зачастую хотим выполнять проверку считанных данных в ходе их ввода. Рассмотрим общую стратегию, предполагая, что объект ist представляет собой поток istream.

```
for(My_type var; ist>>var; ) { // Читаем до конца файла
    // Здесь можно было бы проверить корректность var

    // Здесь мы что-то делаем с переменной var
}
// Восстановиться из состояния bad удается довольно редко;
// не делайте этого без крайней необходимости.
if (ist.bad()) error("Плохой поток ввода");
```

```
if (ist.fail()) {
    // Принят ли сигнал окончания ввода?
}
// Если мы здесь, обнаружен конец файла
```

Иначе говоря, мы считываем последовательность значений, записывая их в переменные, а когда не можем больше считать ни одного значения, проверяем состояние потока, чтобы понять, что случилось. Как и в разделе 10.6, эту стратегию можно усовершенствовать, заставив поток **istream** генерировать исключение типа **failure** в случае сбоя. Это позволит нам не выполнять проверку постоянно.

```
// Где-то: пусть поток ist генерирует исключение при сбое
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

Можно также назначить признаком завершения ввода какой-нибудь символ.

```
for(My_type var; ist>>var; ) { // Читаем до конца файла
    // Здесь можно было бы проверить корректность var

    // Здесь мы что-то делаем с переменной var
}
if (ist.fail()) { // Символ '|' – конец ввода и/или разделитель
    ist.clear();
    char ch;
    if (!(ist >> ch && ch == '|'))
        error("Неправильное завершение ввода");

}
// Если мы здесь, обнаружен конец файла или признак завершения
```

Если вы не хотите использовать в качестве признака завершения ввода какой-то символ, т.е. хотите ограничиться только признаком конца файла, удалите проверку перед вызовом функции **error()**. Однако признаки завершения чтения оказываются очень полезны, когда считываются файлы с вложенными конструкциями, например файлы с помесячной информацией, содержащей ежедневную информацию, включающую почасовую информацию, и т.д. В таких ситуациях стоит подумать о символе завершения ввода.

К сожалению, этот код остается довольно запутанным. В частности, слишком утомительно при считывании многих файлов каждый раз повторять проверку символа завершения ввода. Для решения этой проблемы следует написать отдельную функцию.

```
// Где-то: пусть поток ist генерирует исключение при сбое
ist.exceptions(ist.exceptions()|ios_base::badbit);

void end_of_loop(istream& ist, char term, const string& message)
{
```

```
    if (ist.fail()) { // Используем term как символ
                      // завершения ввода и/или разделитель
        ist.clear();
        char ch;
        if (ist >> ch && ch == term) return; // Все хорошо
        error(message);
    }
}
```

Это позволяет нам сократить цикл ввода до

```
for(My_type var; ist>>var; ) { // Читаем до конца файла
    // Здесь можно было бы проверить корректность var

    // Здесь мы что-то делаем с переменной var
}

// Проверяем, можно ли продолжать
end_of_loop(ist,'|',"неправильное завершение файла");
// Продолжаем: обнаружен конец файла или признак завершения ввода
```

Функция `end_of_loop()` не выполняет никаких действий, кроме проверки, находится ли поток в состоянии `fail()`. Мы считаем, что эту достаточно простую и универсальную функцию можно использовать для разных целей.

10.11. Чтение структурированного файла

Попробуем применить этот "стандартный цикл" в конкретном примере. Как обычно, используем этот пример для иллюстрации широко распространенных методов проектирования и программирования. Предположим, в файле записаны результаты измерения температуры, имеющие определенную структуру.

- В файле записаны годы, в течение которых производились измерения.
  - Запись о годе начинается символами `{ year`, за которыми следует целое число, обозначающее год, например `1900`, и заканчивается символом `}`.
- Год состоит из месяцев, в течение которых производились измерения.
  - Запись о месяце начинается символами `{ month`, за которыми следует трехбуквенное название месяца, например `jan`, и заканчивается символом `}`.
- Данные содержат показания времени и температуры.
  - Показания начинаются с символа `(`, за которыми следует день месяца, час дня и температура, и заканчиваются символом `)`.

Рассмотрим пример.

```
{ year 1990 }
{year 1991 { month jun }}
{year 1992 {month jan (1 0 61.5)} {month feb (1 1 64) (2 2 65.2)}}
{year 2000
    { month feb (1 1 68 ) (2 3 66.66 ) ( 1 0 67.2)}
    {month dec (15 15 - 9.2 ) (15 14 - 8.8) (14 0 - 2) }
}
```

Этот формат довольно своеобразен. Форматы записи файлов вообще часто оказываются довольно специфическими. В целом в программной индустрии наблюдается тенденция к широкому использованию все более упорядоченных и иерархически структурированных файлов (например, HTML и XML), но в действительности мы по-прежнему редко можем управлять форматом файла, который необходимо прочитать. Файлы таковы, каковы они есть, и нам нужно их прочитать. Если формат слишком неудачен или файлы содержат много ошибок, можно написать программу преобразования формата в более подходящий. С другой стороны, мы, как правило, имеем возможность выбирать представление данных в памяти в удобном для себя виде, а при выборе формата вывода часто руководствуемся лишь собственными потребностями и вкусом.

Предположим, данные о температуре записаны в указанном выше формате и нам нужно их прочитать. К счастью, формат содержит автоматически идентифицируемые компоненты, такие как годы и месяцы (немного напоминает форматы HTML и XML). С другой стороны, формат отдельной записи довольно неудобен. Например, в ней нет информации, которая могла бы нам помочь, если бы кто-то перепутал день месяца с часом или представил температуру по шкале Цельсия, хотя нужно было по шкале Фаренгейта, и наоборот. Нам просто придется справиться с тем, что мы имеем.

10.11.1. Представление в памяти

Как представить эти данные в памяти? На первый взгляд, необходимо создать три класса, **Year**, **Month** и **Reading**, точно соответствующие входной информации. Классы **Year** и **Month** очевидным образом могли бы оказаться полезными при обработке данных; мы хотим сравнивать температуры разных лет, вычислять среднемесячные температуры, сравнивать разные месяцы одного года, одинаковые месяцы разных лет, показания температуры с записями о солнечном излучении и влажности и т.д. В принципе, классы **Year** и **Month** точно отображают наши представления о температуре и погоде: класс **Month** содержит ежемесячную информацию, а класс **Year** — ежегодную. А как насчет класса **Reading**? Это понятие низкого уровня, связанное с частью аппаратного обеспечения (сенсором). Данные в классе **Reading** (день месяца, час и температура) имеют смысл только в рамках

класса **Month**. Кроме того, они не структурированы: никто не обещал, что данные будут записаны по дням или по часам. В общем случае для того, чтобы сделать с данными что-то полезное, сначала их необходимо упорядочить.

Для представления данных о температуре в памяти сделаем следующие предположения.

- Если есть показания для какого-то месяца, то их обычно бывает много.
- Если есть показания для какого-то дня, то их тоже обычно бывает много.

В этом случае целесообразно представить класс **Year** как вектор, состоящий из 12 объектов класса **Month**, класс **Month** — как вектор, состоящий из 30 объектов класса **Day**, а класс **Day** — как 24 показания температуры (по одному в час). Это позволяет просто и легко манипулировать данными при решении самых разных задач. Итак, классы **Day**, **Month** и **Year** — это простые структуры данных, каждая из которых имеет конструктор. Поскольку мы планируем создавать объекты классов **Month** и **Day** как часть объектов класса **Year** еще до того, как узнаем, какие показания температуры у нас есть, то должны сформулировать, что означает "пропущены данные" для часа дня, до считывания которых еще не подошла очередь.

```
const int not_a_reading = -7777; // Ниже абсолютного нуля
```

Аналогично мы заметили, что часто в течение некоторых месяцев не производилось ни одного измерения, поэтому ввели понятие "пропущенный месяц", вместо того чтобы проверять пропуски для каждого дня.

```
const int not_a_month = -1;
```

Три основных класса принимают следующий вид:

```
struct Day {
    vector<double> hour {vector<double>(24,not_a_reading)};
};
```

Иначе говоря, **Day** включает 24 часа, каждый из которых инициализирован значением **not_a_reading**.

```
struct Month {                  // Месяц
    int month {not_a_month}; // [0,11] (январю соответствует 0)
    vector<Day> day {32};       // [1,31] по одному вектору на день
};
```

Мы решили пожертвовать одним элементом **data[0]**, чтобы упростить код.

```
struct Year {                   // Год состоит из месяцев
    int year;                   // Положительное значение
    vector<Month> month{12}; // [0,11] (январю соответствует 0)
};
```

В принципе, каждый класс — это просто вектор, а классы `Month` и `Year` содержат идентифицирующие члены `month` и `year` соответственно.

В этом примере существует несколько "магических констант" (например, `24`, `32` и `12`). Как правило, мы пытаемся избегать таких литеральных констант в коде. Данные константы носят фундаментальный характер (количество месяцев в году изменяется редко) и в остальной части кода не используются. Однако мы оставили их в коде, в первую очередь, для того, чтобы напомнить вам о проблеме "магических констант"; символические константы использовать намного предпочтительнее (см. раздел 7.6.1). Применение числа `32` для количества дней в месяце, определенно, требует объяснений; в этом случае число `32` действительно становится "магическим".

Вы можете спросить, почему мы не написали

```
struct Day {
    vector<double> hour {24,not_a_reading};
};
```

Да, так было бы проще, но — увы! — мы получили бы при этом вектор с двумя элементами (`24` и `-1`). Если мы хотим указать количество элементов вектора, у которого значение типа `int` может быть преобразовано в тип элемента, мы должны использовать синтаксис инициализатора с круглыми скобками `()` (§18.2).

10.11.2. Чтение структурированных значений

Класс `Reading` будет использован только для ввода данных, и он еще проще остальных:

```
struct Reading {
    int day;
    int hour;
    double temperature;
};

istream& operator>>(istream& is, Reading& r)
    // Считываем показания температуры из потока is в r
    // Формат: ( 3 4 9.7 )
    // Проверяем формат, но не корректность данных
{
    char ch1;
    if (is >> ch1 && ch1 != '(' ){// Может ли это быть Reading?
        is.unget();
        is.clear(ios_base::failbit);
        return is;
    }

    char ch2;
```

```
    int d;
    int h;
    double t;
    is >> d >> h >> t >> ch2;
    if (!is || ch2 != ')')
        error("Плохая запись");    // Некорректная запись
    r.day = d;
    r.hour = h;
    r.temperature = t;
    return is;
}
```

Мы начинаем с проверки, правильно ли начинается формат. Если нет, то переводим файл в состояние `fail()` и выходим. Это позволяет нам попытаться считать информацию как-то иначе. С другой стороны, если ошибка формата обнаруживается после считывания данных и нет реальных шансов на возобновление работы, то мы вызываем функцию `error()`.

Операция ввода класса `Month` почти такая же, за исключением того, что в ней считывается произвольное количество объектов класса `Reading`, а не фиксированный набор значений (как делает оператор `>>` в классе `Reading`).

```
istream& operator>>(istream& is, Month& m)
    // Считываем объект класса Month из потока is в объект m
    // Формат: { month feb ... }
{
    char ch = 0;
    if (is >> ch && ch!='{') {
        is.unget();
        is.clear(ios_base::failbit); // Ошибка ввода Month
        return is;
    }

    string month_marker;
    string mm;
    is >> month_marker >> mm;
    if (!is || month_marker!="month")
        error("Неверное начало Month");
    m.month = month_to_int(mm);

    Reading r;
    int duplicates = 0;
    int invalids = 0;
    for(Reading r; is >> r; ) {
        if (is_valid(r)) {
            if (m.day[r.day].hour[r.hour] != not_a_reading)
                ++duplicates;
            m.day[r.day].hour[r.hour] = r.temperature;
        }
        else
            ++invalids;
    }
```

```
    if (invalids)
        error("Неверные данные в Month, всего ", invalids);
    if (duplicates)
        error("Повторяющиеся показания в Month, всего ",
              duplicates);
    end_of_loop(is,'}',"Неправильный конец Month");
    return is;
}
```

Позднее мы еще вернемся к функции `month_to_int()`; она преобразовывает символические обозначения месяцев, такие как `jun`, в число из диапазона [0,11]. Обратите внимание на использование функции `end_of_loop()` из раздела 10.10 для проверки признака завершения ввода. Мы подсчитываем количество неправильных и повторяющихся объектов класса `Readings` (эта информация может кому-нибудь понадобиться).

Оператор `>>` в классе `Month` выполняет грубую проверку корректности объекта класса `Reading`, прежде чем записать его в память.

```
constexpr int implausible_min = -200;
constexpr int implausible_max = 200;

bool is_valid(const Reading& r)
// Грубая проверка
{
    if (r.day <1 || 31<r.day) return false;
    if (r.hour<0 || 23<r.hour) return false;
    if (r.temperature < implausible_min ||
        implausible_max < r.temperature)
        return false;
    return true;
}
```

Наконец, мы можем прочитать объекты класса `Year`. Оператор `>>` в классе `Year` аналогичен оператору `>>` в классе `Month`.

```
istream& operator>>(istream& is, Year& y)
    // Считываем объект класса Year из потока is в объект y
    // Формат: { year 1972 ... }
{
    char ch;
    is >> ch;
    if (ch!='{') {
        is.unget();
        is.clear(ios::failbit);
        return is;
    }

    string year_marker;
    int yy;
    is >> year_marker >> yy;
    if (!is || year_marker!="year")
```

```
        error("Неправильное начало Year");
    y.year = yy;
    while(true) {
        Month m;    // Каждый раз создаем новый объект m
        if(!(is >> m)) break;
        y.month[m.month] = m;
    }

    end_of_loop(is,'}',"Неправильный конец Year");
    return is;
}
```

Можно было бы даже сказать не просто "аналогичен", а "удручающе аналогичен", если бы не один важный нюанс. Посмотрите на цикл чтения. Наверное, вы ожидали чего-то наподобие

```
for (Month m; is >> m; )
    y.month[m.month] = m;
```

Возможно, да, поскольку именно так мы до сих пор записывали все циклы ввода. Именно этот фрагмент мы написали изначально, и он оказался неправильным. Проблема в том, что функция `operator>>(istream& is, Month& m)` не присваивает объекту `m` совершенно новое значение; она просто добавляет в него данные из объекта класса `Reading`. Таким образом, повторяющаяся инструкция `is >> m` добавляла бы данные в один и тот же объект `m`. К сожалению, в этом случае каждый новый объект класса `Month` содержал бы все показания всех предшествующих месяцев текущего года. Для того чтобы считывать данные с помощью инструкции `is >> m`, нам нужен совершенно новый, пустой объект класса `Month`. Проще всего поместить определение объекта `m` в цикл так, чтобы он на каждой итерации создавался и инициализировался заново. В качестве альтернативы можно было бы сделать так, чтобы функция `operator>>(istream& is, Month& m)` перед считыванием в цикле присваивала бы объекту `m` пустой объект.

```
for (Month m; is >> m; ) {
    y.month[m.month] = m;
    m = Month{}; // "Реинициализация" m
}
```

Попробуем воспользоваться написанным.

```
// Открываем файл ввода
cout << "Введите имя входного файла\n";
string iname;
cin >> iname;
ifstream ist {iname};
if (!ifs) error("Невозможно открыть входной файл ",iname);

// Генерация исключения в случае состояния bad()
ifs.exceptions(ifs.exceptions()|ios_base::badbit);
```

```
// Открываем файл вывода
cout << "Введите имя выходного файла\n";
string oname;
cin >> oname;
ofstream ost {oname};
if (!ofs) error("Невозможно открыть выходной файл ",oname);

// Чтение произвольного количества лет:
vector<Year> ys;
while(true) {
    Year y; // На каждой итерации получаем вновь
            // инициализированный объект типа Year
    if (!(ifs>>y)) break;
    ys.push_back(y);
}
cout << "Считано " << ys.size() << " годичных записей\n";

for (Year& y : ys) print_year(ofs,y);
```

Разработку функции `print_year()` мы оставляем читателям в качестве упражнения.

10.11.3. Изменение представлений

Для нормальной работы оператора `>>` класса `Month` необходимо предусмотреть способ чтения символических представлений месяца. Для симметрии мы обеспечим заодно соответствующий способ вывода с использованием символического представления. Было бы слишком утомительно писать инструкции `if`, подобные следующей:

```
if (s=="jan")
    m = 1;
else if (s=="feb")
    m = 2;
...
```

Это не просто утомительно; таким образом мы к тому же встраиваем названия месяцев в код. Куда лучше занести их в таблицу, чтобы основная программа оставалась неизменной, даже если мы изменим символическое представление месяцев. Мы решили представить входную информацию в виде класса `vector<string>`, добавив к нему функцию инициализации и просмотра.

```
vector<string> month_input_tbl = {
    "jan", "feb", "mar", "apr", "may", "jun",
    "jul", "aug", "sep", "oct", "nov", "dec"
};

int month_to_int(string s)
// Является ли s названием месяца? Если да, возвращаем
```

```
// индекс в диапазоне [0,11], в противном случае – -1
{
for (int i=0; i<12; ++i) if (month_input_tbl[i]==s) return i;
return -1;
}
```

На всякий случай заметим, что стандартная библиотека C++ предусматривает более простой способ решения этой задачи — тип `map<string,int>` из раздела 21.6.1.

Если мы хотим вывести данные, то должны решить обратную задачу. У нас есть представление месяца в виде числа `int`, и мы хотели бы представить его в символическом виде. Наше решение очень простое, но вместо таблицы перехода от типа `string` к типу `int` мы теперь используем таблицу перехода от типа `int` к типу `string`.

```
vector<string> month_print_tbl = {
    "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December"
};

string int_to_month(int i)
// Месяц в диапазоне [0,11]
{
    if (i<0 || 12<=i) error("Неверный индекс месяца");
    return month_print_tbl[i];
}
```

Интересно, вы действительно прочитали все фрагменты кода и пояснения к ним? Или ваши глаза устали, и вы перешли сразу в конец главы? Помните, что самый простой способ научиться писать хорошие программы — читать много чужих программ. Хотите — верьте, хотите — нет, но методы, использованные в описанном примере, весьма просты, хотя разработать их самому, без посторонней помощи — задача не тривиальная. Ввод данных — фундаментальная задача. Правильная разработка циклов ввода (с корректной инициализацией каждой использованной переменной) также очень важна. Не меньшее значение имеет задача преобразования одного представления в другое. Иначе говоря, вы никуда не денетесь и *научитесь* делать такие вещи. Единственный вопрос — насколько хорошо вы научитесь их делать и сколько времени затратите на это.

Задание

1. Разработайте программу, работающую с точками (см. раздел 10.4). Начните с определения типа данных `Point`, имеющего два члена — координаты `x` и `y`.
2. Используя код и обсуждение из раздела 10.4, предложите пользователю ввести семь пар (x, y). После ввода данных сохраните их в вектор объектов `Point` с именем `original_points`.
3. Выведите на экран данные из `original_points`, чтобы увидеть, как они выглядят.
4. Откройте поток `ofstream` и выведите все точки в файл `mydata.txt`. В операционной системе Windows расширение файла `.txt` облегчает просмотр данных с помощью простого текстового редактора (такого, как WordPad).
5. Закройте поток `ofstream`, а затем откройте поток `ifstream` для файла `mydata.txt`. Введите данные из файла `mydata.txt` и запишите их в новый вектор с именем `processed_points`.
6. Выведите на экран данные из обоих векторов.
7. Сравните эти два вектора и выведите на печать сообщение `Что-то не так!`, если количество элементов (или значения элементов) в векторах не совпадает.

Контрольные вопросы

1. Насколько разнообразны устройства ввода и вывода у современных компьютеров?
2. Что делает поток `istream`?
3. Что делает поток `ostream`?
4. Что такое файл?
5. Что такое формат файла?
6. Назовите четыре разных типа устройств для ввода и вывода данных из программ.
7. Перечислите четыре этапа чтения файла.
8. Перечислите четыре этапа записи файлов.
9. Назовите и определите четыре состояния потоков.
10. Обсудите возможные способы разрешения следующих проблем при вводе.

- Пользователь набрал значение, выходящее за пределы допустимого диапазона.
- Данных нет (при чтении сразу же встречается конец файла).
- Пользователь набрал значение неправильного типа.

11. В чем ввод сложнее вывода?
12. В чем вывод сложнее ввода?
13. Почему мы (зачастую) хотим отделить ввод и вывод от вычислений?
14. Назовите две распространенные ситуации, в которых используется функция `clear()` класса `istream`.
15. Как выглядят объявления функций операторов `<<` и `>>` для пользовательского типа `X`?

Термины

`bad()`
`buffer`
`clear()`
`close()`
`eof()`
`fail()`
`good()`
`ifstream`
`iostream`
`istream`
`ofstream`
`open()`
`ostream`
`unget()`
драйвер устройства
оператор ввода
оператор вывода
признак завершения ввода
состояние потока
структурированный файл
устройство ввода
устройство вывода
файл

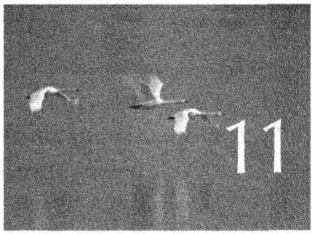
Упражнения

1. Напишите программу, вычисляющую сумму всех целых чисел, записанных в файле и разделенных пробелами.
2. Напишите программу, создающую файл из данных, записанных в виде объектов класса `Reading`, определенного в разделе 10.5. Для проверки заполните файл как минимум 50 показаниями температуры. Назовите эту программу `store_temps.cpp`, а создаваемый файл — `raw_temps.txt`.
3. Напишите программу, считывающую данные из файла `raw_temps.txt`, созданного в упр. 2, в вектор, а затем вычислите среднее значение и медиану температур. Назовите программу `temp_stats.cpp`.
4. Модифицируйте программу `store_temps.cpp` из упр. 2, включив в нее суффикс `c` для шкалы Цельсия и суффикс `f` для шкалы Фаренгейта. Затем модифицируйте программу `temp_stats.cpp`, чтобы перед записью в вектор она проверяла каждое показание и преобразовывала температуру в градусах Цельсия в градусы Фаренгейта.
5. Напишите функцию `print_year()`, упомянутую в разделе 10.11.2.

6. Определите класс `Roman_int` для хранения римских цифр (как чисел типа `int`) с операторами `<<` и `>>`. Включите в класс `Roman_int` функцию `as_int()`, возвращающую значение типа `int`, так, чтобы, если объект `r` имеет тип `Roman_int`, мы могли написать `cout << "Римское " << r << " равно " << r.as_int() << '\n';`.
7. Разработайте вариант калькулятора из главы 7, который работал бы не с арабскими, а с римскими цифрами, например `XXI + CIV == CXXV`.
8. Напишите программу, принимающую на вход имена двух файлов и создающую новый файл, в котором за содержимым первого файла следует содержимое второго; иначе говоря, программа должна конкатенировать два файла.
9. Напишите программу, принимающую на вход два файла, содержащие лексикографически упорядоченные слова, разделенные пробелами, и объедините их, сохранив упорядочение.
10. Добавьте в калькулятор из главы 7 команду `from x`, осуществляющую ввод данных из файла `x`. Добавьте в калькулятор команду `to y`, выполняющую вывод (как обычных данных, так и сообщений об ошибках) в файл `y`. Напишите набор тестов, основанных на идеях из раздела 7.3, и примените его для проверки калькулятора. Объясните, как использовать эти команды для тестирования.
11. Напишите программу, вычисляющую сумму всех целых чисел, хранящихся в текстовом файле и разделенных пробелами и словами. Например, строка "`bears: 17 elephants 9 end`" дает результат, равный `26`.

Послесловие

Большинство вычислений связано с переносом больших объемов данных из одного места в другое, например копирование текста из файла на экран или пересылка музыки из компьютера в MP3-плеер. Часто по ходу дела приходится производить определенные преобразования данных. Библиотека ввода-вывода позволяет решать множество задач, в которых данные можно интерпретировать как последовательность (поток) значений. Ввод и вывод могут оказаться удивительно большой частью программирования. Частично это объясняется тем, что мы (или наши программы) нуждаемся в больших объемах данных, а частично — тем, что точка, в которой данные поступают в систему, отличается тем, что в ней может происходить невероятное количество ошибок. Так что мы должны сделать ввод и вывод как можно более простыми и минимизировать возможность "просачивания" некорректных данных в нашу систему.



Настройка ввода и вывода

Все должно быть просто, как можно более просто, — но не проще.
— Альберт Эйнштейн (Albert Einstein)

В этой главе мы обсудим, как адаптировать потоки ввода-вывода, описанные в главе 10, к конкретным потребностям и вкусам. Сюда входит множество деталей, которые обусловлены тем, как люди воспринимают читаемые тексты, а также практическими ограничениями на использование файлов. Заключительный пример иллюстрирует проект потока ввода, в котором можно задать собственный набор разделителей.

11.1. Регулярность и нерегулярность

Библиотека iostream (часть стандартной библиотеки языка C++, отвечающая за ввод-вывод) обеспечивает единообразную и расширяемую базу для ввода и вывода текста. Под словом "текст" мы подразумеваем нечто, что можно представить в виде последовательности символов. Таким образом, когда мы говорим о вводе и выводе, мы можем рассматривать целое число `1234` как текст, поскольку его можно записать с помощью четырех символов: `1`, `2`, `3` и `4`.

До сих пор мы не делали различий между источниками входной информации. Однако иногда этого оказывается недостаточно. Например, файлы отличаются от других источников данных (например, линий связи) тем, что позволяют адресовать отдельные байты. Кроме того, мы работали, основываясь на предположении, что тип объекта полностью определен схемой его ввода и вывода. Это не совсем правильно и совсем недостаточно. Например, при выводе мы часто хотим указывать количество цифр, используемых для представления числа с плавающей точкой (его точность). В данной главе описано много способов, с помощью которых можно настроить ввод и вывод для своих потребностей.

Как программисты мы предпочитаем регулярность. Единообразная работа с любыми объектами в памяти, одинаковый подход ко всем источникам входной информации и стандартное унифицированное представление объектов, входящих в систему и выходящих из нее, позволяют создавать наиболее ясный, простой, поддерживаемый и часто самый эффективный код. Однако наши программы существуют для того, чтобы служить людям, а люди имеют свои стойкие предпочтения. Таким образом, как программисты мы должны поддерживать баланс между сложностью программы и настройкой на персональные вкусы пользователей.

11.2. Форматирование вывода

Люди уделяют много внимания мелким деталям, связанным с представлением выходной информации, которую им необходимо прочитать. Например, для физика число `1.25` (округленное до двух цифр после точки) может сильно отличаться от числа `1.24670477`, а для бухгалтера запись `(1.25)` может сильно отличаться от записи `( 1.2467)` и совершенно не совпадать с числом `1.25` (в финансовых документах скобки иногда означают убытки, т.е. отрицательные величины). Как программисты мы стремимся сделать наш вывод как можно более ясным и как можно более близким к ожиданиям потребителей нашей программы. Потоки вывода (`ostream`) предоставляют массу возможностей для форматирования вывода данных встроенных типов. Для пользовательских типов программист должен сам определить подходящие операции `<<`.

Количество деталей, уточнений и возможностей при выводе кажется бесконечным, а при вводе, наоборот, есть лишь несколько вариантов. Например, для обозначения десятичной точки можно использовать разные символы (как правило, точку или запятую); денежные суммы в разных валютах также выводятся по-разному; логическое значение "истина" можно выражать как словом `true` (или `vrai` or `sandt`), так и числом `1`, а также можно использовать разные способы представления символов, не входящих в набор ASCII (например, символы в системе Unicode). Кроме того, существуют разные способы ограничения количества символов, считываемых в строку. Эти возможности не слишком интересны, пока они не становятся нужны, поэтому мы отсылаем читателей к справочникам и специализированным книгам, таким как *Standard C++ IOStreams and Locales* Лангера (Langer); главе 21 и приложению D книги *The C++ Programming Language* Страуструпа; а также к §22 и §27 стандарта ISO C++. В настоящей книге мы рассмотрим лишь самые распространенные варианты вывода и некоторые общие концепции.

11.2.1. Вывод целых чисел

Целые числа можно выводить как восьмеричные (в системе счисления с основанием 8), десятичные (в обычной системе счисления с основанием 10) и шестнадцатеричные (в системе счисления с основанием 16). Если вы ничего не знаете об этих системах, сначала прочитайте раздел A.2.1.1. В большинстве случаев при выводе используется десятичная система счисления. Шестнадцатеричная система широко распространена при выводе информации, связанной с аппаратным обеспечением. Причина популярности шестнадцатеричной системы кроется в том, что шестнадцатеричные цифры представляет собой в точности четырехбитовые значения. Таким образом, две шестнадцатеричные цифры можно использовать для

представления восьмибитового байта, четыре шестнадцатеричные цифры представляют два байта (которые часто являются полусловом), восемь шестнадцатеричных цифр могут представить четыре байта (что часто соответствует размеру слова или регистра). Когда был разработан язык C — предшественник языка C++ (в 1970-х годах), не менее популярной была и восьмеричная система, но сейчас она используется редко.

Мы можем указать, что (десятичное число) **1234** при выводе должно трактоваться как десятичное, шестнадцатеричное или восьмеричное:

```
cout << 1234 << "\t(decimal)\n"
     << hex << 1234 << "\t(hexadecimal)\n"
     << oct << 1234 << "\t(octal)\n";
```

Символ **'\t'** означает "символ табуляции". Он обеспечивает следующее представление выходной информации:

```
1234    (decimal)
4d2     (hexadecimal)
2322    (octal)
```

Запись **<< hex** и **<< oct** не приводит к выводу значений в поток. Выражение **<< hex** сообщает потоку, что любое целое число в дальнейшем должно быть представлено в шестнадцатеричном виде, а выражение **<< oct** означает, что любое целое число в дальнейшем должно быть представлено как восьмеричное. Рассмотрим пример.

```
cout << 1234 << '\t' << hex << 1234
     << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n'; // Продолжает действовать
                      // восьмеричная система счисления
```

В итоге получаем следующий вывод:

```
1234    4d2     2322
2322 // Целые числа продолжают выводиться как восьмеричные
```

Обратите внимание на то, что последнее число выведено как восьмеричное; иначе говоря, выражения **oct**, **hex** и **dec** (для десятичных чисел) являются постоянными, или "клейкими" (sticky), — они применяются к каждому целому числу, пока не будет указано иное. Члены **hex** и **oct** используются для изменения поведения потока и называются *манипуляторами* (manipulators).

ПОПРОБУЙТЕ

Выведите свой день рождения в десятичной, восьмеричной и шестнадцатеричной записях. Пометьте каждое из этих значений. Выровняйте вывод по столбцам с помощью символа табуляции и выведите свой возраст.

Представление чисел в системе счисления, отличной от десятичной, может ввести читателя в заблуждение. Например, если заранее не знать, в какой системе представлено число, то строка 11 может означать десятичное число 11, а не восьмеричное число 9 (т.е. 11 в восьмеричной системе) или шестнадцатеричное число 17 (т.е. 11 в шестнадцатеричной системе). Для того чтобы избежать таких проблем, можно попросить поток указывать основание системы счисления, в которой представлено целое число. Рассмотрим пример.

```
cout << 1234 << '\t' << hex << 1234
     << '\t' << oct << 1234 << '\n';
cout << showbase << dec;   // Указывать основание счисления
cout << 1234 << '\t' << hex << 1234
     << '\t' << oct << 1234 << '\n';
```

В результате получаем следующий вывод:

```
1234    4d2     2322
1234    0x4d2   02322
```

Как видите, десятичные числа не имеют префиксов, восьмеричные числа имеют префикс 0, а шестнадцатеричные числа имеют префикс **0x** (или **0X**). Именно так записываются целочисленные литералы в языке C++. Рассмотрим пример.

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

В десятичном виде эти числа выглядели бы так:

```
1234    1234    1234
```

Как вы могли заметить, манипулятор **showbase** также является постоянным, как и манипуляторы **oct** и **hex**. Манипулятор **noshowbase** отменяет действие манипулятора **showbase**, возвращая поток в состояние по умолчанию, в котором любое число выводится без указания основания системы счисления.

Подытоживая, перечислим манипуляторы вывода для целых чисел.

Манипуляторы для вывода целых чисел	
oct	Использовать восьмеричную систему счисления
dec	Использовать десятичную систему счисления
hex	Использовать шестнадцатеричную систему счисления
showbase	Префикс 0 для восьмеричных и **0x** для шестнадцатеричных чисел
noshowbase	Не использовать префиксы

11.2.2. Ввод целых чисел

По умолчанию оператор >> предполагает, что используются числа в десятичной системе счисления, но его можно заставить вводить целые числа как шестнадцатеричные или восьмеричные.

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

Если вы введете

```
1234 4d2 2322 2322
```

то получите на выходе

```
1234 1234 1234 1234
```

Обратите внимание на то, что при вводе манипуляторы `oct`, `dec` и `hex` являются такими же постоянными, как и при выводе.

ПОПРОБУЙТЕ

Завершите фрагмент кода, приведенный выше, и преобразуйте его в программу. Попробуйте ввести предложенные выше числа; затем введите числа

```
1234   1234   1234   1234
```

Объясните результат. Попробуйте ввести другие числа, чтобы увидеть, что произойдет.

Для того чтобы принять и правильно интерпретировать префиксы `0` и `0x`, можно использовать оператор >>. Для этого необходимо отменить установки, принятые по умолчанию, например

```
cin.unsetf(ios::dec); // Не считать десятичным (т.е. 0x может означать
                      // шестнадцатеричное число)
cin.unsetf(ios::oct); // Не считать восьмеричным
                      // (т.е. 12 может означать двенадцать)
cin.unsetf(ios::hex); // Не считать шестнадцатеричным
                      // (т.е. 12 может означать двенадцать)
```

Функция-член потока `unsetf()` сбрасывает флаг (или флаги), указанный в качестве аргумента. Теперь, если вы напишете

```
cin >> a >> b >> c >> d;
```

и введете

```
1234 0x4d2 02322 02322
```

то получите

```
1234 1234 1234 1234
```

11.2.3. Вывод чисел с плавающей точкой

Если вы непосредственно работаете с аппаратным обеспечением, то вам нужны шестнадцатеричные числа (и, возможно, восьмеричные). Аналогично, если вы проводите научные вычисления, то должны форматировать числа с плавающей точкой. Они обрабатываются манипуляторами потока **iostream** почти так же, как и целые числа. Рассмотрим пример.

```
cout << 1234.56789 << "\t\t(по умолчанию)\n"
     << fixed << 1234.56789 << "\t(фиксированный)\n"
     << scientific << 1234.56789 << "\t(научный)\n";
```

В результате мы получим следующий вывод:

```
1234.57          (по умолчанию)
1234.567890      (фиксированный)
1.234568e+003    (научный)
```

Манипуляторы **fixed**, **scientific** и **defaultfloat** используются для выбора форматов для представления чисел с плавающей точкой. Манипулятор **defaultfloat** устанавливает формат, принятый по умолчанию (и известный также как *общий формат* (general format)). Теперь можно написать следующий код:

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';          // Формат "клейкий"
cout << defaultfloat << 1234.56789   // Формат по умолчанию
     << '\t' << fixed << 1234.56789
     << '\t' << scientific << 1234.56789 << '\n';
```

В итоге получается следующий вывод:

```
1234.57    1234.567890    1.234568e+003
1.234568e+003   // Манипулятор научного формата "клейкий"
1234.57    1234.567890    1.234568e+003
```

Подытожим манипуляторы для работы с числами с плавающей точкой.

Формат чисел с плавающей точкой	
fixed	Использовать представление с фиксированной точкой
scientific	Использовать мантиссу и показатель степени; мантисса всегда находится в диапазоне [1,10), т.е. перед десятичной точкой всегда стоит ненулевая десятичная цифра
defaultfloat	Выбирает манипулятор **fixed** или **scientific** для наиболее точного представления чисел в рамках точности самого **defaultfloat**

11.2.4. Точность

По умолчанию число с плавающей точкой выводится на печать с помощью шести цифр в формате `defaultfloat`. Формат, состоящий из шести цифр (точность по умолчанию формата `defaultfloat`), считается наиболее подходящим, а такое округление числа — наилучшим. Рассмотрим пример.

`1234.567` выводится на печать как `1234.57`

`1.2345678` выводится на печать как `1.23457`

Округление, как правило, выполняется по правилу 4/5: от 0 до 4 — округление вниз, а от 5 до 9 — вверх. Обратите внимание на то, что такое форматирование относится только к числам с плавающей точкой, так что

`1234567` выводится на печать как `1234567` (поскольку число целое)

`1234567.0` выводится на печать как `1.23457e+006`

В последнем случае поток `ostream` распознает, что число `1234567.0` нельзя вывести на печать в формате `fixed`, используя только шесть цифр, и переключается на формат `scientific`, чтобы обеспечить как можно более точное представление числа. По сути, формат `defaultfloat` выбирает между форматами `scientific` и `fixed`, чтобы обеспечить максимально точное представление числа с плавающей точкой в рамках формата `defaultfloat`, предусматривающего использование шести цифр.

ПОПРОБУЙТЕ

Напишите программу, три раза выводящую на печать число `1234567.89`, сначала в формате `defaultfloat`, затем — в `fixed`, потом — в `scientific`. Какая форма вывода обеспечивает наиболее точное представление числа и почему?

Программист может установить точность представления числа, используя манипулятор `setprecision()`, например:

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(5)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(0)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
```

Этот код выводит на печать следующие числа (обратите внимание на округление):

```
1234.57      1234.567890      1.234568e+003
1234.6       1234.56789       1.23457e+003
1234.5679    1234.56789000    1.23456789e+003
```

Точность определятся по правилам, приведенным ниже.

Точность чисел с плавающей точкой	
`defaultfloat`	Точность представляет собой общее количество цифр
`scientific`	Точность представляет собой количество цифр после десятичной точки
`fixed`	Точность представляет собой количество цифр после десятичной точки

Мы рекомендуем использовать формат, принятый по умолчанию (формат `defaultfloat` с точностью, равной шести цифрам), если у вас нет весомых причин для применения другого формата. Обычно причина, по которой выбираются другие форматы, такова: "Мы хотим получить большую точность при выводе".

11.2.5. Поля

С помощью научного и фиксированного формата программист может точно контролировать, сколько места займет число на выходе. Это очень полезно при распечатке таблиц и т.п. Эквивалентный механизм для целых чисел называют *полями* (fields). Вы можете точно указать ширину поля, используя манипулятор `setw()`:

```
cout << 123456 <<'|'              // Поля не используются
     << setw(4) << 123456 << '|' // Число 123456 не помещается
                                  // в поле из 4 символов
     << setw(8) << 123456 << '|' // Ширина поля равна 8
     << 123456 << "|\n";          // Размер поля не "клейкий"
```

В итоге получим следующий результат:

```
123456|123456|  123456|123456|
```

Обратите внимание на два пробела перед третьим числом `123456`. Это — результат того, что мы выводим шесть цифр в поле, состоящее из восьми символов. Однако число `123456` невозможно усечь так, чтобы оно помещалось в поле, состоящем из четырех символов. Почему? Конечно, числа `|1234|` и `|3456|` можно интерпретировать как вполне допустимые для поля, состоящего из четырех символов, однако в этом случае читатель увидит числа, которые не имеют ничего общего с правильными значениями, и при этом он не получит об усечении никакого предупреждения. Уж лучше нарушить форматирование, чем вывести неверную информацию. Плохое форматирование почти всегда лучше, чем "плохие результаты".

В большинстве случаев (например, при выводе таблиц) переполнение полей сразу бросается в глаза, так что его можно откорректировать.

Поля можно использовать и при выводе строк и чисел с плавающей точкой:

```
cout << 12345 <<'|'<< setw(4) << 12345 << '|'
     << setw(8) << 12345 << '|' << 12345 << "|\n";
cout << 1234.5 <<'|'<< setw(4) << 1234.5 << '|'
     << setw(8) << 1234.5 << '|' << 1234.5 << "|\n";
cout << "asdfg" <<'|'<< setw(4) << "asdfg" << '|'
     << setw(8) << "asdfg" << '|' << "asdfg" << "|\n";
```

Этот код выводит на печать следующее:

```
12345|12345|   12345|12345|
1234.5|1234.5|  1234.5|1234.5|
asdfg|asdfg|   asdfg|asdfg|
```

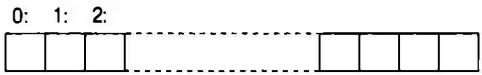
Обратите внимание на то, что ширина поля не является постоянной ("клейкой"). Во всех трех случаях первое и последнее значения выведены в формате по умолчанию, с тем количеством символов, которые они занимают. Иначе говоря, если мы не укажем ширину поля непосредственно перед выводом, то понятие поля вообще не будет использовано.

ПОПРОБУЙТЕ

Создайте простую таблицу, содержащую фамилию, имя, номер телефона и адрес электронной почты не менее пяти ваших друзей. Поэкспериментируйте с разной шириной поля, пока не добьетесь приемлемого вида таблицы.

11.3. Открытие файла и позиционирование

В языке C++ файл — это абстракция того, что предоставляет операционная система. Как указано в разделе 10.3, файл — это последовательность байтов, пронумерованных начиная с нуля.



Вопрос заключается лишь в том, как получить доступ к этим байтам. При работе с потоками `iostream` вид доступа в большой степени определяется в тот момент, когда мы открываем файл и связываем с ним поток. Поток сам определяет, какие операции можно выполнить после открытия файла и каков их смысл. Например, открыв для файла поток `istream`, мы можем прочитать его содержимое, а открыв для файла поток `ostream`, можем записать в него данные.

11.3.1. Режимы открытия файлов

Файл можно открыть в одном из нескольких режимов. По умолчанию поток `ifstream` открывает файлы для чтения, а поток `ofstream` — для записи. Эти операции удовлетворяют большинству наших потребностей. Однако существует несколько альтернатив.

Режимы открытия файлов	
`ios_base::app`	Добавление (запись в конец файла)
`ios_base::ate`	В конец (открыть и перейти в конец файла)
`ios_base::binary`	Бинарный режим (зависит от конкретной системы)
`ios_base::in`	Открытие файла для чтения
`ios_base::out`	Открытие файла для записи
`ios_base::trunc`	Обрезать файл до нулевой длины

Режим открытия файла можно указать после его имени:

```
ofstream ofl{name1};                  // По умолчанию ios_base::out
ifstream ifl{name2};                  // По умолчанию ios_base::in
ofstream ofs{name, ios_base::app}; // По умолчанию ofstream
                                      // включает ios_base::out
fstream fs{"myfile",
    ios_base::in|ios_base::out};      // И для ввода, и для вывода
```

Символ `|` в последнем примере — это побитовый оператор ИЛИ (раздел А.5.5), который можно использовать для объединения режимов. Опция `app` часто используется для журнальных файлов, в которых записи всегда добавляются в конец.

В любом случае точный режим открытия файла может зависеть от операционной системы. Если операционная система не может открыть файл в требуемом режиме, результатом будет поток в состоянии, отличном от `good()`.

```
if (!fs) // Ошибка: мы не можем открыть файл в таком режиме
```

В большинстве ситуаций причиной сбоя при открытии файла для чтения является его отсутствие.

```
ifstream ifs{"redungs"};
if (!ifs) // Ошибка открытия файла "readings" для чтения
```

В данном случае причиной ошибки, конечно же, стала опечатка.

Обычно, когда вы пытаетесь открыть несуществующий файл для вывода, операционная система создает новый файл, но (к счастью) она не делает этого, когда вы пытаетесь открыть несуществующий файл для ввода.

```
ofstream ofs{"no-such-file"};          // Создает новый файл
                                       // "no-such-file"
ifstream ifs{"no-file-of-this-name"}; // Ошибка: поток ifs не
                                       // в состоянии good()
```

Не пытайтесь "умничать" с режимами открытия файла. Операционные системы не обрабатывают "необычные" режимы согласованно. По возможности выполняйте чтение из файлов, открытых с помощью **istream**, а запись — в файлы, открытые с помощью **ostream**.

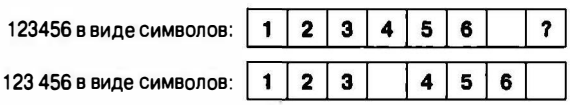
11.3.2. Бинарные файлы

В памяти можно представить значение 123 как целое или как строку. Рассмотрим пример.

```
int n = 123;
string s = "123";
```

В первом случае число **123** интерпретируется как (бинарное) число. Объем памяти, который оно занимает, совпадает с объемом памяти, который занимает любое другое число типа **int** (4 байта, т.е. 32 бита на персональном компьютере). Если вместо числа **123** мы выберем число **12345**, то оно по-прежнему будет занимать те же самые четыре байта. Во втором варианте значение **123** хранится как строка из трех символов. Если мы выберем строку **"12345"**, то для ее хранения нам потребуются пять символов (плюс накладные расходы памяти на управление объектом класса **string**). Проиллюстрируем сказанное, используя обычные десятичное и символьное представления, а не бинарное, как в памяти компьютера.

Когда мы используем символьное представление, то какой-то символ должен служить признаком конца числа, так же как на бумаге, когда мы записываем одно число 123456 и два числа 123 456. На бумаге для разделения чисел мы используем пробел. То же самое можно сделать в памяти компьютера.



Разница между хранением бинарного представления фиксированного размера (например, в виде типа **int**) и символьного представления

переменного размера (например, в виде типа **string**) проявляется и при работе с файлами. По умолчанию потоки **iostream** работают с символьными представлениями; иначе говоря, поток **istream** считывает последовательность символов и превращает их в объект заданного типа. Поток **ostream** принимает объект заданного типа и преобразует его в последовательность записываемых символов. Однако можно потребовать, чтобы потоки **istream** и **ostream** просто копировали байты из файла и в файл. Такой ввод-вывод называется *бинарным* (binary I/O). В этом случае файл необходимо открывать в режиме **ios_base::binary**. Рассмотрим пример, в котором считываются и записываются бинарные файлы, содержащие целые числа. Ключевые строки, работающие с бинарными файлами, поясняются ниже.

```
int main()
{
    // Открытие потока istream для бинарного чтения файла
    cout << "Введите имя файла для ввода\n";
    string iname;
    cin >> iname;
    ifstream ifs {iname,ios_base::binary}; // Обратите внимание,
        // что бинарный режим требует от файлового потока не
        // пытаться что-либо делать с потоком байтов
    if (!ifs) error("Невозможно открыть входной файл ",iname);

    // Открытие потока ostream для бинарной записи файла
    cout << "Введите имя файла для вывода\n";
    string oname;
    cin >> oname;
    ofstream ofs {oname,ios_base::binary}; // Обратите внимание,
        // что бинарный режим требует от файлового потока не
        // пытаться что-либо делать с потоком байтов
    if (!ofs) error("Невозможно открыть выходной файл ",oname);

    vector<int> v;

    // Чтение из бинарного файла:
    for(int x; ifs.read(as_bytes(x),sizeof(int));) // Чтение
        v.push_back(x);                            // байтов

    // ... работа с вектором v ...

    // Запись в бинарный файл:
    for(int x : v)
    ofs.write(as_bytes(x),sizeof(int)); // Запись байтов
    return 0;
}
```

Мы открываем файлы с использованием **ios_base::binary** в качестве режима работы потока:

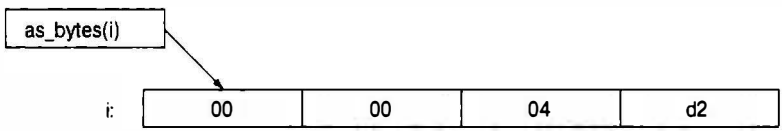
```
ifstream ifs{name.c_str(), ios_base::binary};
ofstream ofs{name.c_str(), ios_base::binary};
```

В обоих случаях мы выбрали более сложное, но зачастую более компактное бинарное представление. При переходе от символьно-ориентированного ввода-вывода к бинарному мы не можем использовать обычные операторы ввода и вывода >> и <<. Эти операторы преобразуют значения в последовательности символов, руководствуясь установленными по умолчанию правилами (например, строка "asdf" превращается в символы a, s, d, f, а число 123 превращается в символы 1, 2, 3). Если вы не хотите работать с бинарным представлением чисел, достаточно ничего не делать и использовать режим по умолчанию. Мы рекомендуем применять опцию binary, только если вы (или кто-нибудь еще) считаете, что так будет лучше. Например, с помощью опции binary можно сообщить потоку, что он ничего не должен делать с байтами.

А что вообще мы могли бы сделать с типом int? Очевидно, записать его в память размером четыре байта; иначе говоря, мы могли бы обратиться к представлению типа int в памяти (последовательность четырех байтов) и записать эти байты в файл в том виде, в котором они находятся в памяти. Позднее мы могли бы преобразовать эти байты обратно в целое число.

```
ifs.read(as_bytes(i),sizeof(int))      // Чтение байтов
ofs.write(as_bytes(v[i]),sizeof(int)) // Запись байтов
```

Функция write() потока ostream и функция read() потока istream получают адрес в памяти (получаемый функцией as_bytes()) и количество байтов (символов), полученное с помощью оператора sizeof. Адрес должен указывать на первый байт в памяти, хранящей значение, которое мы хотим прочитать или записать. Например, если у нас есть объект типа int со значением 1234, то мы могли бы получить четыре байта (используя шестнадцатеричную систему обозначений) — 00, 00, 04, d2:



Функция as_bytes() позволяет получить адрес первого байта объекта. Ее определение выглядит так (некоторые особенности языка, использованные здесь, будут рассмотрены в разделах 17.8 и 19.3):

```
template<class T>
char* as_bytes(T& i) // Рассматривает объект T как
                     // последовательность байтов
{
    void* addr = &i; //Получаем адрес первого байта
                     //памяти объекта
```

```
    return static_cast<char*>(addr); // Указатель на байты
}
```

Небезопасное преобразование типа с помощью оператора **static_cast** необходимо для того, чтобы получить переменную в виде последовательности байтов. Понятие адреса будет подробно изучено в главах 17 и 18. Здесь мы просто показываем, как представить любой объект, хранящийся в памяти, в виде последовательности байтов, чтобы прочитать или записать его с помощью функций **read()** и **write()**.

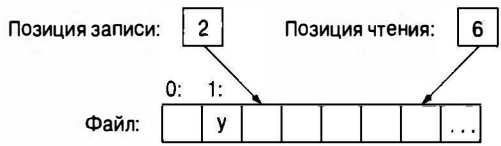
Этот двоичный вывод запутан, сложен и уязвим для ошибок. Однако программисты не всегда имеют полную свободу выбора формата файла, поэтому иногда они просто вынуждены использовать двоичный ввод-вывод по не зависящим от них причинам. Кроме того, отказ от символьного представления иногда имеет логичные причины. Типичными примерами являются рисунок или звуковой файл, не имеющий разумного символьного представления: фотография или фрагмент музыкального произведения по своей природе является совокупностью битов.

Символьный ввод-вывод, по умолчанию предусмотренный в библиотеке **iostream**, переносим, доступен для непосредственного чтения человеком и поддерживается любыми средствами набора текстов. Если у вас есть возможность выбора, рекомендуем использовать именно символьный ввод-вывод, а бинарный ввод-вывод применять только в случаях, когда без него никак не обойтись.

11.3.3. Позиционирование в файлах

При малейшей возможности считывайте и записывайте файлы от начала до конца. Это проще всего и оставляет меньше возможностей для совершения ошибок. Каждый раз, когда вы понимаете, что пора изменить файл, лучше создайте новый и запишите в него все изменения.

Однако, если вы не можете так поступить, выполните позиционирование и укажите конкретное место для чтения или записи в файле. В любом файле, открытом для чтения, существует позиция для чтения/ввода ("read/get position"), а в любом файле, открытом для записи, есть позиция для записи/вывода ("write/put position").



Эти позиции можно использовать следующим образом.

```
fstream fs {name}; // Открыть для ввода и вывода
if (!fs) error("Невозможно открыть файл ",name);
```

```
fs.seekg(5); // Перенести позицию чтения (g означает "get")
             // на пять ячеек вперед (шестой символ)
char ch;
fs>>ch;      // Считать и увеличить позицию чтения
cout << "шестой символ — это " << ch << '(' << int(ch) << ")\n";
fs.seekp(1); // Перенести позицию записи (p означает "put")
             // на одну ячейку вперед
fs<<'y';     // Записать и увеличить позицию записи
```

Обратите внимание, что позиции **seekg()** и **seekp()** при чтении и записи увеличиваются, так что на рисунке показано состояние дел *после* выполнения кода.

Будьте осторожны: ошибки позиционирования не распознаются. В частности, если вы попытаетесь выйти за пределы файла (используя функцию **seekg()** или **seekp()**), то последствия могут быть непредсказуемыми.

11.4. Строковые потоки

В качестве источника ввода для потока **istream** или целевого объекта вывода для потока **ostream** можно использовать объект класса **string**. Поток **istream**, считывающий данные из объекта класса **string**, называется **istringstream**, а поток **ostream**, записывающий символы в объект класса **string**, называется **ostringstream**. Например, поток **istringstream** полезен для извлечения числовых значений из строк.

```
double str_to_double(string s)
    // По возможности преобразует символы из строки s
    // в число с плавающей точкой
{
    istringstream is{s}; // Создаем поток ввода из строки s
    double d;
    is >> d;
    if (!is) error("Ошибка формата типа double: ",s);
    return d;
}

double d1 = str_to_double("12.4");              // Проверка
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three"); // Вызов error()
```

Если попытаться прочитать данные за пределами строки, предназначенной для ввода в поток **istringstream**, поток перейдет в состояние **eof()**. Это значит, что для потока **istringstream** можно использовать обычный цикл ввода; фактически поток **istringstream** является разновидностью потока **istream**.

Поток **ostringstream**, наоборот, может быть полезен для форматирования вывода в системах, ожидающих аргумента в виде простой строки,

например в системах графического пользовательского интерфейса (раздел 16.5), например:

```
void my_code(string label, Temperature temp)
{
    // ...
    ostringstream os;  // Поток для составления сообщения
    os << setw(8) << label << ": "
       << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    // ...
}
```

Функция-член `str()` класса `ostringstream` возвращает объект класса `string`, составленный операциями вывода в поток `ostringstream`. Функция `c_str()` — это функция-член класса `string`, возвращающая строки в стиле языка C, которые требуются многими системными интерфейсами.

Потоки `stringstream` обычно используются, когда мы хотим отделить собственно ввод-вывод от обработки данных. Например, аргумент типа `string` в функции `str_to_double()` обычно поступает из файла (например, из журнала событий веб) или с клавиатуры. Аналогично сообщение, составленное функцией `my_code()`, в конце концов выводится на экран. Например, в разделе 11.7 мы используем поток `stringstream` при выводе для фильтрации нежелательных символов. Таким образом, потоки `stringstream` можно интерпретировать как механизм настройки ввода-вывода для особых потребностей и вкусов.

Продемонстрируем использование потока `ostringstream` на примере построения строки путем конкатенации.

```
int seq_no = get_next_number();        // Номер журнального файла
ostringstream name;
name << "myfile" << seq_no << ".log"; // Например, myfile17.log
ofstream logfile{name.str()};          // Открытие myfile17.log
```

Обычно поток `istringstream` инициализируется объектом класса `string`, а затем считывает из него символы, используя операторы ввода. И наоборот, поток `ostringstream` инициализируется пустым объектом класса `string`, а затем заполняется с помощью операторов вывода. Существует более простой способ доступа к символам в потоке `stringstream`, который иногда оказывается полезным: функция `ss.str()` возвращает копию строки из объекта `ss`, а функция `ss.str(s)` присваивает строке в объекте `ss` копию строки `s`. В разделе 11.7 приведен пример, в котором функция `ss.str(s)` играет существенную роль.

11.5. Ввод, ориентированный на строки

Оператор >> считывает данные в объекты заданного типа в соответствии со стандартным форматом, установленным для этого типа. Например, при вводе чисел в объект типа `int` оператор >> будет выполнять ввод, пока не обнаружит символ, не являющийся цифрой, а при вводе в объект класса `string` оператор >> будет считывать символы, пока не обнаружит пробельный символ (whitespace). Стандартная библиотека `istream` содержит также средства для ввода отдельных символов и целых строк:

```
string name;
cin >> name;           // Ввод: Dennis Ritchie
cout << name << '\n';  // Вывод: Dennis
```

Но что если мы захотим прочитать всю строку сразу, а как она форматирована, будем решать потом? Это можно сделать с помощью функции `getline()`:

```
string name;
getline(cin,name);     // Ввод: Dennis Ritchie
cout << name << '\n';  // Вывод: Dennis Ritchie
```

Теперь мы считали целую строку. Почему нам это может понадобиться? Один неплохой ответ звучит так: "Потому что мы сделали то, чего не может оператор >>". Часто можно слышать неудачное объяснение: "Потому что пользователь ввел целую строку". Если это все, что вы можете сказать, то используйте оператор >>, потому что, если вы ввели строку, то должны как-то разбить ее на части, например:

```
string first_name;
string second_name;
stringstream ss(name);
ss >> first_name;       // Ввод строки Dennis
ss >> second_name;      // Ввод строки Ritchie
```

Непосредственное чтение данных в строки `first_name` и `second_name` явно выглядит проще.

Одна из распространенных причин для считывания полной строки заключается в том, что определение пробельного символа не всегда является достаточно приемлемым. Иногда переход на новую строку желательно трактовать не как пробельный символ. Например, в ходе обмена сообщениями в компьютерной игре строку разумнее интерпретировать как предложение, не полагаясь на общепринятую пунктуацию:

```
Идти налево, пока не увидишь картину справа на стене
Сними картину со стены и открой дверь позади нее. Возьми сундук
```

В данном случае мы сначала читаем всю строку, а затем извлекаем из нее отдельные слова.

```
string command;
getline(cin,command);       // Читаем строку
stringstream ss {command};

vector<string> words;
for (string s; ss>>s; )
words.push_back(s);         // Извлекаем отдельные слова
```

С другой стороны, если есть выбор, то лучше всего ориентироваться на знаки пунктуации, а не на символ перехода на новую строку.

11.6. Классификация символов

Как правило, мы вводим целые числа, числа с плавающей точкой, слова и так далее, в соответствии с общепринятым форматом. Однако мы можем, а иногда и должны, снизить уровень абстракции и вводить отдельные символы. Для этого необходимо затратить больше усилий, но, считывая отдельные символы, мы получаем полный контроль на тем, что делаем. Рассмотрим задачу распознавания лексем в выражениях из раздела 7.8.2. Допустим, мы хотим разделить выражение `1+4*x<=y/z*5` на одиннадцать лексем.

```
1 + 4 * x <= y / z * 5
```

Для ввода чисел мы могли бы использовать оператор `>>`, но, пытаясь вводить идентификаторы как строки, мы бы прочли `x<=y` как единую строку (поскольку символы `<` и `=` не являются пробельными). Сочетание символов `z*` мы также должны были бы ввести как целую строку (поскольку символ `*` также не является пробельным).

Вместо этого можно сделать следующее:

```
for(char ch;cin.get(ch);) {
    if (isspace(ch)) { // Если символ ch является пробельным,
                       // ничего не делаем (игнорируем
                       // пробельный символ)
    }
    if (isdigit(ch)) {
                       // Читаем число
    }
    else if (isalpha(ch)) {
                       // Читаем идентификатор
    }
    else {
                       // Обрабатываем операторы
    }
}
```

Функция `istream::get()` считывает отдельный символ в свой аргумент. Пробельные символы при этом не игнорируются. Как и оператор `>>`,

функция `get()` возвращает ссылку на поток `istream`, так что можно проверить его состояние.

При вводе отдельных символов мы обычно хотим классифицировать их: это символ или цифра? В верхнем регистре или в нижнем? И так далее. Для этого существует набор стандартных библиотечных функций.

Классификация символов	
`isspace(c)`	Является ли с пробельным символом (`' '`, `'\t'`, `'\n'` и т.д.)?
`isalpha(c)`	Является ли с буквой (`'a'..'z'`, `'A'..'Z'`) (примечание: не `'_'`)?
`isdigit(c)`	Является ли с десятичной цифрой (`'0'..'9'`)?
`isxdigit(c)`	Является ли с шестнадцатеричной цифрой (десятичной цифрой или символом `'a'..'f'` или `'A'..'F'`)?
`isupper(c)`	Является ли `c` буквой в верхнем регистре?
`islower(c)`	Является ли `c` буквой в нижнем регистре?
`isalnum(c)`	Является ли `c` буквой или десятичной цифрой?
`iscntrl(c)`	Является ли `c` управляющим символом (ASCII 0..31 или 127)?
`ispunct(c)`	Проверка, что `c` не является ни буквой, ни цифрой, ни разделителем, ни невидимым управляющим символом
`isprint(c)`	Выводится ли символ `c` на печать (ASCII `' '..'~'`)?
`isgraph(c)`	Выполняется ли для `c` условие `isalpha()` или `isdigit()` или `ispunct()` (примечание: не пробел)?

Заметим, что эти классификации можно объединять с помощью оператора ИЛИ (`||`). Например, выражение `isalnum(c)` означает `isalpha(c) ||isdigit(c)`; иначе говоря, оно истинно, если символ `c` является буквой или цифрой.

Кроме того, в стандартной библиотеке есть две полезные функции для устранения различий между символами, набранными в разных регистрах.

Регистр символа	
`toupper(c)`	`c` или его эквивалент в верхнем регистре
`tolower(c)`	`c` или его эквивалент в нижнем регистре

Это удобно, когда мы хотим, например, сравнивать символы без учета регистра. Так, если пользователь ввел слова `Right`, `right` и `rigHT`, то, скорее всего, он имел в виду одно и то же (например, слово `rigHT` чаще всего является результатом нечаянного нажатия клавиши <Caps Lock>). Применив функцию `tolower()` к каждому символу в каждой из строк, мы можем получить одно и то же значение: `right`. Эту операцию можно выполнить с любым объектом класса `string`.

```
void tolower(string& s) // Перевод строки в нижний регистр
{
```

```
    for (char&x: s) x = tolower(x);
}
```

Для того чтобы действительно изменить объект класса **string**, мы используем передачу аргумента по ссылке (см. раздел 8.5.5). Если бы мы хотели сохранить старую строку без изменения, то могли бы написать функцию, создающую ее копию в нижнем регистре. Мы предпочитаем функцию **tolower()**, а не **toupper()**, поскольку она лучше работает с текстами на некоторых естественных языках, например немецком, в которых не у каждого символа в нижнем регистре есть эквивалент в верхнем регистре.

11.7. Использование нестандартных разделителей

В этом разделе мы рассмотрим полуреалистичные примеры использования потоков **iostream** для решения реальных задач. При вводе строк слова по умолчанию разделяются пробельными символами. К сожалению, поток **istream** не имеет средств, позволяющих определять, какие символы должны играть роль разделителей, или непосредственно изменять способ, с помощью которого оператор >> считывает строки. Итак, что делать, если мы хотим дать другое определение пробельного символа? Рассмотрим пример из раздела 4.6.3, в котором мы считывали слова и сравнивали их между собой. Между этими словами стояли разделители, поэтому, если мы вводили строку

```
As planned, the guests arrived; then,
```

то получали слова

```
As
planned,
the
guests
arrived;
then,
```

Это слова невозможно найти в словаре: "**planned,**" и "**arrived;**" вообще не являются словами. Это набор букв, состоящий из слов, к которым присоединены лишние и не относящиеся к делу знаки пунктуации. В большинстве случаев мы должны рассматривать знаки пунктуации как разделители. Как же нам избавиться от этих знаков пунктуации? Мы могли бы считать символы, удалить знаки пунктуации или преобразовать их в пробелы, а затем ввести "очищенные" данные снова.

```
string line;
getline(cin,line);      // Чтение в line
for (char& ch : line)   // Замена символов пунктуации пробелами
    switch(ch) {
```

```
    case ';': case '.': case ',': case '?': case '!':
        ch = ' ';
    }

stringstream ss(line); // Создаем istream ss для чтения line
vector<string> vs;
                       // Чтение слов без символов пунктуации
for (string word; ss>>word; )
    vs.push_back(word);
```

Применив такой способ, мы получаем желаемый результат:

```
As
planned
the
guests
arrived
then
```

К сожалению, этот код слишком сложен и излишне специализирован. А что делать, если знаки пунктуации определены иначе? Опишем более общий и полезный способ удаления нежелательных символов из потока ввода. Как должен выглядеть этот поток? Как должен выглядеть наш код? Может быть, так?

```
ps.whitespace(";:,."); // Точка с запятой, двоеточие, запятая и точка
                       // считаются разделителями
for(string word;ps>>word;)
    vs.push_back(word);
```

Как определить поток, работающий так, как поток **ps**? Основная идея заключается в том, чтобы считывать слова в обычный поток ввода, а затем обрабатывать символы-разделители, заданные пользователем, как настоящие разделители, т.е. не передавать разделители пользователю, а просто использовать их для отделения слов друг от друга. Рассмотрим пример.

```
as.not
```

Слова **as** и **not** должны быть двумя самостоятельными словами

```
as
not
```

Можно определить класс, который будет делать это для нас. Он должен принимать символы из потока **istream** и иметь оператор **>>**, работающий так же, как оператор ввода потока **istream**, за исключением того, что мы сами сможем указывать, какие символы являются разделителями. Для простоты мы не будем считать существующие пробельные символы (пробел, символ перехода на новую строку и т.д.) обычными символами; мы просто позволим пользователю указывать дополнительные "пробельные" символы. Кроме того, мы не будем полностью удалять эти символы

из потока; как и ранее, мы будем превращать их в пробелы. Назовем наш класс **Punct_stream**.

```
class Punct_stream { // Аналогичен istream, но пользователь
                     // может задавать пробельные символы
public:
    Punct_stream(istream& is)
        : source{is}, sensitive{true} { }

    void whitespace(const string& s) // Делает s строкой
        { white = s; }               // пробельных символов
    void add_white(char c)           // Добавляет символ в
        { white += c; }              // набор пробельных
    bool is_whitespace(char c);      // Входит ли c в набор
                                     // пробельных символов?

    void case_sensitive(bool b) { sensitive = b; }
    bool is_case_sensitive()    { return sensitive; }

    Punct_stream& operator>>(string& s);
    operator bool();
private:
    istream& source;        // Источник символов
    istringstream buffer;   // Буфер для форматирования
    string white;           // Пробельные символы
    bool sensitive;         // Чувствителен ли поток к регистру?
};
```

Основная идея, как и в предыдущем примере, — ввести строку из потока **istream** как единое целое, преобразовать "пробельные символы" в пробелы, а затем использовать поток **istringstream** для форматирования. Кроме обработки разделителей, заданных пользователем, в классе **Punct_stream** есть еще одна возможность: если вызвать функцию **case_sensitive()**, то она преобразует ввод, чувствительный к регистру, в нечувствительный. Например, можно приказать объекту класса **Punct_stream** прочитать строку

```
Man bites dog!
```

как

```
man
bites
dog
```

Конструктор класса **Punct_stream** получает поток **istream**, используемый как источник символов, и дает ему локальное имя **source**. Кроме того, конструктор по умолчанию делает поток чувствительным к регистру, как обычно. Можно создать объект класса **Punct_stream**, считывающий данные из потока **cin**, рассматривающий точку с запятой, двоеточие и точку

как разделители, а также переводящий все символы в нижний регистр, с помощью следующего кода:

```
Punct_stream ps(cin);      // ps считывает данные из потока cin
ps.whitespace(";:.");      // Точка с запятой, двоеточие и точка
                           // также являются разделителями
ps.case_sensitive(false); // Нечувствительный к регистру
```

Очевидно, что наиболее интересной операцией является оператор ввода **>>**. Он также является самым сложным для определения. Наша общая стратегия состоит в том, чтобы считать всю строку из потока **istream** в строку **line**. Затем мы превратим все символы, определенные пользователем как пробельные, в пробелы (**' '**). После этого мы поместим строку в поток **istringstream** с именем **buffer**. Теперь для считывания данных из потока **buffer** можно использовать обычный оператор **>>**, работающий со стандартными пробельными символами. Код будет выглядеть немного сложнее, поскольку мы просто пытаемся считать данные из потока **buffer**, и только когда он пуст, заполняем его.

```
Punct_stream& Punct_stream::operator>>(string& s)
{
    while (!(buffer>>s)) {          // Попытка прочитать данные
                                    // из потока buffer
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        string line;
        getline(source,line);       // Считываем строку line
                                    // из потока source

        // При необходимости заменяем символы
        for(char& ch: line)
             if (is_whitespace(ch))
                ch = ' ';           // Преобразуем в пробел
             else if (!sensitive)
                ch = tolower(ch); // Преобразуем в нижний регистр

        buffer.str(line);           // Записываем строку в поток
    }
    return *this;
}
```

Рассмотрим этот код шаг за шагом. Сначала обратим внимание на нечто необычное.

```
while (!(buffer>>s)) {
```

Если в потоке **buffer** класса **istringstream** есть символы, то выполняется инструкция **buffer>>s** и объект **s** получает слово, отделенное от других "пробельными символами"; больше эта инструкция ничего не делает. Эта

инструкция будет выполняться, пока в объекте **buffer** есть символы для ввода. Однако, когда инструкция **buffer>>s** не сможет выполнить свою работу, т.е. когда выполняется условие **!(buffer>>s)**, мы должны заполнить объект **buffer** данными из потока **source**. Обратите внимание на то, что инструкция **buffer>>s** выполняется в цикле; после попытки заполнить объект **buffer** мы должны снова попытаться выполнить ввод.

```
while (!(buffer>>s)) { // Попытка прочитать символы из буфера
     if (buffer.bad() || !source.good()) return *this;
     buffer.clear();

     // Заполняем объект buffer
}
```

Если объект **buffer** находится в состоянии **bad()** или существуют проблемы с источником данных, работа прекращается; в противном случае мы выполняем сброс состояния объекта **buffer** и делаем новую попытку. Мы должны сбросить состояние объекта **buffer**, потому что попадаем в "цикл заполнения", только если попытка ввода закончится неудачно, обычно — по достижении "конца файла" (**eof()**) объекта **buffer**; иначе говоря, когда из объекта **buffer** больше нечего читать. Обработка состояний потока всегда запутанна и часто является причиной очень тонких ошибок, требующих утомительной отладки. К счастью, остаток цикла заполнения вполне очевиден.

```
string line;
getline(source,line);      // Считываем строку line
                           // из потока source

// При необходимости заменяем символы
for(char& ch: line)
    if (is_whitespace(ch))
        ch = ' ';          // Преобразуем в пробел
    else if (!sensitive)
        ch = tolower(ch); // Преобразуем в нижний регистр

buffer.str(line);          // Записываем строку в поток
```

Мы считываем строку в объект **buffer**, затем просматриваем каждый символ строки в поисках кандидатов на замену. Функция **is_whitespace()** является членом класса **Punct_stream**, и ее мы определим позже. Функция **tolower()** — это стандартная библиотечная функция, выполняющая очевидную из названия работы, например превращает символ **A** в символ **a** (см. раздел 11.6).

Обработанную строку **line** необходимо записать в поток **istringstream**. Эту задачу выполняет вызов **buffer.str(line)**; данная инструкция может быть прочитана так: "Установить содержимое объекта **buffer** класса **istringstream** равным строке **line**".

Обратите внимание на то, что мы "забыли" проверить состояние объекта **source** после чтения данных с помощью функции **getline()**. Это не обязательно, поскольку в конечном итоге мы встретимся с проверкой условия **!source.good()** в начале тела цикла.

Как всегда, оператор **>>** возвращает ссылку на поток ***this** (раздел 17.10).

Проверка, является ли символ пробельным, проста: мы сравниваем проверяемый символ с каждым символом из строки, в которой записаны разделители.

```
bool Punct_stream::is_whitespace(char c)
{
    for(char w: white)
        if (c==w) return true;
    return false;
}
```

Напомним, что поток **istringstream** работает с обычными пробельными символами (например, символами перехода на новую строку или пробелами) так же, как и ранее, поэтому никаких особых действий предпринимать не надо.

Осталась одна загадочная функция.

```
Punct_stream::operator bool()
{
    return !(source.fail() || source.bad()) && source.good();
}
```

Обычное использование потока **istream** сводится к проверке результата оператора **>>**. Рассмотрим пример.

```
while(ps>>s) { /* . . . */ }
```

Это значит, что нам нужен способ для проверки результата выполнения инструкции **ps>>s**, представленного в виде булева значения. Результатом инструкции **ps>>s** является объект класса **Punct_stream**, поэтому нам нужен способ неявного преобразования класса **Punct_stream** в тип **bool**. Эту задачу решает функция **operator bool()**, определенная в классе **Punct_stream**. Функция-член **operator bool()** определяет преобразование класса **Punct_stream** в тип **bool**. В частности, она возвращает значение **true**, если операция с классом **Punct_stream** выполнена успешно.

Теперь мы можем написать следующую программу.

```
int main()
    // Вводит текст и создает упорядоченный список всех слов
    // заданного текста, игнорируя знаки пунктуации и регистр,
    // а также удаляя из вывода дубликаты
{
    Punct_stream ps(cin);
```

```
    ps.whitespace(";:,.?!()\"{}<>/&$@#%^*|~"); // \" в строке
                                               // означает "
    ps.case_sensitive(false);

    cout << "Пожалуйста, введите слова\n";
    vector<string> vs;
    for(string word; ps>>word;)
        vs.push_back(word);     // Ввод слов
    sort(vs.begin(),vs.end()); // Сортировка в лексикографическом
                               // порядке
    for (int i=0; i<vs.size(); ++i) // Вывод словаря
         if (i==0 || vs[i]!=vs[i-1]) cout << vs[i] << '\n';
}
```

Этот код создает упорядоченный список введенных слов. Инструкция

```
if (i==0 || vs[i]!=vs[i-1])
```

удаляет дубликаты. Если в программу ввести слова

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
```

то результат ее работы будет выглядеть следующим образом:

```
about
and
are
complain
don't
kind
languages
of
only
people
that
there
two
use
```

Почему мы получили на выходе `don't`, а не `dont`? Потому что апостроф не вошел в список пробельных символов `whitespace()`.

Внимание: класс `Punct_stream` ведет себя во многом схоже с классом `istream`, но на самом деле им не является. Например, мы не можем проверить его состояние с помощью функции `rdstate()`, функция `eof()` не определена, и нет даже оператора `>>`, который вводит целые числа. Очень важно то, что мы не можем передать объект класса `Punct_stream` в качестве аргумента функции, ожидающей поток `istream`. А можно ли определить класс `Punct_istream`, который был бы потоком `istream`? Можно, но у нас пока что нет достаточного опыта программирования, не освоены основы проектирования и неизвестны все возможности

языка (если впоследствии — гораздо позже — вы вернетесь к этой задаче, то сможете реализовать потоковые буфера на профессиональном уровне).

Легко ли вам читать исходный текст класса `Punct_stream`? Понятны ли вам объяснения? Могли бы вы самостоятельно написать такую программу? Еще несколько дней назад вы были новичком и честно закричали бы "Нет, нет! Никогда!" или даже "Нет, нет! Вы что, с ума сошли?!" Цель данного примера заключается в следующем:

- показать реальную задачу и способ ее решения;
- доказать, что задачу можно решить с помощью вполне доступных средств;
- предоставить простое в использовании решение простой задачи;
- продемонстрировать разницу между интерфейсом и реализацией.

Для того чтобы стать программистом, вы должны читать программы, причем не только приукрашенные решения учебных задач. Приведенный пример относится как раз к таким программам. Через несколько дней или недель вы без труда будете читать такие программы и видеть возможные пути их улучшения.

Этот пример можно сравнить с уроком, на котором учитель английского языка для иностранцев произносит выражения на сленге, чтобы показать его колорит и живость.

11.8. И еще много чего

Подробности ввода-вывода можно описывать бесконечно. Этот процесс ограничен лишь терпением слушателей. Например, мы не рассмотрели сложности, связанные с естественными языками. То, что в английском языке записывается как `12.35`, в большинстве европейских языков записывается как `12,35`. Естественно, стандартная библиотека C++ предоставляет возможности для устранения этих и многих других проблем, касающихся аспектов ввода-вывода, связанных с естественными языками. А как записать китайские иероглифы? Как сравнивать строки, записанные символами малайского языка? Ответы на эти вопросы существуют, но они выходят далеко за рамки нашей книги. Если вам потребуется более детальная информация, можете обратиться к более специализированным книгам (например, книге Лангера (Langer) *Standard C++ IOStreams and Locales* и книге Страуструпа (Stroustrup) *The C++ Programming Language*), а также к библиотечной и системной документации. Ищите ключевое слово *locale* (местная специфика); этот термин обычно применяется к функциональным возможностям для обработки различий между естественными языками.

Другим источником сложностей является буферизация; стандартные библиотечные потоки **iostream** основаны на концепции под названием **streambuf**. Для сложных задач, связанных с потоками **iostream**, при решении которых важна производительность или функциональность, без объектов класса **streambuf** обойтись нельзя. Если хотите определить собственный класс **iostream** или настроить объекты класса **iostream** для работы с новыми источниками данных, читайте главу 38 книги Страуструпа *The C++ Programming Language* или системную документацию.

При программировании на языке C++ вы можете обнаружить семейство стандартных функций ввода-вывода **printf()/scanf()**, определенных в языке C. В этом случае прочитайте разделы 27.6, Б.10.2 или прекрасный учебник Кернигана (Kernighan) и Ритчи (Ritchie) *The C Programming Language*, или же любой из бесчисленных источников информации в вебе. Каждый язык имеет собственные (и существенно отличающиеся в разных языках программирования) средства ввода-вывода; но в большинстве случаев они отражают (пусть и совершенно по-разному) основные концепции, изложенные в главах 10 и 11.

Стандартная библиотека ввода-вывода описана в приложении Б.

Вопросы, связанные с графическими интерфейсами пользователя, рассматриваются в главах 12–16.

Задание

1. Напишите программу с именем **Test_output.cpp**. Объявите в ней целочисленную переменную **birth_year** и присвойте ей год своего рождения.
2. Выведите переменную **birth_year** в десятичном, шестнадцатеричном и восьмеричном виде.
3. Укажите основание системы счисления для каждого выводимого числа.
4. Выровняли ли вы результаты по столбцам с помощью символа табуляции? Если нет, то сделайте это.
5. Теперь выведите свой возраст.
6. Были ли какие-то проблемы? Что произошло? Замените свой вывод десятичным.
7. Вернитесь к п. 2 и укажите основания используемой системы счисления для каждого вывода.
8. Попытайтесь прочитать данные как восьмеричные, шестнадцатеричные и т.д.

```
cin  >> a >>oct >> b >> hex >> c >> d;
cout << a << '\t'<< b << '\t'<< c << '\t'<< d << '\n' ;
```

Запустите программу со следующими входными данными:

```
1234 1234 1234 1234
```

Поясните полученные результаты.

9. Напишите программу, три раза выводящую на печать число `1234567.89`: сначала в формате `defaultfloat`, затем — в `fixed` и в `scientific`. Какой способ представления обеспечивает наибольшую точность? Почему?
10. Создайте простую таблицу, содержащую фамилию, имя, телефонный номер и адрес электронной почты хотя бы пяти ваших друзей. Поэкспериментируйте с разной шириной полей, пока не найдете приемлемый.

Контрольные вопросы

1. Почему ввод-вывод является сложной задачей для программиста?
2. Что означает выражение `<< hex`?
3. Какие шестнадцатеричные числа используются в компьютерных науках? Почему?
4. Перечислите несколько возможностей, которые вы хотели бы реализовать при форматировании вывода целых чисел.
5. Что такое манипулятор?
6. Назовите префикс десятичного, восьмеричного и шестнадцатеричного чисел.
7. Какой формат по умолчанию применяется при выводе чисел с плавающей точкой?
8. Что такое поле вывода?
9. Объясните, что делают функции `setprecision()` и `setw()`.
10. Для чего нужны разные режимы при открытии файлов?
11. Какие из перечисленных далее манипуляторов не являются "клейкими": `hex`, `scientific`, `setprecision`, `showbase`, `setw`?
12. Укажите разницу между символьным и бинарным вводом.
13. Приведите пример, демонстрирующий преимущество использования бинарного файла вместо текстового.
14. Приведите два примера, в которых может оказаться полезным класс `stringstream`.
15. Что такое позиция в файле?
16. Что произойдет, если позиция в файле будет установлена за его пределами?
17. Когда ввод строк предпочтительнее, чем ввод, ориентированный на тип?
18. Что делает функция `isalnum(c)`?

Термины

`defaultfloat`	бинарный	нестандартный разделитель
`fixed`	восьмеричный	позиционирование в файле
`noshowbase`	десятичный	регулярность
`scientific`	классификация символов	строковый ввод
`setprecision`	манипулятор	форматирование вывода
`showbase`	нерегулярность	шестнадцатеричный

Упражнения

1. Напишите программу, читающую текстовый файл и записывающую его содержимое в новый файл в нижнем регистре.
2. Напишите программу, которая для данного имени файла и слова выводит все строки файла, содержащие это слово, вместе с номерами строк. *Указание*: `getline()`.
3. Напишите программу, удаляющую из файла все гласные буквы. Например, фраза `Once upon a time!` принимает вид `nc pn tm!`. Удивительно часто результат остается вполне удобочитаемым; проверьте это на своих друзьях.
4. Напишите программу под названием `multi_input.cpp`, которая предлагает пользователю ввести несколько целых восьмеричных, десятичных и шестнадцатеричных чисел в любом сочетании, используя суффиксы `0` и `0x`; правильно интерпретируйте эти числа и преобразуйте их в десятичный вид. Ваша программа должна выводить на экран примерно такие результаты:

```
0x43   (шестнадцатеричное) превращается в десятичное  67
0123   (восьмеричное)      превращается в десятичное  83
65     (десятичное)        превращается в десятичное  65
```

5. Напишите программу, считывающую строки и выводящую категории каждого символа в соответствии с правилами, описанными в разделе 11.6. Помните, что один и тот же символ может относиться к разным категориям (например, `x` — это и буквенный, и буквенно-цифровой символ).
6. Напишите программу, заменяющую знаки пунктуации пробелами. Считайте таковыми точку (`.`), точку с запятой (`;`), запятую (`,`), вопросительный знак (`?`), дефис (`-`), одинарные кавычки (`'`). Не изменяйте символы, находящиеся между парами двойных кавычек (`"`). Например, строка `" - don't use the as-if rule."` должна принять вид `"  don t use the as if rule "`.
7. Модифицируйте программу из предыдущего упражнения, чтобы она заменяла сокращения `don't` словами `do not`, `can't` — `cannot` и т.д.;

дефисы внутри слов не трогайте (таким образом, мы получим строку `"  do not use the as-if rule "`); переведите все символы в нижний регистр.

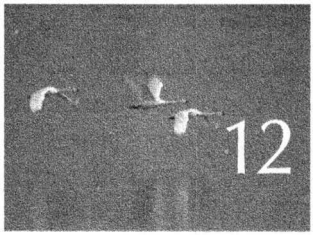
8. Используйте программу из предыдущего упражнения для создания словаря (в качестве альтернативы подходу, описанному в разделе 11.7). Примените ее к многостраничному текстовому файлу, проанализируйте результат и подумайте, можно ли улучшить эту программу, чтобы получить более качественный словарь.
9. Разделите программу ввода-вывода из раздела 11.3.2 на две: одна программа пусть конвертирует обычный текстовый файл в бинарный, а другая считывает бинарный файл и преобразует его в текстовый. Протестируйте эти программы, сравнивая текстовые файлы до и после преобразования в двоичный файл и обратно.
10. Напишите функцию `vector<string> split(const string& s)`, возвращающую вектор подстрок аргумента `s`, разделенных пробельными символами.
11. Напишите функцию `vector<string> split(const string& s, const string& w)`, возвращающую вектор подстрок аргумента `s`, разделенных пробельными символами, при условии, что в качестве пробельного символа могут использоваться как "обычные" пробельные символы, так и символы из строки `w`.
12. Измените порядок следования символов в текстовом файле. Например, строка `asdfghjkl` примет вид `lkjhgfdsa`. *Предупреждение*: не существует реально хорошего, переносимого и эффективного способа чтения файла в обратном направлении.
13. Измените порядок следования слов (определенных как строки, разделенные пробелами). Например, строка `Norwegian Blue parrot` примет вид `parrot Blue Norwegian`. Можете считать, что все строки из файла могут поместиться в памяти одновременно.
14. Напишите программу, считывающую текстовый файл и записывающую в другой файл количество символов каждой категории (см. раздел 11.6).
15. Напишите программу, считывающую из файла числа, разделенные пробельными символами, и выводящую в другой файл числа, используя научный формат и точность, равную восьми, в четырех полях по двадцать символов в строке.
16. Напишите программу, считывающую из файла числа, разделенные пробельными символами, и выводящую их в порядке возрастания по одному числу в строке. Каждое число должно быть записано только один раз, если обнаружится дубликат, то необходимо вывести

количество таких дубликатов в строке. Например, результат обработки ввода `7 5 5 7 3 117 5` должен иметь следующий вид:

```
3
5 3
7 2
117
```

Послесловие

Ввод и вывод сложны, поскольку вкусы и предпочтения у людей разные и не подчиняются простым правилам и математическим законам. Как программисты мы редко имеем право навязывать пользователям свои взгляды, а когда и имеем его, то должны сдерживаться и стараться предлагать простые альтернативы, выдержавшие проверку временем. Следовательно, мы должны смириться с определенными неудобствами ввода и вывода и стремиться, чтобы наши программы были как можно более простыми, — но не проще.



12

Модель вывода на экран

Сначала мир был черно-белым, и стал цветным только в 1930-х годах.
— Папаша Кэлвина (Calvin's dad)[1]

В главе описана модель вывода на экран (часть графического пользовательского интерфейса, отвечающая за вывод информации), приведены примеры ее использования, а также описаны такие фундаментальные понятия, как как координаты экрана, линии и цвет. Здесь рассмотрены классы **Line**, **Lines**, **Polygon**, **Axis** и **Text**, являющиеся подклассами **Shape**. Объект класса **Shape** представляет собой объект в памяти, которым мы можем управлять и который мы можем выводить на экран. В следующих двух главах мы детальнее исследуем эти классы. В главе 13 будет рассмотрена их реализация, а в главе 14 — вопросы проектирования.

[1] Папаша Кэлвина — персонаж популярного в США комикса Calvin and Hobbes. — *Примеч. ред.*

12.1. Почему графика?

Почему мы посвящаем четыре главы графике и одну главу — графическим пользовательским интерфейсам (graphical user interface — GUI)? Все же наша книга о программировании, а не о графике? Имеется масса интересных тем, связанных с программированием, которые мы не обсуждаем и в лучшем случае можем сделать лишь краткий обзор вопросов, касающихся графики. Итак, почему графика? В основном потому, что графика — это предмет, позволяющий исследовать важные вопросы, относящиеся к проектированию программного обеспечения, программирования, а также к инструментам программирования.

- *Графика полезна.* Программирование как тема намного шире графики, а программное обеспечение намного содержательнее, чем проблемы манипулирования кодом с помощью графического пользовательского интерфейса. Однако во многих областях хорошая графика играет существенную или очень важную роль. Например, мы не могли бы и мечтать об изучении проблем, связанных с научными вычислениями, анализом данных или просто с количественными исследованиями, не имея возможности изображать результаты с помощью графики. Простые (но содержательные) примеры использования графики для представления данных приведены в главе 15.
- *Графика доставляет удовольствие.* Это одна из редких сфер деятельности, связанных с вычислениями, в которых результат выполнения фрагмента кода столь нагляден и (после устранения ошибок) приятен. С графикой приятно работать даже тогда, когда она не приносит ощутимой пользы!

- *Графические программы очень интересны.* Обучение программированию подразумевает чтение множества программ, чтобы получить представление о хорошем коде. Аналогично, для того чтобы хорошо овладеть английским языком, необходимо прочитать много книг, журналов и газет. Благодаря прямой зависимости между тем, что мы видим на экране, и тем, что написано в программе, простой графический код легче для понимания, чем большинство программ, сравнимых с ним по сложности. В этой главе мы начнем читать графические коды практически сразу после введения, а в главе 13 покажем, как написать эти коды за несколько часов.
- *Графика — богатый источник примеров проектирования.* Разработать и реализовать хорошую графику и библиотеку графического пользовательского интерфейса в действительности достаточно трудно. Графика — очень богатый источник конкретных и практических примеров проектных решений и методов проектирования. Некоторые из наиболее полезных методов проектирования классов и функций, разделения программного обеспечения на слои (абстракций) и создания библиотек можно проиллюстрировать с помощью относительно небольшого количества программ, реализующих графический вывод данных и графический пользовательский интерфейс.
- *Графика является хорошим введением в объектно-ориентированное программирования и языковые средства его поддержки.* Несмотря на то что молва утверждает обратное, объектно-ориентированное программирование было изобретено вовсе не для того, чтобы появилась возможность работать с графическими программами (подробнее об этом речь пойдет в главе 22), но этот подход практически сразу же был применен при реализации графики, которая стала одним из наиболее ярких примеров, демонстрирующих преимущество объектно-ориентированного проектирования.
- *Некоторые ключевые концепции графики нетривиальны.* Поэтому они заслуживают тщательного изучения. Эти вопросы нельзя пускать на самотек, надеясь на инициативность, пытливость и терпение читателей. Если не показать, как работают графические программы, читатели станут относиться к ним, как к "черным ящикам", а это прямо противоречит основным целям нашей книги.

12.2. Модель вывода на дисплей

Библиотека ввода-вывода `iostream` ориентирована на чтение и запись потоков символов. Единственными символами, непосредственно связанными с понятием графической позиции, являются символы перехода на

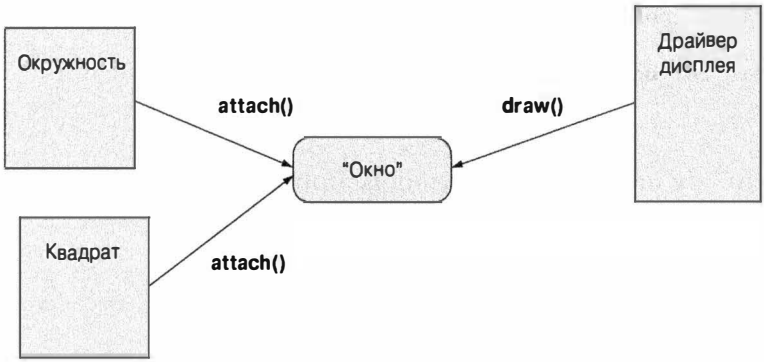
новую строку и табуляции. Впрочем, в одномерный поток символов можно внедрить понятия цвета и двумерных позиций. Именно так устроены языки разметки, такие как Troff, TeX, Word, HTML и XML (а также связанные с ними графические пакеты). Рассмотрим пример.

```
<hr>
<h2>
Организация
</h2>
Этот список состоит из трех частей:
<ul>
     <li><b>Предложения</b>, пронумерованные EPddd, ...</li>
     <li><b>Пункты</b>, пронумерованные EIddd, ...</li>
     <li><b>Предположения</b>, пронумерованные ESddd, ...</li>
</ul>
<p>Мы пытаемся ...
<p>
```

Это фрагмент кода на языке HTML, в котором указаны заголовок (<h2>...</h2>), ненумерованный список (<ul>...) с пунктами (<li>...) и абзацы (<p>). Мы удалили большую часть текста, поскольку он в данном случае не играет особой роли. Главное — мы можем выразить понятия разметки в обычном тексте, а связи между тем, как записаны символы и как они появляются на экране, являются опосредованными и управляются программой, интерпретирующей команды разметки. Такой метод очень прост и чрезвычайно полезен (все, что вы сейчас читаете, было создано с его помощью), но имеет некоторые ограничения.

В данной и следующих четырех главах мы описываем альтернативный подход: понятие графики и графического пользовательского интерфейса, непосредственно связанных с экраном компьютера. Основные концепции — такие, как координаты, линии, прямоугольники и окружности — по своей сути являются графическими (и двумерными, адаптированными к прямоугольному экрану компьютера). С точки зрения программирования цель этих понятий заключается в создании прямой зависимости между объектами памяти и образами на экране.

Базовая модель представляет собой следующее. Мы составляем объекты из элементарных объектов, предоставляемых графической системой, таких как линии. Затем мы "связываем" эти графические объекты с объектом окна, представляющим наш физический экран. Затем программа, которую мы можем рассматривать как сам дисплей, "драйвер (или проще — "движок") дисплея", "графическую библиотеку", "библиотеку графического интерфейса" и даже (шутка) как "маленького гномика, рисующего на обратной стороне экрана", получает объекты, которые мы связываем с окном, и выводит их на экране.



Драйвер дисплея рисует линии на экране, размещает на нем текстовые строки, закрашивает его области и т.д. Для простоты мы используем термины "графическая библиотека" и даже "система", несмотря на то, что библиотека графического пользовательского интерфейса может значительно большее, чем просто рисовать объекты. Точно так же, как наш код может заставить библиотеку графического пользовательского интерфейса делать вместо нас большую часть работы, сама библиотека делегирует большинство своих заданий операционной системе.

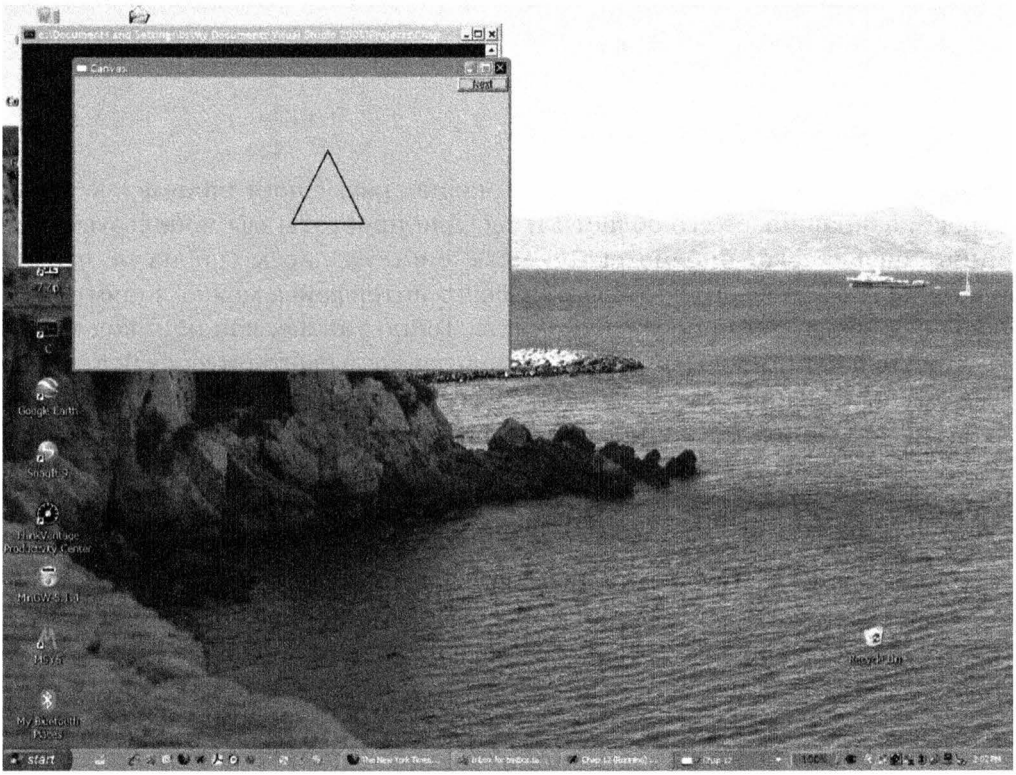
12.3. Первый пример

Наша задача — определить классы, из которых можно создавать объекты для вывода на экран. Например, мы можем нарисовать график в виде ломаной линии. Вот очень простая программа, выполняющая эту работу:

```
#include "Simple_window.h"      // Обеспечивает доступ к
                                // оконной библиотеке
#include "Graph.h"              // Обеспечивает доступ к
                                // графической библиотеке
int main()
{
    using namespace Graph_lib;  // Графические средства
                                // находятся в Graph_lib
    Point tl{100,100};          // Задаем левый верхний угол окна

    Simple_window win{tl,600,400,"Canvas"}; // Простое окно

    Polygon poly;               // Создаем фигуру (многоугольник)

    poly.add(Point{300,200});   // Добавляем точку
    poly.add(Point{350,100});   // Добавляем вторую точку
    poly.add(Point{400,200});   // Добавляем третью точку

    poly.set_color(Color::red); // Уточняем свойства poly
```

```
    win.attach (poly);          // Связываем poly с окном

    win.wait_for_button();      // Передаем управление драйверу
                                // дисплея
}
```

Запустив эту программу, мы увидим примерно такую картину.



Пройдемся по строкам программы и посмотрим, как она работает. Сначала в программу включаются заголовочные файлы библиотеки графического интерфейса.

```
#include "Simple_window.h"      // Обеспечивает доступ к
                                // оконной библиотеке
#include "Graph.h"              // Обеспечивает доступ к
                                // графической библиотеке
```

Затем в функции **main()** мы сообщаем компьютеру, что средства нашей графической библиотеки находятся в пространстве имен **Graph_lib**.

```
using namespace Graph_lib;  // Графические средства
                            // находятся в Graph_lib
```

После этого мы определяем точку, которая будет использоваться в качестве левого верхнего угла нашего окна.

```
Point tl{100,100};          // Задаем левый верхний угол окна
```

Затем создаем окно на экране.

```
Simple_window win{tl,600,400,"Canvas"}; // Простое окно
```

Здесь мы используем класс **Simple_window**, представляющий окно в нашей библиотеке **Graph_lib**. Конкретный объект класса **Simple_window** носит имя **win**; иначе говоря, **win** — это переменная класса **Simple_window**. Список инициализации объекта **win** начинается с точки **tl**, которая будет использована в качестве левого верхнего угла, за ней следуют числа 600 и 400. Это, соответственно, ширина и высота окна, измеренные в пикселях. Мы объясним их смысл позднее, а пока лишь укажем, что они позволяют задать прямоугольник с заданными шириной и высотой. Строка **Canvas** используется для метки окна. Если присмотреться, то можно увидеть слово **Canvas** в левом верхнем углу рамки окна.

Далее помещаем в окно некоторый объект.

```
Polygon poly;               // Создаем фигуру (многоугольник)

poly.add(Point{300,200});   // Добавляем точку
poly.add(Point{350,100});   // Добавляем вторую точку
poly.add(Point{400,200});   // Добавляем третью точку
```

Мы определяем многоугольник **poly**, а затем добавляем к нему точки. В нашей графической библиотеке объекты класса **Polygon** создаются пустыми; затем мы можем добавить в них любое количество точек, какое пожелаем. Поскольку мы добавили три точки, то получили треугольник. Точки представляют собой простые пары чисел, задающих горизонтальные и вертикальные координаты x и y в окне.

Для того чтобы продемонстрировать такую возможность, мы сделали стороны многоугольника красными.

```
poly.set_color(Color::red); // Уточняем свойства poly
```

В заключение мы связываем объект **poly** с нашим окном **win**.

```
win.attach (poly);          // Связываем poly с окном
```

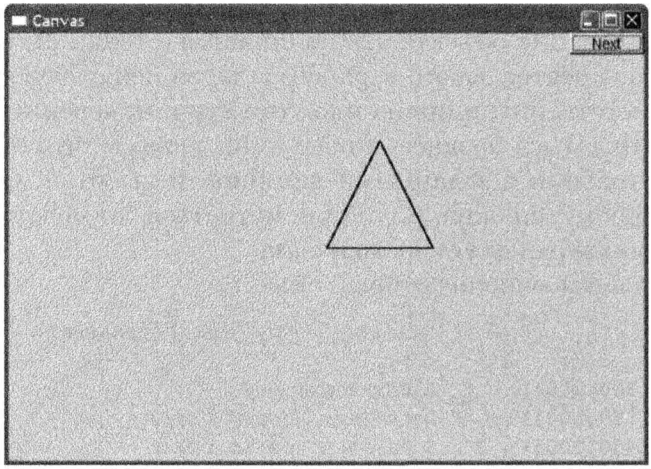
Легко заметить, что на экране пока не происходит вообще ничего. Мы создали окно (точнее, объект класса **Simple_window**) и многоугольник (объект **poly**), окрасили многоугольник в красный цвет (**Color::red**) и связали его с окном **win**, но мы не дали команду отобразить это окно на экране. Это делает последняя строка в программе.

```
win.wait_for_button();      // Передаем управление драйверу
                            // дисплея
```

Для того чтобы система графического пользовательского интерфейса отображала объекты на экране, мы передали управление системе. Эту

задачу выполняет функция `wait_for_button()`, которая заставляет систему ждать, когда вы щелкнете на кнопке `Next` в окне `Simple_window`. Это позволяет нам увидеть окно прежде, чем программа завершит свою работу и окно исчезнет. Когда вы щелкнете на кнопке, программа прекратит работу, закрыв окно.

Наше окно выглядит так.



Обратите внимание на то, что мы немного схитрили. Откуда взялась кнопка `Next`? Мы встроили ее в классе `Simple_window`. В главе 16 мы перейдем от класса `Simple_window` к "обычному" классу `Window`, в котором нет скрытых возможностей, и покажем, как написать собственный код, позволяющий управлять взаимодействием с окном.

В следующих трех главах мы будем просто использовать кнопку `Next` для перехода от одного дисплея к другому для отображения информации, связанной с разными этапами некоего процесса ("кадр за кадром").

Вы настолько привыкли к тому, что вокруг каждого окна операционная система автоматически рисует рамку, что уже не замечаете ее. Рисунки в этой и следующих главах созданы с помощью системы Microsoft Windows, поэтому в правом верхнем углу каждого окна расположены три кнопки. Они могут быть полезными, если ваша программа зашла в тупик (а это в ходе отладки иногда случается), вы можете прекратить ее выполнение, щелкнув на кнопке с пиктограммой `x`. Если вы запустите программу в другой операционной системе, рамка изменится. Наш вклад в оформление рамки заключается лишь в создании метки (в данном случае — `Canvas`).

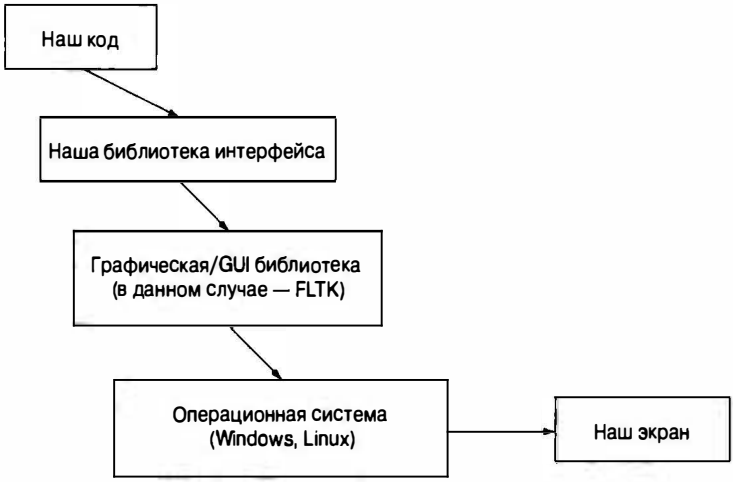
12.4. Использование библиотеки графического пользовательского интерфейса

В этой книге мы не используем возможности графики и графического пользовательского интерфейса конкретных операционных систем непосредственно. Это ограничило бы использование наших программ одной операционной системой и вынудило бы учитывать массу запутанных деталей. Как и для ввода-вывода текстов, чтобы упростить наши программы, мы будем использовать библиотеку, сглаживающую различия между операционными системами, устройствами ввода-вывода и т.д. К сожалению, язык C++ не имеет стандартной библиотеки графического пользовательского интерфейса, аналогично библиотеке стандартных потоков ввода-вывода, поэтому мы используем одну из многих доступных библиотек. Поскольку нам не хотелось бы жестко привязываться ни к одной из этих библиотек и погружаться в тонкости их устройства, мы используем набор простых интерфейсных классов, которые можно реализовать с помощью нескольких сотен строк кода и которые можно найти практически в любой библиотеке графического пользовательского интерфейса.

Набор инструментов для создания графического пользовательского интерфейса, который мы используем в нашей книге, называется FLTK (Fast Light Tool Kit, произносится как "full tick") и находится по адресу `www.fltk.org`. Наш код переносим на любую платформу, где выполняется код библиотеки (Windows, Unix, Mac, Linux и др.). Наши интерфейсные классы можно было бы реализовать с помощью другой библиотеки, так что программы стали бы еще более переносимыми.

Модель программирования, представленная в наших интерфейсных классах, намного проще, чем предлагает обычный набор инструментальных средств. Например, наша полная библиотека графических средств и графического пользовательского интерфейса содержит около 600 строк кода на языке C++, в то время как чрезвычайно немногословная документация библиотеки FLTK содержит 370 страниц. Вы можете загрузить ее с веб-сайта `www.fltk.org`, но мы пока не рекомендуем делать это. Пока что можно вполне работать без такого высокого уровня детализации. Общие идеи, представленные в главах 12–16, можно использовать с любым популярным инструментарием графического пользовательского интерфейса. Разумеется, мы объясним, как наши интерфейсные классы связаны с библиотекой FLTK, так что, если захотите, вы в конечном итоге сможете применять эту библиотеку (и подобные ей) непосредственно.

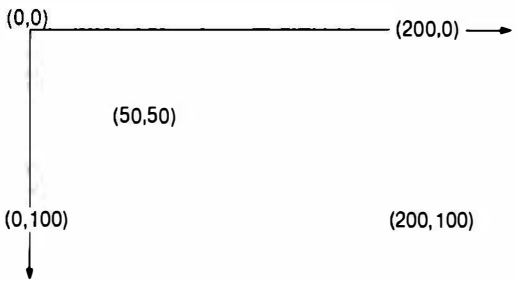
Части нашего "графического мира" можно представить следующим образом.



Наши классы интерфейса образуют простую и расширяемую пользователями основу для создания двумерных фигур с ограниченной поддержкой цвета. Для управления этими классами мы предоставляем простой механизм графического пользовательского интерфейса, основанный на функциях обратного вызова, запускаемых кнопками и другими элементами управления, расположенными на экране (подробнее они будут рассмотрены в главе 16).

12.5. Координаты

Экран компьютера — это прямоугольная область, составленная из пикселей. Пиксель — это маленькая цветная точка. Чаще всего экран в программе моделируется как прямоугольник пикселей. Каждый пиксель имеет горизонтальную координату x и вертикальную координату y. Начальная координата x равна нулю и соответствует крайнему слева пикселю. Ось x направлена вправо к крайнему справа пикселю. Начальная координата y равна нулю и соответствует первому сверху пикселю. Ось y направлена вниз к первому снизу пикселю.



Пожалуйста, обратите внимание на то, что координаты *y* возрастают по направлению вниз. Математикам это покажется странным, но экраны (и окна, возникающие на экране) могут иметь разные размеры, и верхняя левая точка — это единственное, что у них есть общего.

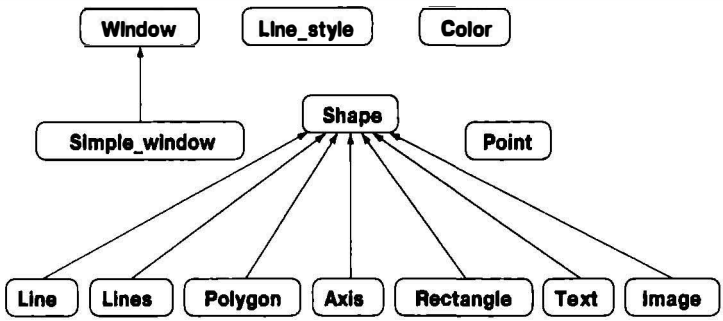
Количество пикселей зависит от экрана: самыми распространенными являются 1024×768, 1280×1024, 1450×1050 и 1600×1200. В контексте взаимодействия с компьютером окно рассматривается как прямоугольная область экрана, имеющая определенное предназначение и управляемая программой. Окно размечается точно так же, как и экран. В принципе, окно можно рассматривать как маленький экран. Например, если программа содержит инструкцию

```
Simple_window win{tl,600,400,"Canvas"};
```

то это значит, что мы хотим создать прямоугольную область, ширина которой равна 600 пикселям, а высота — 400, чтобы адресовать ее от 0 до 599 слева направо и от 0 до 399 сверху вниз. Область окна, которую можно изобразить на экране, называется *канвой* (canvas). Область 600×400 считается "внутренней" областью окна, т.е. областью, расположенной в кадре, предоставленном системой; она не содержит строки заголовка, кнопок выхода и пр.

12.6. Класс Shape

Наш основной набор инструментов для рисования на экране состоит из двенадцати классов.



Стрелка означает, что класс, из которого она выходит, может быть использован там, где требуется класс, на который она указывает. Например, класс `Polygon` может быть использован там, где требуется класс `Shape`; иначе говоря, класс `Polygon` является разновидностью класса `Shape`.

Сначала опишем использование следующих классов:

- `Simple_window`, `Window`;
- `Shape`, `Text`, `Polygon`, `Line`, `Lines`, `Rectangle`, `Function` и т.д.;

- **Color**, **Line_style**, **Point**;
- **Axis**.

Позднее (в главе 16) добавим к ним классы графического пользовательского интерфейса:

- **Button**, **In_box**, **Menu** и т.д.

К этому набору можно было бы более или менее легко добавить много других классов, например

- **Spline**, **Grid**, **Block_chart**, **Pie_chart** и т.д.

Однако описание полного набора инструментов для создания графического пользовательского интерфейса со всеми его возможностями выходит за рамки нашей книги.

12.7. Использование примитивов Shape

В этом разделе мы рассмотрим некоторые элементарные примитивы нашей графической библиотеки: **Simple_window**, **Window**, **Shape**, **Text**, **Polygon**, **Line**, **Lines**, **Rectangle**, **Color**, **Line_style**, **Point**, **Axis**. Цель этого обзора — дать читателям представление о том, что можно сделать с помощью этих средств, без углубления в детали реализации этих классов. Каждый из этих классов будет подробно изучен в следующих главах.

Начнем с простой программы, объясняя ее строчка за строчкой и демонстрируя результаты ее работы на экране. Когда вы запустите эту программу, то увидите, как изменяется изображение при добавлении новых и модификации существующих фигур, расположенных в окне. В принципе, такой анализ напоминает анимацию.

12.7.1. Заголовочные файлы и функция main

Сначала включим заголовочные файлы, в которых определены графические классы и класс графического пользовательского интерфейса.

```
#include "Window.h"          // Обычное окно
#include "Graph.h"
```

или

```
#include "Simple_window.h" // Если нам нужна кнопка Next
#include "Graph.h"
```

Как вы, возможно, уже догадались, файл **Window.h** содержит средства, связанные с окнами, а файл **Graph.h** — инструменты, связанные с рисованием фигур (включая текст) в окне. Эти средства определены в пространстве имен **Graph_lib**. Для упрощения обозначений мы используем директиву **using namespace**, чтобы получить доступ к именам из пространства **Graph_lib**.

```
using namespace Graph_lib;
```

Как обычно, функция `main()` содержит код, который мы хотим выполнить (прямо или косвенно), а также обработку исключений.

```
int main ()
try
{
    // ... Здесь находится наш код ...
}
catch(exception& e) {
    // Сообщения об ошибках
    return 1;
}
catch(...) {
    // Другие сообщения об ошибках
    return 2;
}
```

Для компиляции данной функции `main()` нам требуется, чтобы был определен класс `exception`. Для этого можно, как обычно, включить заголовочный файл `std_lib_facilities.h` или работать непосредственно со стандартными заголовочными файлами и включить заголовочный файл `<stdexcept>`.

12.7.2. Почти пустое окно

Здесь мы не будем обсуждать обработку ошибок (см. главу 5, в частности раздел 5.6.3), а сразу перейдем к описанию графики в функции `main()`:

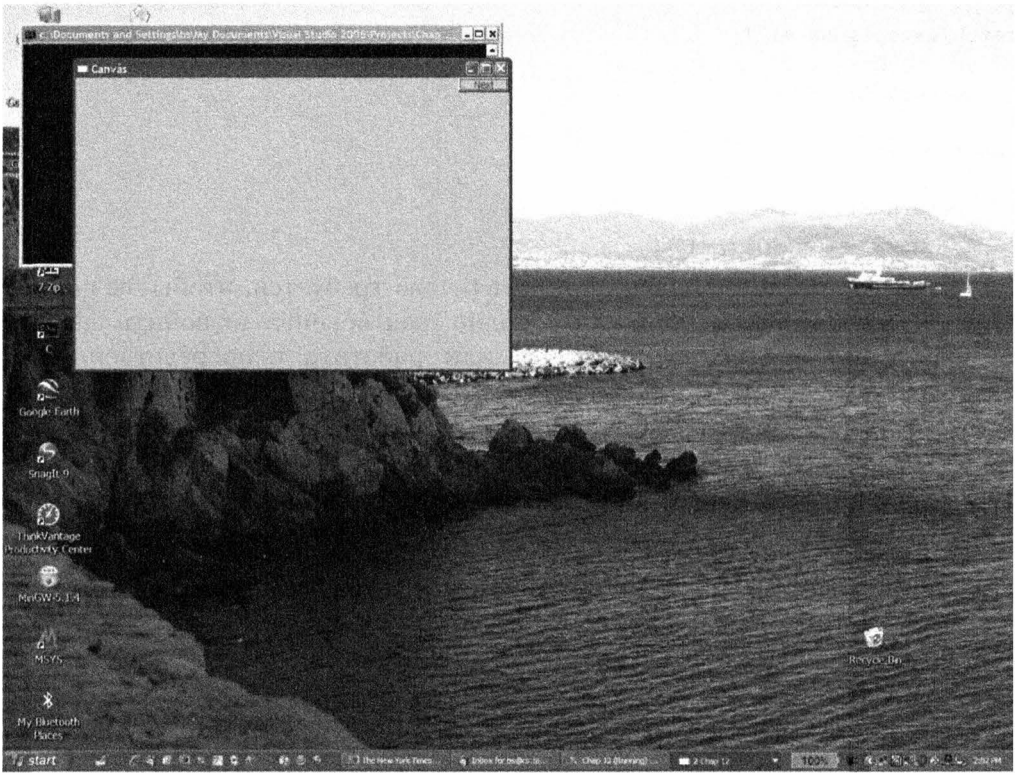
```
Point tl{100,100};      // Левый верхний угол нашего окна

Simple_window win{tl,600,400,"Canvas"};
    // Координаты окна tl задают положение левого верхнего угла
    // Размер окна: 600×400
    // Заголовок: Canvas
win.wait_for_button(); // Вывести окно
```

Этот фрагмент программы создает объект класса `Simple_window`, т.е. окно с кнопкой `Next`, и выводит его на экран. Очевидно, что для создания объекта класса `Simple_window` нам необходимо включить в программу заголовочный файл `Simple_window.h`, а не `Window.h`. Здесь мы указываем, в каком месте экрана должно появиться окно: его левый верхний угол должен находиться в точке `Point{100,100}`. Это близко, но не слишком близко к левому верхнему углу экрана. Очевидно, что `Point` — это класс, конструктор которого получает пару целых чисел и интерпретирует их как пару координат (x, y). Эту инструкцию можно было бы написать так:

```
Simple_window win{Point{100,100},600,400,"Canvas"};
```

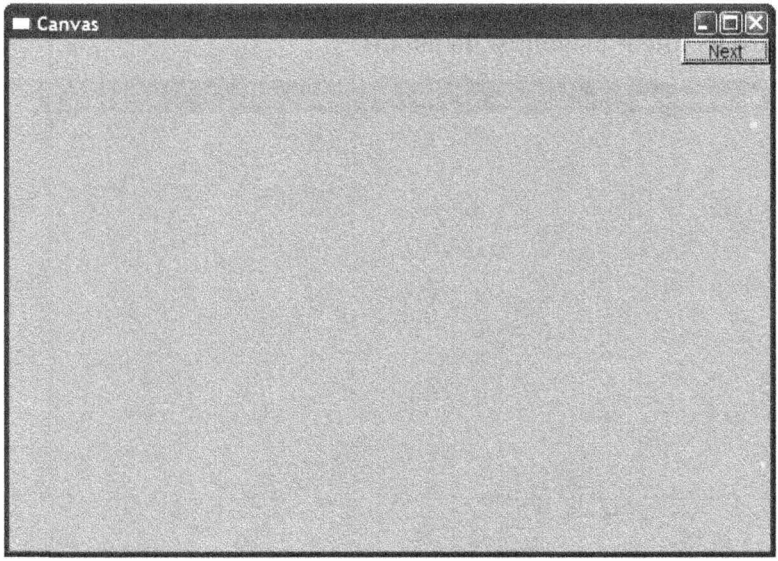
Однако мы хотим использовать точку (100, 100) несколько раз, поэтому удобнее присвоить ей имя. Число 600 — это ширина окна, 400 — его высота, а строка `"Canvas"` — метка, которую мы хотим поместить на рамке окна.

Для того чтобы окно действительно было нарисовано на экране, мы должны передать управление системе графического пользовательского интерфейса. Для этого вызываем функцию `win.wait_for_button()`. Результат показан на следующем рисунке.



На фоне окна мы видим экран ноутбука (на всякий случай очищенный от лишних пиктограмм). Для любопытных людей, интересующихся деталями, не относящимися к делу, сообщаю, что эту фотографию я сделал, стоя возле библиотеки Пикассо в Антибе и глядя через залив на Ниццу. Черное консольное окно, частично скрытое нашим окном, автоматически открывается при запуске нашей программы. Консольное окно выглядит некрасиво, но позволяет эффективно закрыть наше окно при отладке программы, если мы попадем в бесконечный цикл и не сможем выйти из программы обычным способом. Если внимательно присмотреться, то можно заметить, что мы использовали компилятор Microsoft C++, но вместо него можно было бы использовать любой другой компилятор (например, Borland или GNU).

Для дальнейшей демонстрации нашей программы мы удалили все лишнее, оставив только само окно.



Реальный размер окна (в сантиметрах) зависит от разрешающей способности вашего экрана. На одних экранах размер пикселей больше, чем на других.

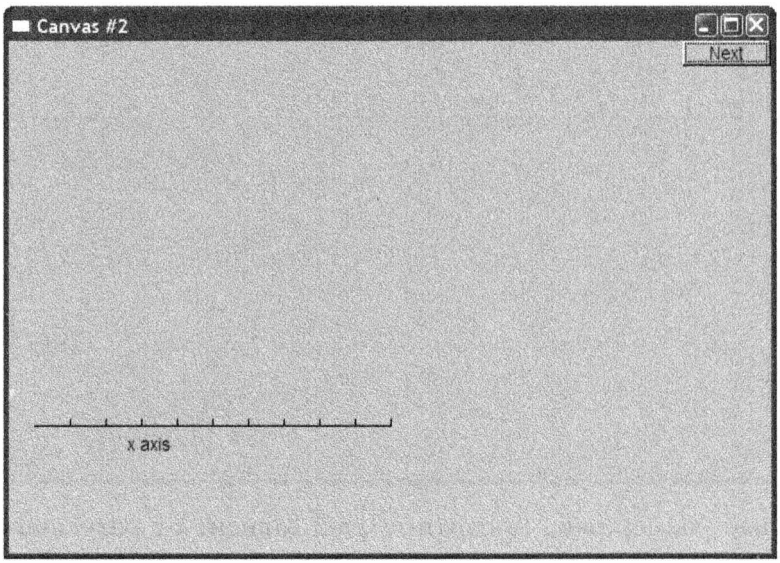
12.7.3. Оси координат

Почти пустое окно не очень интересно, поэтому попробуем добавить в него какую-нибудь информацию. Что бы мы хотели изобразить? Давайте вспомним, что графика — это не только игры и развлечения, и сделаем что-нибудь серьезное и сложное, например оси координат. График без осей координат, как правило, ужасен. Невозможно себе представить, какие данные можно изобразить, не пользуясь осями координат. Может быть, вам удастся оправдать их отсутствие в сопроводительной документации к программе, но намного лучше добавить оси координат: люди часто не читают объяснений, а хорошее графическое представление обычно не нуждается в комментариях. Итак, нам необходимы оси координат.

```
Axis xa{Axis::x, Point(20,300), 280, 10, "x axis"}; // Создаем объект Axis
    // Класс Axis – разновидность класса Shape
    // Axis::x означает горизонтальную ось
    // Начало оси – в точке (20,300)
    // Длина оси – 280 пикселей
    // Ось содержит 10 делений
    // "Ось x" – метка оси
win.attach(xa);             // Связываем объект xa с окном win
```

```
win.set_label("Canvas #2"); // Изменяем метку окна
win.wait_for_button();      // Изобразить!
```

Последовательность действий такова: создаем объект класса **Axis**, добавляем его в окне и выводим на экран.



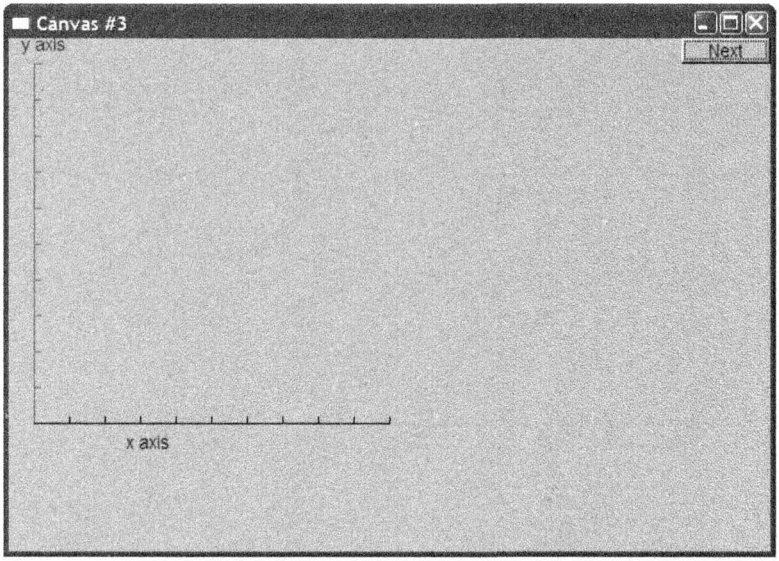
Как видим, параметр **Axis::x** определяет горизонтальную линию. Кроме того, ось имеет десять делений и метку "**x axis**". Как правило, метка поясняет, что представляют собой ось и ее деления. Естественно, ось *x* следует выбирать где-то ближе к нижнему краю окна. В реальной программе мы обозначили бы ширину и высоту какими-нибудь символическими константами, чтобы придать фразе "где-то ближе к нижнему краю окна" конкретный смысл, например, выраженный в виде инструкции **y_max-bottom_margin**, и не использовали бы "магические константы", такие как **300** (см. разделы 4.3.1 и 15.6.2).

Для того чтобы идентифицировать окно вывода, мы изменили его метку на строку **"Canvas #2"** с помощью функции-члена **set_label()** класса **Window**.

Теперь добавим ось *y*.

```
Axis ya{Axis::y, Point{20,300}, 280, 10, "y axis"};
ya.set_color(Color::cyan);              // Выбираем цвет
ya.label.set_color(Color::dark_red);    // Выбираем цвет текста
win.attach(ya);
win.set_label("Canvas #3");
win.wait_for_button();                  // Вывод на экран
```

Просто для того, чтобы продемонстрировать некоторые возможности, мы окрасили ось *y* в голубой цвет, а метку сделали темно-красной.



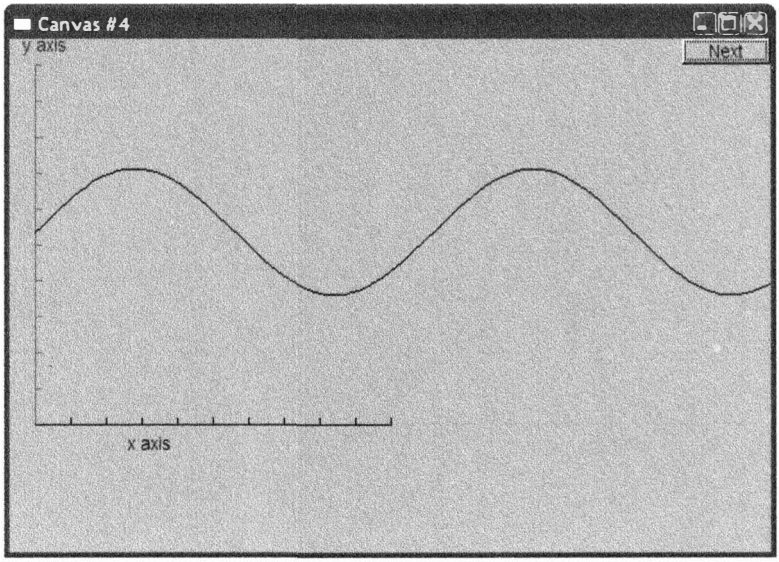
На самом деле мы не считаем удачной идею присваивать разные цвета осям *x* и *y*. Мы просто хотели показать, как можно задать цвет фигуры и ее отдельных элементов. Использование большого количества цветов не всегда оправдано. В частности, новички часто злоупотребляют раскраской графиков, демонстрируя избыток энтузиазма и недостаток вкуса.

12.7.4. График функции

Что дальше? Теперь у нас есть окно с координатными осями, и кажется естественным нарисовать на нем график функции. Создадим объект, представляющий график синуса, и свяжем его с окном.

```
Function sine{sin,0,100,Point{20,150},1000,50,50};
    // График синуса
    // Рисуем sin() в диапазоне [0,100) от (0,0) до (20,150),
    // Используя 1000 точек; для масштабирования координаты
    // Умножаются на 50 каждая

win.attach(sine);
win.set_label("Canvas #4");
win.wait_for_button();
```

Здесь объект класса **Function** с именем **sine** рисует график синуса, используя стандартную библиотечную функцию **sin()**. Детали построения графиков функций описываются в разделе 15.3. А пока отметим, что для построения такого графика необходимо выбрать отправную точку (объект класса **Point**) и диапазон изменения входных значений, а также указать информацию о масштабировании, чтобы график поместился в окне.



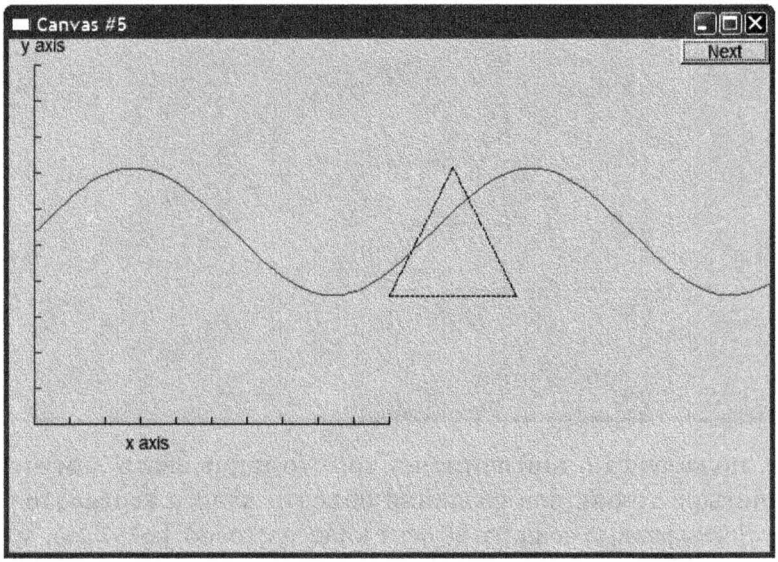
Теперь кривая будет заканчиваться на краю окна. Точки, изображенные за пределами окна, игнорируются системой графического пользовательского интерфейса и остаются невидимыми.

12.7.5. Многоугольники

График функции является примером представления данных. Более подробно эта тема исследуется в главе 15. Однако в окне можно рисовать и другие объекты, например геометрические фигуры. Эти фигуры используются для создания графических иллюстраций, рисования элементов пользовательского интерфейса (таких, как кнопки) и просто для украшения результатов работы программы. Объект класса **Polygon** задается последовательностью точек, соединенных линиями. Первая линия соединяет первую точку со второй, вторая линия соединяет вторую точку с третьей, а последняя линия соединяет последнюю точку с первой.

```
sine.set_color(Color::blue); // Мы изменили цвет графика синуса

Polygon poly;                // Polygon - разновидность Shape
poly.add(Point{300,200});    // Три точки образуют треугольник
poly.add(Point{350,100});
poly.add(Point{400,200});

poly.set_color(Color::red);
poly.set_style(Line_style::dash);
win.attach(poly);
win.set_label("Canvas #5");
win.wait_for_button();
```

На этот раз мы изменили цвет графика синуса (**sine**) просто для того, чтобы показать, как это делается. Затем мы добавили треугольник, так же как в первом примере из раздела 12.3, представляющий собой разновидность многоугольника. Здесь мы также задали цвет и стиль. Линии в классе **Polygon** имеют стиль. По умолчанию они сплошные, но их можно сделать пунктирными, точечными и т.п. (подробнее об этом — в разделе 13.5). Так мы получаем следующий результат.



12.7.6. Прямоугольник

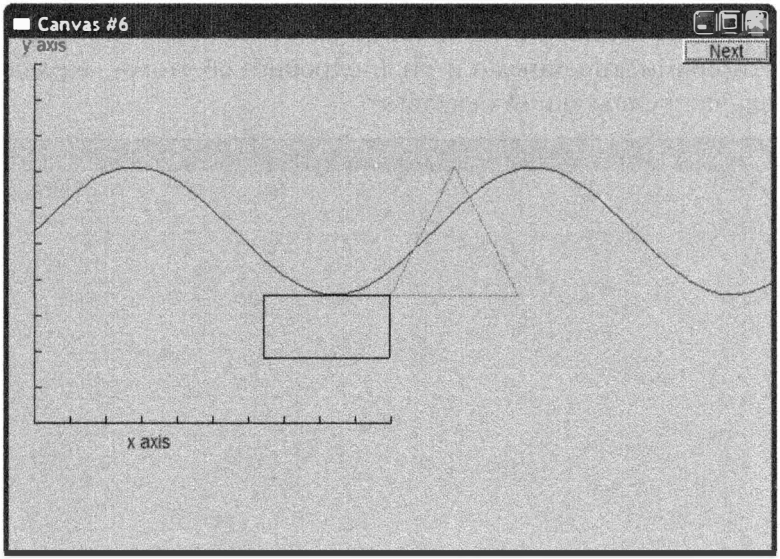
Экран — это прямоугольник, окно — это прямоугольник и лист бумаги — это тоже прямоугольник. Фактически прямоугольниками (или прямоугольниками с закругленными углами) является огромное количество фигур, потому что это одна из простейших фигур. Его легко описать (координаты левого верхнего угла, ширина и высота или координаты левого верхнего и правого нижнего углов), как в нем, так и за его пределами легко задать местоположение точки. Кроме того, его можно легко и быстро нарисовать на экране.

По этой причине большинство высокоуровневых графических библиотек эффективнее работают с прямоугольниками, чем с любыми другими замкнутыми фигурами. Следовательно, целесообразно описать прямоугольник с помощью отдельного класса **Rectangle**, отдельного от класса **Polygon**. Класс **Rectangle** характеризуется координатами верхнего левого угла, шириной и высотой.

```
Rectangle r{Point{200,200},100,50}; // Левый верхний угол, ширина, высота
win.attach(r);
```

```
win.set_label("Canvas #6");
win.wait_for_button();
```

Этот фрагмент дает нам следующее окно.



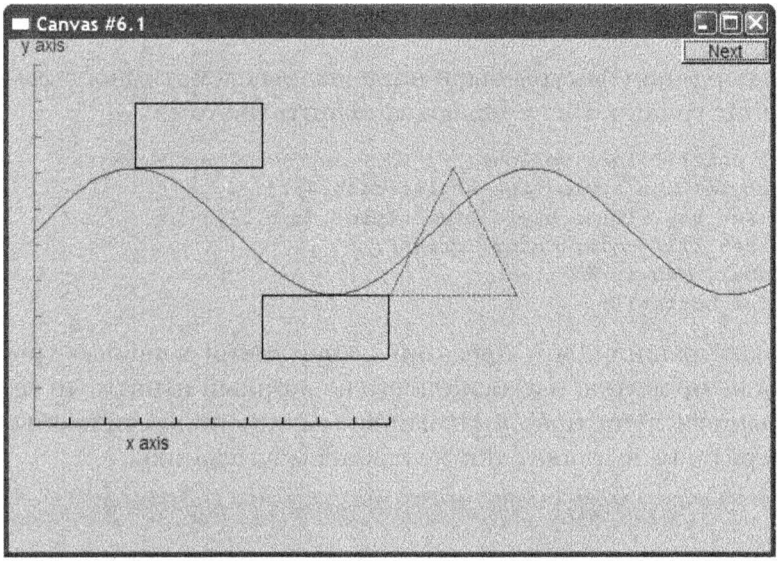
Обратите, пожалуйста, внимание на то, что нарисовать ломаную, соединяющую четыре точки, для создания объекта класса **Rectangle** еще недостаточно. Легко можно создать объект класса **Closed_polyline**, который на экране выглядит как объект класса **Rectangle** (можно даже создать объект класса **Open_polyline**, который будет выглядеть точно так же).

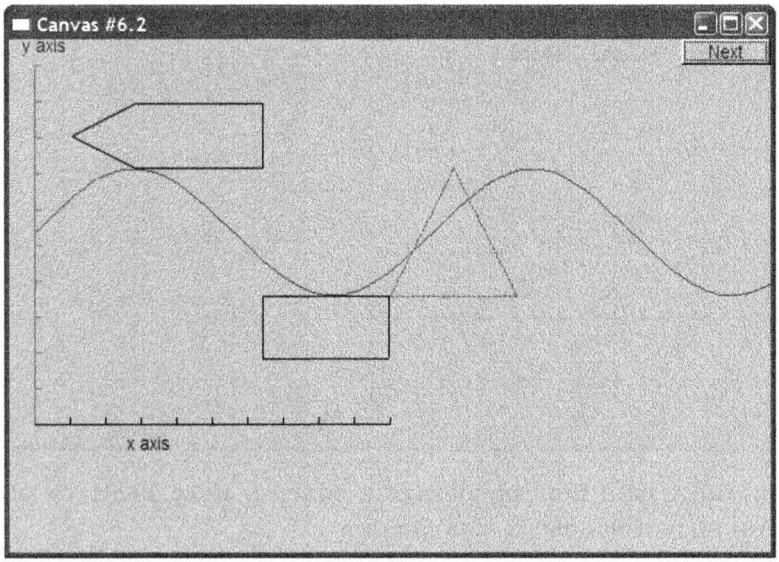
```
Closed_polyline poly_rect;
poly_rect.add(Point{100,50});
poly_rect.add(Point{200,50});
poly_rect.add(Point{200,100});
poly_rect.add(Point{100,100});
win.attach(poly_rect);
```

Фактически *изображение* объекта **poly_rect** на экране *является* прямоугольником. Однако объект класса **poly_rect** в памяти не является объектом класса **Rectangle** и ничего не "знает" о прямоугольниках. Проще всего это доказать, попытавшись добавить новую точку.

```
poly_rect.add(Point(50,75));
```

Прямоугольник не может состоять из пяти точек.





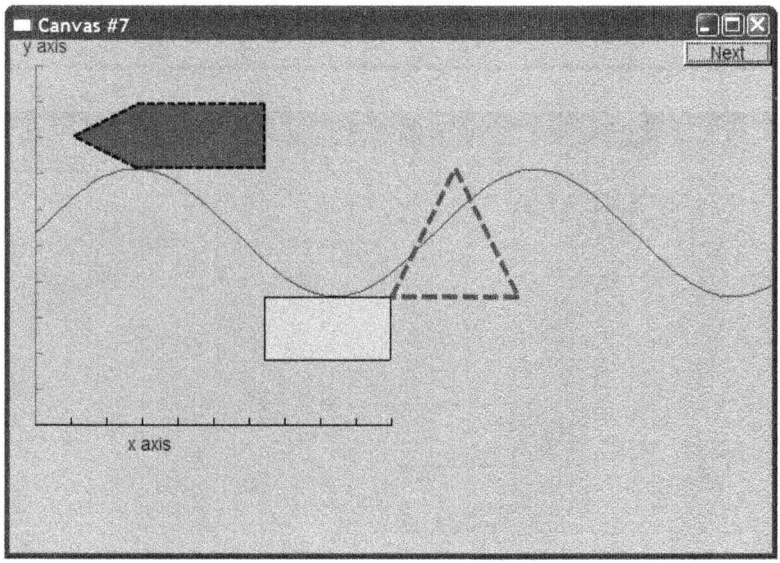
Важно понимать, что объект класса **Rectangle** должен не только выглядеть, как прямоугольник на экране, но и обладать его фундаментальными свойствами, известными нам из геометрии. Мы пишем код, зависящий от того обстоятельства, что объект класса **Rectangle** действительно является прямоугольником.

12.7.7. Заполнение

До сих пор наши фигуры были нарисованы схематично, с помощью одних внешних границ. Но их можно заполнить цветом.

```
r.set_fill_color(Color::yellow); // Цвет внутри прямоугольника
poly.set_style(Line_style(Line_style::dash,4));
poly_rect.set_style(Line_style(Line_style::dash,2));
poly_rect.set_fill_color(Color::green);
win.set_label("Canvas #7");
win.wait_for_button();
```

Мы также решили, что прежний стиль линии в нашем треугольнике (**poly**) нам не нравится, и изменили его на жирный пунктир (в четыре раза толще обычного пунктира). Аналогично мы изменили стиль объекта **poly_rect** (теперь он не выглядит так же как прямоугольник).



Если внимательно присмотреться к объекту **poly_rect**, то можно увидеть, что он рисуется поверх заполнения.

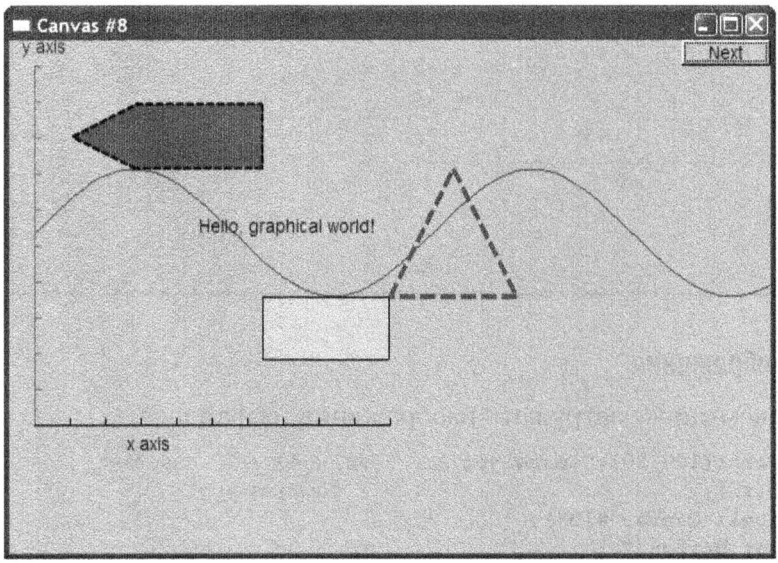
Заполнить цветом можно любую замкнутую фигуру (раздел 13.9). Просто заполнение прямоугольников выполняется особенно просто (и быстро).

12.7.8. Текст

В заключение укажем, что ни одна система вывода графических изображений не может считаться полной, если она не способна некоторым простым образом выводить текст — вырисовывание каждого символа с помощью набора линий в расчет не принимается. Мы

приписываем окну метку, оси также могут иметь метки, но, помимо этого, мы можем вывести текст в любое место окна, используя объект класса **Text**.

```
Text t{Point{150,150}, "Hello, graphical world!"};
win.attach(t);
win.set_label("Canvas #8");
win.wait_for_button();
```

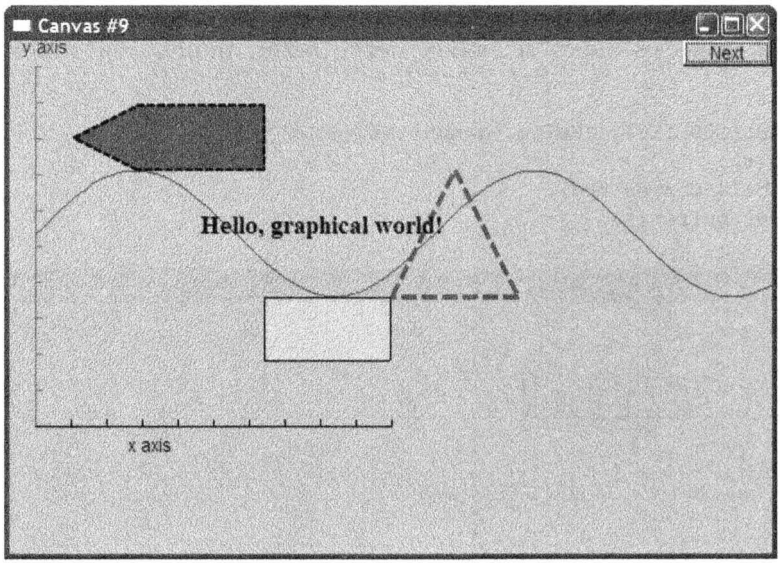


Из графических примитивов, показанных в этом окне, можно создать сколь угодно сложные и утонченные фигуры. Пока мы просто отметим особенность кода в этой главе: в нем нет циклов, условных конструкций, а все данные в нем жестко "прошиты". Выходная информация скомпонована из примитивов простейшим образом. Как только мы начнем составлять из этих примитивов сложные фигуры с помощью данных и алгоритмов, все станет намного интереснее.

Мы видели, как можно управлять цветом текста: метка оси **Axis** (см. раздел 12.7.3) представляет собой просто объект класса **Text**. Кроме того, мы можем выбирать шрифт и размер символов.

```
t.set_font(Font::times_bold);
t.set_font_size(20);
win.set_label("Canvas #9");
win.wait_for_button();
```

Здесь мы увеличили буквы в строке **"Hello, graphical world!"** до 20 пунктов и выбрали полужирный шрифт Times.

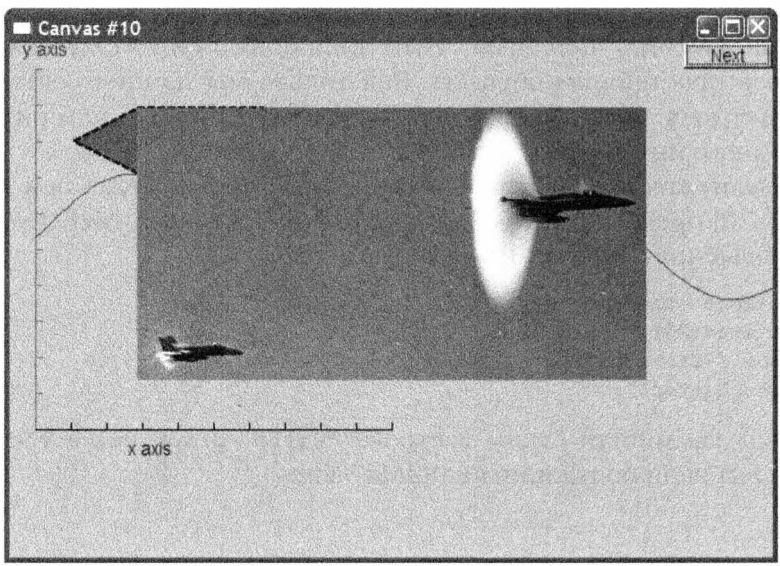


12.7.9. Изображения

Мы можем также загружать изображения из файлов.

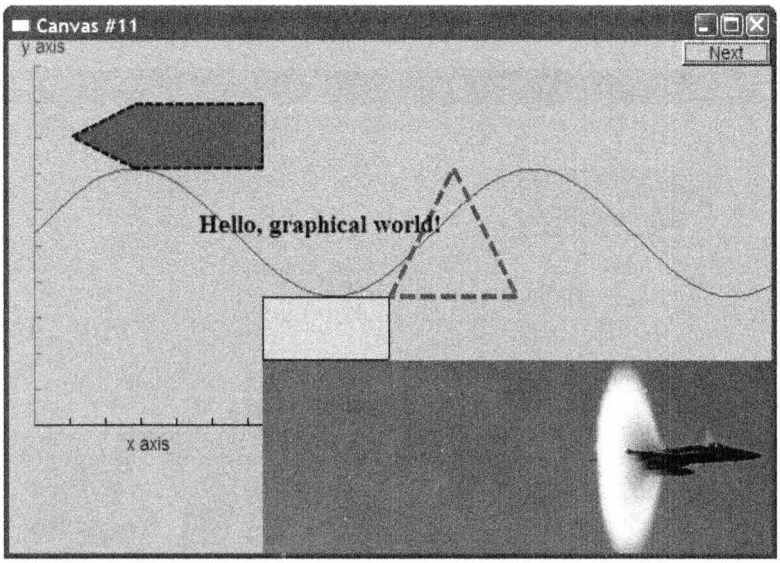
```
Image ii{Point{100,50},"image.jpg"}; // Файл 400×212 пикселей
win.attach(ii);                      // в формате jpg
win.set_label("Canvas #10");
win.wait_for_button();
```

Файл `image.jpg` — это фотография двух самолетов, преодолевающих звуковой барьер.



Эта фотография относительно велика и размещается поверх нашего текста и фигур. Окно требуется немного открыть, для чего мы немного сдвинем фотографию.

```
ii.move(100,200);
win.set_label("Canvas #11");
win.wait_for_button();
```



Обратите внимание на то, что части фотографии, не попавшие в окно, не представлены на экране, поскольку то, что выходит за его пределы, обрезается.

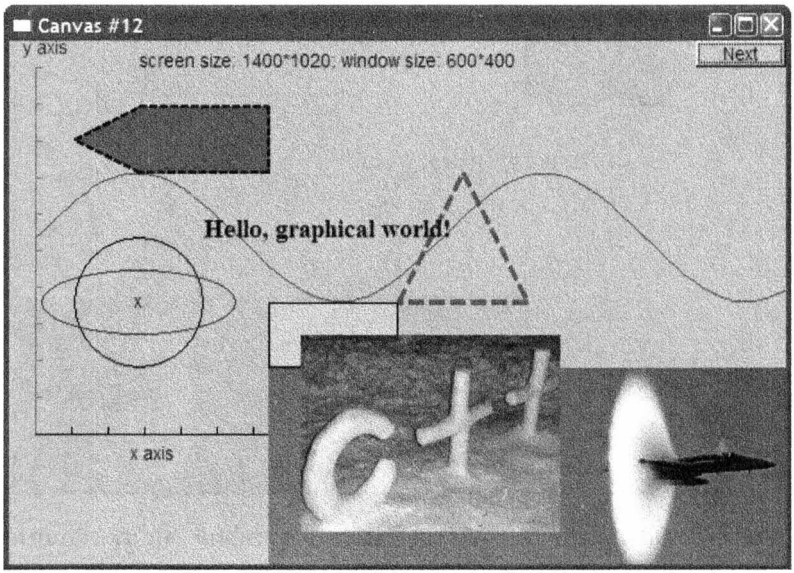
12.7.10. И многое другое

Приведем без объяснений еще один фрагмент кода

```
Circle  c{Point{100,200},50};
Ellipse e{Point{100,200},75,25};
e.set_color(Color::dark_red);
Mark m{Point{100,200},'x'};

ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Text sizes{Point{100,20},oss.str()};

Image cal{Point{225,225},"snow_cpp.gif"}; // 320*240 пикселей, формат gif
cal.set_mask(Point{40,40},200,150);       // Вывод центральной части рисунка

win.attach(c);
```

```
win.attach(m);
win.attach(e);

win.attach(sizes);
win.attach(cal);
win.set_label("Canvas #12");
win.wait_for_button();
```

Можете ли вы догадаться, что делает этот фрагмент? Достаточно ли это очевидно?



Между кодом и тем, что появляется на экране, существует прямая связь. Даже если вам пока непонятно, как этот код приводит к таким результатам, не беспокойтесь: вскоре все станет ясно. Обратите внимание на то, что для форматирования текстовых объектов, содержащих информацию о размерах, мы использовали поток **ostringstream** (см. раздел 11.4).

12.8. Запуск программы

Мы показали, как можно создать окно и нарисовать в нем разные фигуры. В следующих главах мы покажем, как определены класс **Shape** и его подклассы, а также как их использовать.

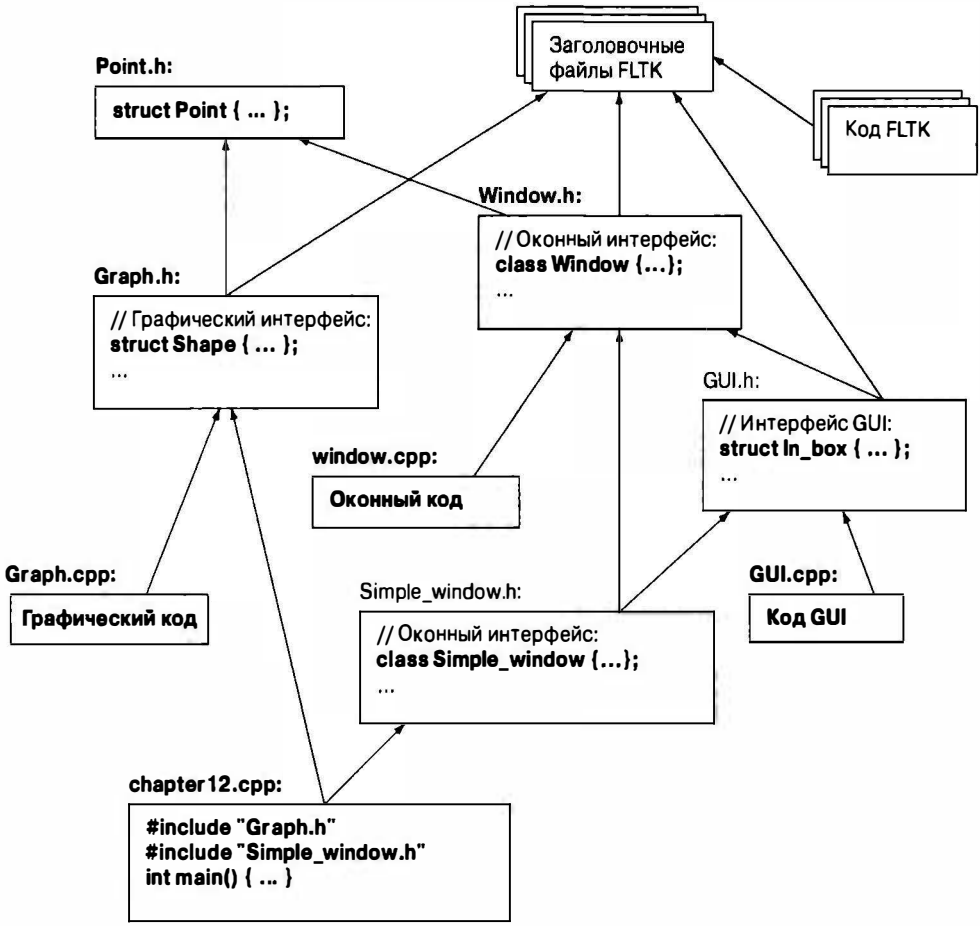
Для того чтобы выполнить эту программу, требуется больше действий, чем для других программ, описанных ранее. Помимо кода в функции **main()**, нам необходимо скомпилировать код библиотеки интерфейса и скомпоновать его с нашей программой, и наконец, чтобы программа

заработала, на компьютере должна быть установлена библиотека FLTK (или другая система графического пользовательского интерфейса).

Итак, можно сказать, что наша программа состоит из четырех частей.

- Код нашей программы (`main()` и т.д.).
- Наша библиотека интерфейса (`Window`, `Shape`, `Polygon` и т.д.).
- Библиотека FLTK.
- Стандартная библиотека языка C++.

Кроме того, мы неявно используем операционную систему. Оставляя в стороне операционную систему и стандартную библиотеку, мы можем проиллюстрировать организацию графической программы следующим образом.



Как собрать все это в работоспособное единое целое, объясняется в приложении Г.

12.8.1. Исходные файлы

Наша библиотека графики и графического пользовательского интерфейса состоит лишь из пяти заголовочных и трех исходных файлов.

- Заголовочные файлы
  - `Point.h`
  - `Window.h`
  - `Simple_window.h`
  - `Graph.h`
  - `GUI.h`
- Исходные файлы
  - `Window.cpp`
  - `Graph.cpp`
  - `GUI.cpp`

До главы 16 мы можем игнорировать файлы графического пользовательского интерфейса.

Задание

Это задание — графический эквивалент простейшей программы "`Hello, World!`" Его цель — ознакомить вас с простейшими графическими средствами.

1. Напишите программу, создающую пустое окно `Simple_window` размером 600×400 пикселей с меткой `My window`, скомпилируйте ее, скомпонуйте и выполните. Помните о том, что вы должны подключить библиотеку FLTK, описанную в приложении Г, включить в свой код заголовочные файлы `Graph.h` и `Simple_Window.h`, а также включить в проект файлы `Graph.cpp` и `Window.cpp`.
2. Добавьте по одному примеры из раздела 12.7, выполняя тестирование после каждого добавленного подраздела.
3. Выполните программу, внеся небольшие изменения (например, измените цвет, местоположение фигур или количество точек) в каждом из примеров.

Контрольные вопросы

1. Зачем нужна графика?
2. Когда можно попытаться обойтись без графики?
3. Почему графика интересна программисту?
4. Что такое окно?
5. В каком пространстве имен находятся наши классы графического интерфейса (наша графическая библиотека)?
6. Какие заголовочные файлы необходимы для использования графических средств нашей библиотеки?

7. Что представляет собой простейшее окно?
8. Что представляет собой минимальное окно?
9. Что такое метка окна?
10. Как задать метку окна?
11. Что собой представляют экранные, оконные и математические координаты?
12. Приведите примеры простых фигур, которые можно отобразить на экране.
13. Какие команды связывают фигуру с окном?
14. Какие базовые фигуры можно использовать для того, чтобы нарисовать шестиугольник?
15. Как вывести текст в окне?
16. Как поместить в окне фотографию вашего лучшего друга или подруги с помощью написанной вами программы.
17. Представьте, что вы создали объект класса **`Window`**, но на экране ничего не появилось. Каковы возможные причины этого?
18. Представьте, что вы создали объект класса **`Shape`**, но на экране ничего не появилось. Каковы возможные причины этого?

Термины

GUI	библиотека GUI	координаты
HTML	графика	окно
JPEG	дисплей	слой программного обеспечения
XML	заполнение цветом	стиль линии
библиотека FLTK	изображение	цвет

Упражнения

Для выполнения приведенных ниже упражнений рекомендуем использовать класс **`Simple_window`**.

1. Нарисуйте прямоугольник как объект класса **`Rectangle`** и как объект класса **`Polygon`**. Сделайте линии объекта класса **`Polygon`** красными, а линии объекта класса **`Rectangle`** — синими.
2. Нарисуйте объект класса **`Rectangle`** с размерами 100×30 и поместите в него текст "Howdy!"
3. Нарисуйте свои инициалы высотой 150 пикселей. Используйте толстую линию. Нарисуйте каждый инициал другим цветом.

4. Нарисуйте доску для игры в крестики-нолики размером 3×3, чередуя белые и красные квадраты.
5. Нарисуйте красную рамку шириной полсантиметра вокруг прямоугольника, высота которого составляет три четверти высоты вашего экрана, а ширина — две трети ширины экрана.
6. Что произойдет, если вы нарисуете фигуру, которая не помещается в окне? Что произойдет, если вы нарисуете окно, которое не помещается на экране? Напишите две программы, иллюстрирующие эти эффекты.
7. Нарисуйте двумерный дом анфас, как это делают дети: дверь, два окна и крыша с дымовой трубой. Детали можете выбрать сами, можете даже нарисовать дымок из трубы.
8. Нарисуйте пять олимпийских колец. Если вы не помните их цвета, найдите их в Интернете.
9. Выведите на экран фотографию вашего друга. Напишите его имя в заголовке окна и в заголовке внутри окна.
10. Нарисуйте диаграмму файлов из раздела 12.8.
11. Нарисуйте ряд правильных многоугольников, вложенных друг в друга. Наиболее глубоко вложенный многоугольник должен быть равносторонним треугольником, вложенным в квадрат, вложенный в пятиугольник, и т.д. Для любителей математики: пусть все точки каждого `N`-угольника касаются сторон `(N+1)`-угольника. *Указание*: тригонометрические функции объявлены в заголовочном файле `<cmath>` (разделы 24.8, Б.9.2)
12. Суперэллипс — это двумерная фигура, определенная уравнением

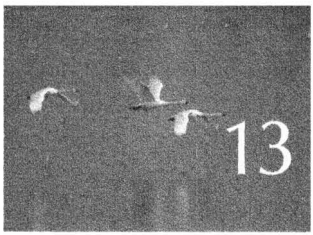
$$\left|\frac{x}{a}\right|^m + \left|\frac{y}{b}\right|^n = 1; \quad m,n > 0.$$

Поищите в вебе информацию о *суперэллипсе*, чтобы лучше представить, как он выглядит. Напишите программу, которая рисует звездообразные шаблоны, соединяя точки, лежащие на суперэллипсе. Пусть параметры `a`, `b`, `m`, `n` и `N` вводятся как аргументы. Выберите `N` точек на суперэллипсе, определенном параметрами `a`, `b`, `m` и `n`. Пусть эти точки лежат на равном расстоянии друг от друга (для некоторого определения "равного расстояния"). Соедините каждую из этих `N` точек с одной или несколькими другими точками (если хотите, можете задать количество таких точек с помощью дополнительного аргумента или использовать число `N-1`, т.е. все другие точки).

13. Придумайте способ раскрасить линии из предыдущего упражнения. Нарисуйте разные линии разным цветом.

Послесловие

В идеальном проекте каждая концепция непосредственно представлена в виде некоторой сущности в программе. Часто мы выражаем свои идеи в виде классов, реальные вещи — в виде объектов классов, а действия и вычисления — в виде функций. Графика — это область, в который эта мысль нашла очевидное воплощение. У нас есть концепции, такие как окружности и многоугольники, и мы выражаем их в программе в виде классов `Circle` и `Polygon`. Графика отличается от других приложений тем, что, создавая графические программы, программист может сразу видеть объекты классов на экране. Иначе говоря, состояние такой программы непосредственно доступно для наблюдения (в большинстве приложений мы лишены этой возможности). Это непосредственное соответствие между идеями, кодом и выводом делает программирование графики очень привлекательным. Однако помните, что графика — это всего лишь иллюстрация общей идеи использования классов для выражения основных понятий в виде кода. Эта идея носит намного более общий характер: все наши идеи могут быть выражены в коде либо в виде класса, либо в виде объекта класса, либо в виде совокупности классов.



Графические классы

Язык, не изменяющий образ мышления, не достоин изучения.
Общепринятое мнение

В главе 12 описано, что можно сделать с помощью графики и набора простых интерфейсных классов и как это можно сделать. В этой главе рассматриваются многие из предложенных классов. Она посвящена проектированию, использованию и реализации индивидуальных интерфейсных классов, таких как **`Point`**, **`Color`**, **`Polygon`** и **`Open_polyline`**, а также методам их использования. В следующей главе будут изложены идеи, касающиеся проектирования связанных классов, а также описаны методы их реализации.

13.1. Обзор графических классов

Библиотеки графики и графического пользовательского интерфейса предоставляют множество возможностей. Слово "множество" означает сотни классов, часто содержащих десятки функций. Их описания, справочные руководства и документация напоминают учебники по ботанике, в которых перечислены тысячи растений, упорядоченных в соответствии с устаревшей классификацией. Это просто пугает! Обзор возможностей современных библиотек графики и графического пользовательского интерфейса может быть увлекательным занятием. Он может вызвать у читателя ощущения ребенка, попавшего в кондитерскую лавку и не понимающего, с чего начать и понравится ли ему то, что он выберет.

Одна из целей нашей библиотеки интерфейса — компенсировать шок, вызванный сложностью библиотек графики и графического пользовательского интерфейса. Мы опишем только десятка два классов с немногими операциями, которые позволят вам создавать полезные графические приложения. Кроме того, эти классы позволяют ввести ключевые концепции графики и графического пользовательского интерфейса. Вы уже можете представлять результаты своей работы в виде простых графиков. После чтения этой главы диапазон ваших приложений будет превосходить начальные требования большинства программистов. После главы 14 вы будете понимать большинство идей и методов проектирования, что позволит вам еще глубже разобраться в графических библиотеках и создавать еще более сложные приложения. Этого можно достичь, либо включая в свои программы описанные здесь возможности, либо адаптируя другие библиотеки графики и графического пользовательского интерфейса для C++.

Основные классы графического интерфейса перечислены в следующей таблице.

Классы графического интерфейса	
`Color`	Используется для создания линий, текста и заполнения фигур
`Line_style`	Используется для рисования линий
`Point`	Используется для задания местоположения на экране и внутри окна `Window`
`Line`	Отрезок линии, видимый на экране, определенный двумя конечными точками `Point`
`Open_polyline`	Последовательность соединенных друг с другом отрезков линий, определенная последовательностью точек `Point`
`Closed_polyline`	Похож на класс `Open_polyline`, за исключением того, что отрезок линии соединяет последнюю точку с первой
`Polygon`	Класс `Closed_polyline`, в котором никакие два отрезка не пересекаются
`Text`	Строка символов
`Lines`	Набор отрезков, определенных парами точек `Point`
`Rectangle`	Прямоугольник — распространенная фигура, оптимизированная для быстрого и удобного отображения
`Circle`	Окружность, определенная центром и радиусом
`Ellipse`	Эллипс, определенный центром и двумя осями
`Function`	Функция одной переменной, заданная в определенном отрезке
`Axis`	Помеченная ось координат
`Mark`	Точка, помеченная символом (таким, как `x` или `o`)
`Marks`	Последовательность точек, помеченных символами (такими, как `x` или `o`)
`Marked_polyline`	Класс `Open_polyline` с точками, помеченными символами
`Image`	Содержимое файла с изображением

Классы `Function` и `Axis` описываются в главе 15. В главе 16 рассматриваются основные классы пользовательского интерфейса.

Классы графического пользовательского интерфейса	
`Window`	Область экрана, в которой отображаются графические объекты
`Simple_window`	Окно с кнопкой `Next`
`Button`	Прямоугольник в окне, обычно помеченный, щелчок на котором приводит к вызову одной из наших функций
`In_box`	Область в окне, обычно помеченная, в которой пользователь может ввести строку
`Out_box`	Область в окне, обычно помеченная, в которой приложение может вывести строку
`Menu`	Вектор объектов класса `Button`

Исходный код организован в виде следующих файлов.

Исходные интерфейсные графические файлы	
`Point.h`	Класс `Point`
`Graph.h`	Все прочие классы графического интерфейса
`Window.h`	Класс `Window`
`Simple_window.h`	Класс `Simple_window`
`GUI.h`	`Button` и прочие классы GUI
`Graph.cpp`	Определения функций из файла `Graph.h`
`Window.ccp`	Определения функций из файла `Window.h`
`GUI.cpp`	Определения функций из файла `GUI.h`

В дополнение к графическим классам мы представим класс, который может оказаться полезным для хранения коллекций объектов классов, таких как `Shape` или `Widget`.

Контейнер объектов классов `Shape` или `Widget`	
`Vector_ref`	Вектор с интерфейсом, обеспечивающим удобное хранение неименованных элементов

Читая следующие разделы, пожалуйста, не торопитесь. Они не содержат ничего такого, что не было бы совершенно очевидным, но цель этой главы — не просто продемонстрировать несколько красивых рисунков (на экране компьютера или телевизора вы каждый день можете увидеть более красивые изображения). Основные цели этой главы перечислены ниже.

- Продемонстрировать связь между кодом и создаваемыми рисунками.
- Научить вас читать исходные тексты и размышлять над тем, как они работают.
- Научить вас размышлять о проектировании программ, в частности о том, как выразить понятия в виде классов. Почему эти классы устроены так, а не иначе? Как их можно было бы написать по-другому? Вы можете принять много-много проектных решений, и в большинстве своем они будут отличаться от наших незначительно (хотя в некоторых случаях — кардинально).

Итак, пожалуйста, не торопитесь, иначе можете пропустить что-то важное и не сумеете выполнить упражнения.

13.2. Классы `Point` и `Line`

Наиболее фундаментальной частью любой графической системы является *точка*. Определив это понятие, вы определите все ваше геометрическое пространство. В данной книге мы используем обычное, компьютерно-ориентированное двумерное представление точек в виде пары

целочисленных координат (x, y). Как указано в разделе 12.5, координаты x изменяются от нуля (левого края экрана) до **x_max()** (правого края экрана); координаты y изменяются от нуля (верхнего края экрана) до **y_max()** (нижнего края экрана).

Как определено в файле **Point.h**, класс **Point** — это просто пара чисел типа **int** (координаты точки):

```
struct Point {
     int x, y;
};

bool operator==(Point a, Point b){ return a.x==b.x && a.y==b.y; }
bool operator!=(Point a, Point b){ return !(a==b); }
```

В файле **Graph.h** определены также класс **Shape**, подробно описанный в главе 14, и класс **Line**.

```
struct Line : Shape {          // Класс Line - это класс Shape,
                               // определенный двумя точками
    Line(Point p1, Point p2); // Создание объекта Line
                               // из двух точек Point
};
```

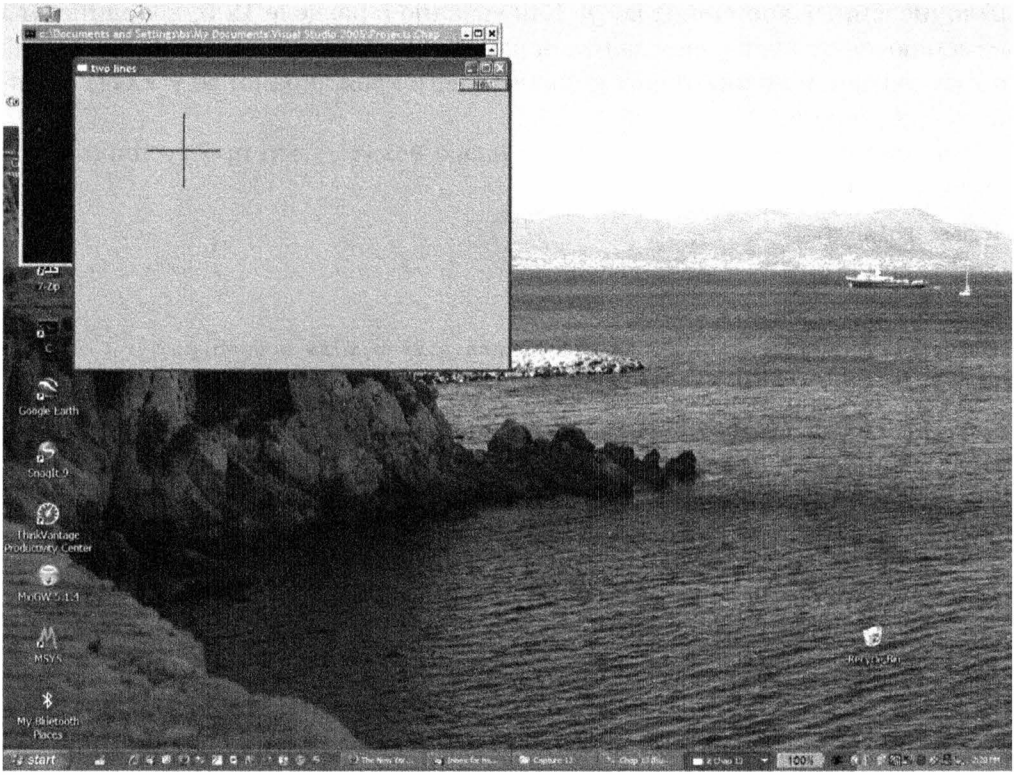
Класс **Line** — это разновидность класса **Shape**. Именно это означает строка "**: Shape**". Класс **Shape** называют *базовым классом* (base class) класса **Line**. В основном класс **Shape** предоставляет возможности, упрощающие определение класса **Line**. Как только мы столкнемся с конкретными фигурами, такими как **Line** или **Open_polyline**, то увидим, что это значит (см. главу 14).

Класс **Line** определяется двумя объектами класса **Point**. Оставляя в стороне "леса" (директивы **#include** и прочие детали, описанные в разделе 12.3), мы можем создать линию и нарисовать ее на экране с помощью следующего исходного текста.

```
// Рисуем две линии
constexpr Point x {100,100}

Simple_window win1{x,600,400,"Two lines"};
Line horizontal{x,Point{200,100}};    // Горизонтальная линия
Line vertical{Point{150,50},          // Вертикальная линия
              Point{150,150}};
win1.attach(horizontal);              // Связываем их с окном
win1.attach(vertical);
win1.wait_for_button();               // Вывести
```

Выполнив этот фрагмент кода, получим на экране такое изображение. В качестве пользовательского интерфейса, предназначенного для того, чтобы упростить работу, класс **Line** работает неплохо. Не нужно быть Эйнштейном, чтобы понять, что инструкция

```
Line vertical{Point{150,50},Point{150,150}};
```



создает (вертикальную) линию, соединяющую точки (150,50) и (150,150). Разумеется, существуют детали реализации, но вам необязательно знать их, чтобы создавать линии. Реализация конструктора класса **Line** довольно проста.

```
Line::Line(Point p1, Point p2) // Создаем линию по двум точкам
{
    add(p1);                   // Добавляем точку p1
    add(p2);                   // Добавляем точку p2
}
```

Иначе говоря, конструктор просто добавляет две точки. Добавляет куда? И как объект класса **Line** рисуется в окне? Ответ кроется в классе **Shape**. Как будет описано в главе 14, класс **Shape** может хранить точки, определяющие линии, знает, как рисовать линии, определенные парами точек, и имеет функцию **add()**, позволяющую добавлять в объект **Shape** объекты **Point**. Ключевой момент заключается в том, что определение класса **Line** тривиально. Большая часть работы по реализации выполняется "системой", поэтому программист может сосредоточиться на создании простых классов, которые легко использовать.

С этого момента оставим в стороне определение класса **Simple_window** и вызовы функции **attach()**. Они не более чем "леса", необходимые для завершения программы, но ничего не добавляющие к специфике объектов класса **Shape**.

13.3. Класс Lines

Как оказывается, мы редко рисуем отдельную линию. Как правило, мы работаем с объектами, состоящими из многих линий, например с треугольниками, многоугольниками, графами, лабиринтами, сетками, диаграммами, графиками математических функций и т.д. Одним из простейших "классов составных графических объектов" является класс **Lines**.

```
struct Lines : Shape {                  // Связанные друг с другом линии
    Lines() {}                          // Пустой набор линий
                                        // Инициализация списком точек:
    Lines(initializer_list<Point> lst);
    void draw_lines() const;
    void add(Point p1, Point p2); // Добавление линии,
};                                      // заданной двумя точками
```

Объект класса **Lines** представляет собой коллекцию линий, каждая из которых определена парой точек **Point**. Например, если бы мы рассматривали две линии из примера в разделе 13.2 как часть одного графического объекта, то могли бы определить его следующим образом:

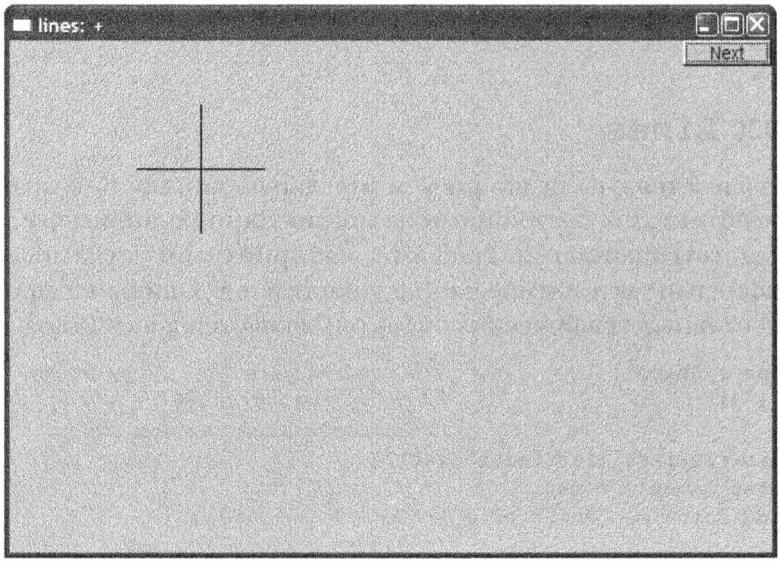
```
Lines x;
x.add(Point{100,100},Point{200,100}); // Первая линия:
                                      // горизонтальная
x.add(Point{150,50},Point{150,150});  // Вторая линия:
                                      // вертикальная
```

В этом случае мы получили бы совершенно такой же результат (вплоть до последнего пикселя), как и в варианте с классом **Line**.

Единственный способ, который позволяет различить эти варианты, — создать отдельное окно и приписать ему другую метку.

Разница между совокупностью объектов класса **Line** и совокупностью линий в объекте класса **Lines** заключается лишь в нашей точке зрения на то, что должно произойти. Используя класс **Lines**, мы выражаем наше мнение, что две линии образуют одно целое и должны обрабатываться одновременно. Например, мы можем изменить цвет всех линий, являющихся частью объекта **Lines**, с помощью одной команды. С другой стороны, мы можем присвоить каждой линии, являющейся отдельным объектом класса **Line**, разные цвета. В качестве более реалистичного примера рассмотрим определение сетки. Сетка состоит из большого количества горизонтальных и вертикальных линий, проведенных на одинаковых

расстояниях друг от друга. Однако мы считаем сетку одним целым, поэтому определяем ее линии как части объекта класса **Lines** с именем **grid**.



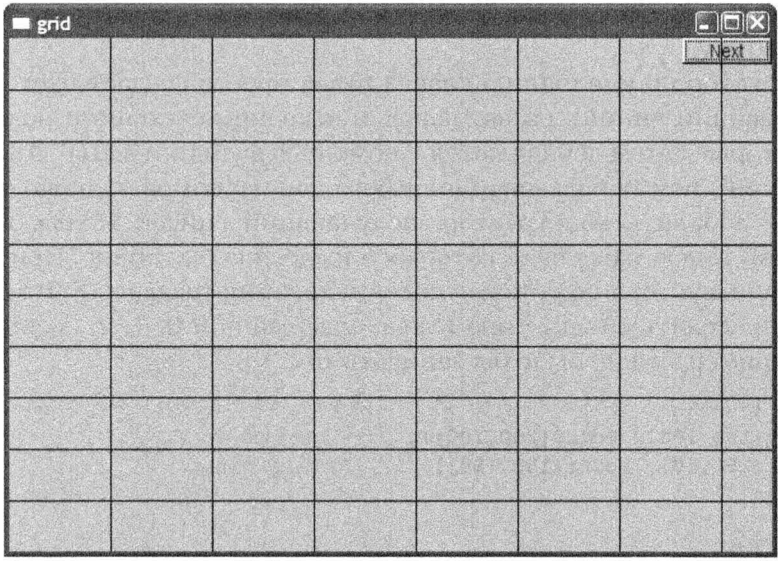

```
int x_size = win3.x_max();                    // Размер окна
int y_size = win3.y_max();
int x_grid = 80;
int y_grid = 40;

Lines grid;
for (int x=x_grid; x<x_size; x+=x_grid)
    grid.add(Point{x,0},Point{x,y_size}); // Вертикальная линия
for (int y = y_grid; y<y_size; y+=y_grid)
    grid.add(Point{0,y},Point{x_size,y}); // Горизонтальная линия
```

Обратите внимание на то, как мы определили размеры нашего окна с помощью функций **x_max()** и **y_max()**. Это первый пример, в котором мы написали код, вычисляющий объект, подлежащий выводу на экран. Было бы невыносимо скучно определять сетку, вводя именованные переменные для каждой линии, из которых она состоит. Данный фрагмент кода создает следующее окно.

Вернемся к классу **Lines**. Как реализованы функции-члены класса **Lines**? Класс **Lines** содержит два конструктора и две операции.

Функция **add()** просто добавляет линию, определенную парой точек, к набору линий, которые будут выведены на экран.

```
void Lines::add(Point p1, Point p2)
{
    Shape::add(p1);
    Shape::add(p2);
}
```



Квалификатор `Shape::` здесь совершенно необходим, поскольку в противном случае компилятор рассматривал бы выражение `add(p1)` как (некорректную) попытку вызвать функцию `add()` из класса `Lines`, а не из класса `Shape`.

Функция `draw_lines()` рисует линии, определенные с помощью функции `add()`.

```
void Lines::draw_lines() const
{
    if (color().visibility())
        for (int i=1; i<number_of_points(); i+=2)
            fl_line(point(i-1).x,point(i-1).y,
                    point(i).x,point(i).y);
}
```

Иначе говоря, функция `Lines::draw_lines()` на каждом шаге цикла получает две точки (начиная с точек с индексами 0 и 1) и рисует линию, соединяющую эти точки с помощью библиотечной функции `fl_line()`. *Видимость* (visibility) — это свойство объекта класса `Color` (раздел 13.4), поэтому, прежде чем рисовать эти линии, мы должны убедиться, что они являются видимыми.

Как будет показано в главе 14, функция `draw_lines()` вызывается "системой". Мы не обязаны проверять, является ли количество точек четным, так как функция `add()` класса `Lines` может добавлять только пары точек. В классе `Shape` определены функции `number_of_points()` и `point()` (см. раздел 14.2), и их смысл очевиден. Эти две функции обеспечивают доступ только для чтения к точкам объекта класса `Shape`. Функция-член

`draw_lines()` определена как `const` (см. раздел 9.7.4), поскольку она не изменяет фигуру.

Конструктор по умолчанию класса `Lines` просто создает пустой объект (не содержащий линий): наша модель в исходном состоянии не содержит точек, которые затем добавляются с помощью функции `add()`. Этот подход более гибкий, чем использование любого конструктора. Однако мы могли бы также добавить конструктор, получающий список `initializer_list` пар точек, каждая из которых определяла бы линию. При наличии конструктора со списком инициализации (раздел 18.2) мы можем определить объект типа `Lines`, содержащий 0, 1, 2, ... линий. Так, первый пример можно было бы записать так:

```
Lines x = {
    {Point{100,100}, Point{200,100}}, // Первая линия
    {Point{150,50}, Point{150,150}}   // Вторая линия
};
```

или даже так:

```
Lines x = {
    {{100,100},{200,100}}, // Первая линия
    {{150,50}, {150,150}}  // Вторая линия
};
```

Определить такой конструктор со списком инициализации очень просто:

```
void Lines::Lines(initializer_list<pair<Point,Point>> lst)
{
    for (auto p : lst) add(p.first,p.second);
}
```

Ключевое слово `auto` представляет собой заменитель для типа `pair <Point,Point>`, а `first` и `second` является именами первого и второго членов пары. Типы `initializer_list` и `pair` определены в стандартной библиотеке (см. разделы Б.6.4 и Б.6.3 соответственно).

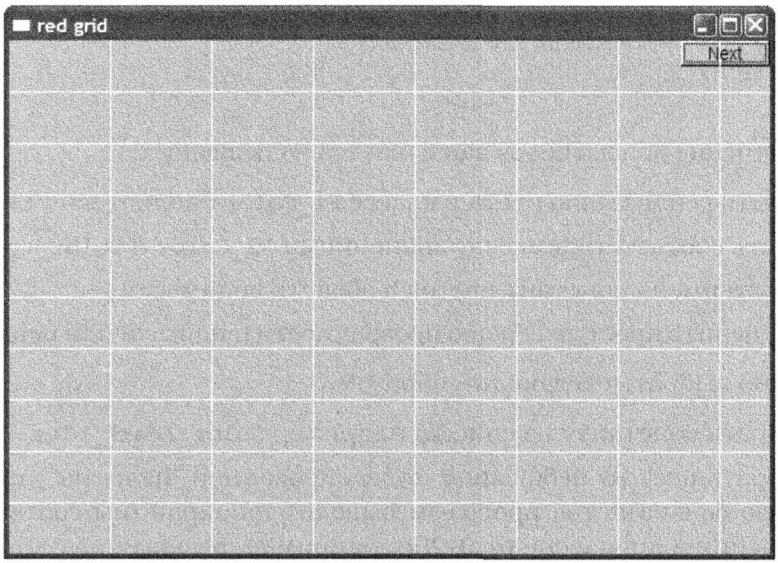
13.4. Класс `Color`

`Color` — это тип, представляющий цвет. Его можно использовать примерно так:

```
grid.set_color(Color::red);
```

Эта инструкция окрашивает линии, определенные в объекте `grid`, в красный цвет. В итоге получается приведенное ниже изображение, на котором сетка изображена красным цветом:

Класс `Color` определяет понятие цвета и приписывает символические имена некоторым наиболее распространенным цветам.




```
struct Color {
    enum Color_type {
        red          =FL_RED,
        blue         =FL_BLUE,
        green        =FL_GREEN,
        yellow       =FL_YELLOW,
        white        =FL_WHITE,
        black        =FL_BLACK,
        magenta      =FL_MAGENTA,
        cyan         =FL_CYAN,
        dark_red     =FL_DARK_RED,
        dark_green   =FL_DARK_GREEN,
        dark_yellow =FL_DARK_YELLOW,
        dark_blue    =FL_DARK_BLUE,
        dark_magenta=FL_DARK_MAGENTA,
        dark_cyan    =FL_DARK_CYAN
    };

    enum Transparency { invisible = 0, visible=255 };

    Color(Color_type cc) :c{Fl_Color(cc)}, v{visible} {}
    Color(Color_type cc, Transparency vv)
            :c{Fl_Color(cc)}, v{vv} { }
    Color(int cc) :c{Fl_Color(cc)}, v{visible} {}
    Color(Transparency vv):c{Fl_Color()}, v{vv} {} // Цвет по
                                                   // умолчанию

    int as_int() const { return c; }

    char visibility() const { return v; }
    void set_visibility(Transparency vv) { v=vv; }
```

```
private:
    char v;       // Видимый или невидимый
    Fl_Color c;
};
```

Предназначение класса `Color` заключается в следующем.

- Скрыть реализацию цвета в классе `Fl_Color` библиотеки FLTK.
- Задать соответствие между значениями `Fl_Color` и `Color_type`.
- Внести имена констант цветов в область видимости.
- Обеспечить простую версию прозрачности (видимый или невидимый).

Цвет можно выбрать следующим образом.

- Выбрать константу из списка, например `Color::dark_blue`.
- Выбрать цвет из небольшой индексированной "палитры" цветов, которую большинство программ выводит на экран (им соответствуют значения в диапазоне от 0–255; например, выражение `Color(99)` означает темно-зеленый цвет). Пример такой программы приведен в разделе 13.9.
- Выбрать значение в системе RGB (Red, Green, Blue — красный, зеленый, синий), которую мы здесь обсуждать не будем. При необходимости в ней нетрудно разобраться самостоятельно. В частности, поиск "RGB color" при поиске в Интернете даст множество ссылок, в том числе такие, как `http://en.wikipedia.org/wiki/RGB_color_model` и `www.rapidtables.com/web/color/RGB_Color.htm`. (См. также упр. 13 и 14.)

Обратите внимание на конструкторы класса `Color`, позволяющие создавать объекты как из объектов типа `Color_type`, так и из обычных чисел типа `int`. Каждый конструктор инициализирует член `c`. Вы можете возразить, что переменная `c` названа слишком коротко и непонятно, но, поскольку она используется в очень небольшой части класса `Color` и не предназначена для широкого использования, это не является недостатком. Мы поместили член `c` в закрытый раздел, чтобы защитить его от непосредственного обращения пользователей. Для представления члена `c` мы используем тип `Fl_Color`, определенный в библиотеке FLTK, который хотели бы скрыть от пользователей. Однако очень часто этот тип интерпретируется как целочисленное представление значения RGB (или другого значения), поэтому на этот случай мы предусмотрели функцию `as_int()`. Обратите внимание на то, что функция `as_int()` является константной функцией-членом, поскольку она не изменяет объект класса `Color`, который ее использует.

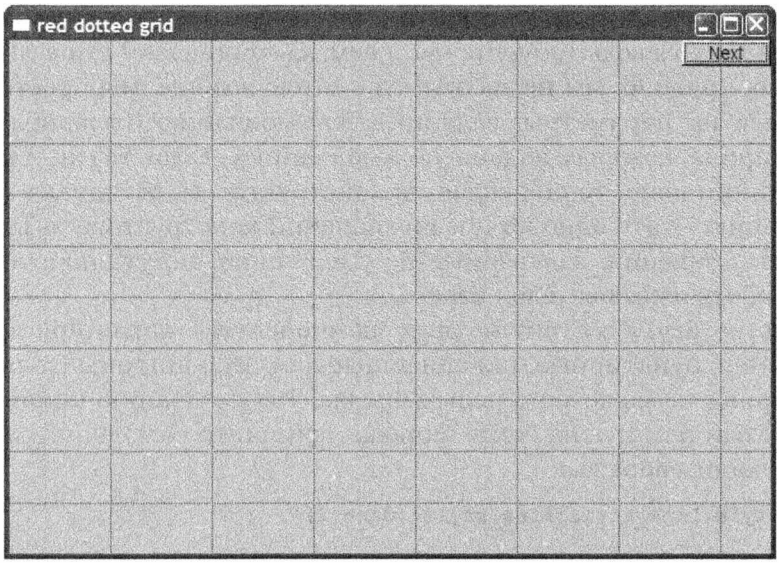
Прозрачность задается членом `v`, который может принимать значения `Color::visible` и `Color::invisible`, имеющие очевидный смысл. Вы можете удивиться: зачем нужен "невидимый цвет". Оказывается, он может быть очень полезен для того, чтобы скрыть часть составной фигуры на экране.

13.5. Класс `Line_style`

Нарисовав на экране несколько линий, мы можем различать их по цвету, стилю или по обоим этим признакам. Стиль линии — это шаблон, задающий ее внешний вид. Класс `Line_style` используется приблизительно так:

```
grid.set_style(Line_style::dot);
```

Эта инструкция выводит на экран линии, заданные в объекте `grid`, как последовательность точек, а не как сплошную линию.



В результате сетка станет немного тоньше, зато менее заметной. Настраивая ширину (толщину) линий, можем придать сетке требуемый вид.

Класс `Line_style` выглядит так:

```
struct Line_style {
    enum Line_style_type {
        solid      =FL_SOLID,       // -------
        dash       =FL_DASH,        // - - - -
        dot        =FL_DOT,         // .......
        dashdot    =FL_DASHDOT,     // - . - .
        dashdotdot=FL_DASHDOTDOT, // -..-..
    };

    Line_style(Line_style_type ss):s{ss}, w{0} {}
    Line_style(Line_style_type lst, int ww):s{lst}, w{ww} {}
    Line_style(int ss):s{ss}, w{0} {}
    int width() const { return w; }
    int style() const { return s; }
private:
    int s;
```

```
    int w;
};
```

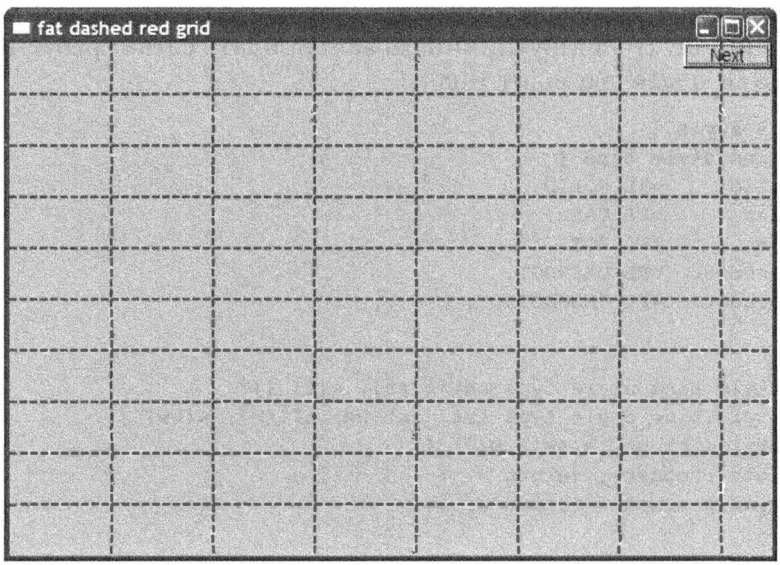
Методы программирования, использованные для определения класса `Line_style`, ничем не отличаются от методов, использованных для класса `Color`. Здесь мы снова скрываем тот факт, что для представления стилей линии библиотека FKTK использует тип `int`. Почему стоит скрывать эту информацию? Потому что эти способы представления при модификации библиотеки могут измениться. В следующей версии библиотеки FLTK может появиться тип `Fl_linestyle`, да и мы сами можем перенастроить наш интерфейс на другую библиотеку. В любом случае не стоит замусоривать свой код переменными типа `int` только потому, что мы знаем, как они задают стиль линий.

Как правило, мы не заботимся о стиле вообще; мы просто полагаемся на параметры, заданные по умолчанию (сплошные линии, ширина которых задана по умолчанию). Если мы не указываем ширину линии явно, то она задается конструктором. Установка значений по умолчанию — это одно из предназначений конструктора, а правильно выбранные значения, задаваемые по умолчанию, могут значительно облегчить работу пользователей класса.

Класс `Line_style` состоит из двух "компонентов": характеристики стиля (например, пунктирные или сплошные линии) и ширины (толщина линий). Ширина измеряется в целых числах. По умолчанию ширина равна единице. Если нам нужна более широкая линия, то ее толщину можно задать следующим образом:

```
grid.set_style(Line_style(Line_style::dash,2));
```

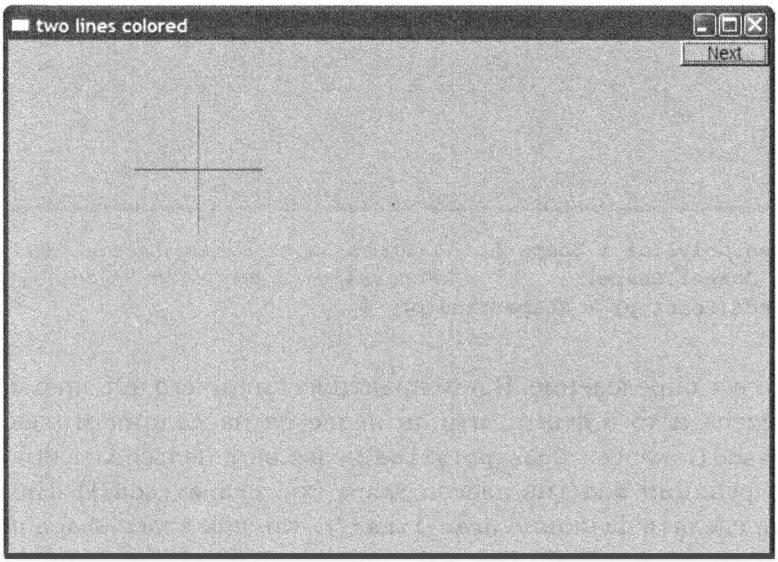
В итоге получим следующее изображение.



Обратите внимание на то, что цвет и стиль относятся ко всем линиям, образующим фигуру. Это одно из преимуществ группирования нескольких линий в один графический объект, например класс **Lines**, **Open_polyline** или **Polygon**. Если мы хотим управлять цветом или стилем линий по отдельности, то их следует задать как отдельные объекты класса **Line**. Рассмотрим пример.

```
horizontal.set_color(Color::red);
vertical.set_color(Color::green);
```

Вы получите окно, приведенное ниже.



13.6. Класс Open_polyline

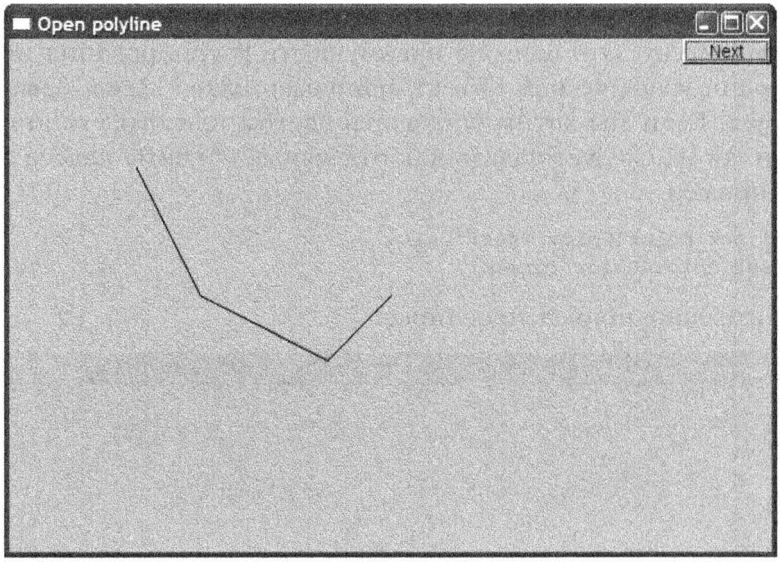
Класс **Open_polyline** определяет фигуру, состоящую из ряда отрезков линий, соединенных между собой и заданных последовательностью точек. Слово *poly* имеет греческое происхождение и означает "много", а *polyline* — это удобное имя для фигуры, состоящей из многих линий. Рассмотрим пример.

```
Open_polyline opl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

Этот фрагмент кода создает фигуру, которую можно нарисовать, соединяя указанные четыре точки:

Вообще говоря, **Open_polyline** — это выдуманное слово, которое мы заимствовали из детской игры "Connect the Dots" ("Соедини точки").

Класс **Open_polyline** определен следующим образом.




```
struct Open_polyline : Shape { // Открытая последовательность линий
    using Shape::Shape;        // Использует конструктор Shape (раздел A.16)
    void add(Point p) { Shape::add(p); }
};
```

Да-да, это все определение. В нем практически ничего нет, кроме указания имени класса и того факта, что он является наследником класса **Shape**. Функция **add()** класса **Open_polyline** позволяет пользователям получить доступ к функции **add()** из класса **Shape** (т.е. **Shape::add()**). Нам даже не нужно определять функцию **draw_lines()**, так как класс **Shape** по умолчанию интерпретирует добавленные точки как последовательность линий, соединенных между собой.

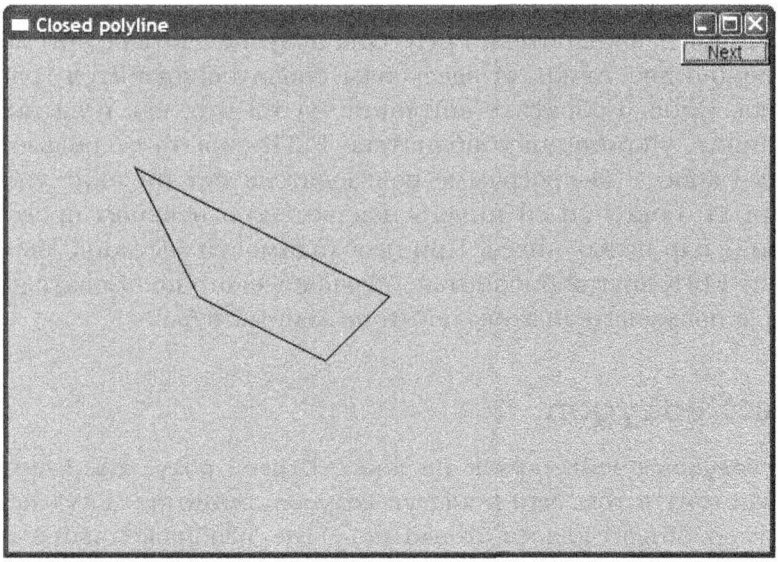
Объявление **using Shape::Shape** представляет собой **using**-объявление. Оно гласит, что класс **Open_polyline** может использовать конструкторы, определенные в классе **Shape**. Класс **Shape** имеет конструктор по умолчанию (раздел 9.7.3) и конструктор со списком инициализации (раздел 18.2), так что **using**-объявление представляет собой просто сокращение для определения двух этих конструкторов в классе **Open_polyline**. Как и в случае класса **Lines**, конструктор со списком инициализации является сокращением для инициализирующей последовательности вызовов **add()**.

13.7. Класс Closed_polyline

Класс **Closed_polyline** похож на класс **Open_polyline**, за исключением того, что последняя точка соединяется с первой. Например, можно было бы создать объект класса **Closed_polyline** из тех же точек, из которых был построен объект класса **Open_polyline** в разделе 13.6.

```
Closed_polyline cpl = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

Как и ожидалось, результат идентичен тому, который мы получили в разделе 13.6, за исключением последнего отрезка.



Определение класса **Closed_polyline** имеет следующий вид.

```
struct Closed_polyline
        : Open_polyline {                  // Замкнутая последовательность линий
  using Open_polyline::Open_polyline; // Использует конструкторы
                                           // Open_polyline (A.16)
  void draw_lines() const;
};

void Closed_polyline::draw_lines() const
{
    Open_polyline::draw_lines();  // Вывод открытой последовательности,
    // Затем – вывод замыкающей линии
    if (2<number_of_points() && color().visibility())
        fl_line(point(number_of_points()-1).x,
                point(number_of_points()-1).y,
                point(0).x,
                point(0).y);
}
```

Использованное **using**-объявление (раздел A.16) гласит, что **Closed_polyline** имеет те же конструкторы, что и **Open_polyline**. Классу **Closed_polyline** нужна собственная функция **draw_lines()**, рисующая замыкающую линию, которая соединяет последнюю точку с первой.

К счастью, для этого достаточно реализовать небольшую деталь, которая отличает класс `Closed_polyline` от класса `Open_polyline`. Этот важный прием иногда называют "программированием различий" ("programming by difference"). Нам нужно запрограммировать лишь то, что отличает наш производный класс (`Closed_polyline`) от базового (`Open_polyline`).

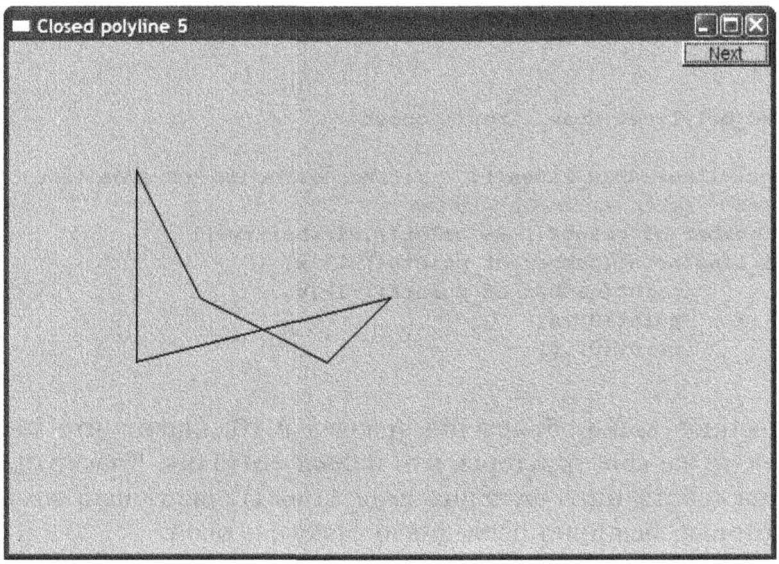
Итак, как же нарисовать замыкающую линию? Воспользуемся функцией `fl_line()` из библиотеки FLTK. Она получает четыре аргумента типа `int`, задающие две точки. И здесь нам снова понадобится графическая библиотека. Однако обратите внимание на то, что, как и во многих других ситуациях, упоминание библиотеки FLTK скрыто от пользователей в реализации класса. В программе пользователя нет никаких упоминаний о функции `fl_line()`, и ей ничего неизвестно о неявном представлении точек в виде пар целых чисел. При необходимости мы могли бы заменить библиотеку FLTK другой библиотекой графического пользовательского интерфейса, а пользователи этого почти не заметили бы.

13.8. Класс `Polygon`

Класс `Polygon` очень похож на класс `Closed_polyline`. Единственная разница состоит в том, что в классе `Polygon` линии не могут пересекаться. Например, объект класса `Closed_polyline`, изображенный выше, представляет собой многоугольник, но если к нему добавить еще одну точку, то ситуация изменится.

```
cpl.add(Point(100,250));
```

Результат изображен ниже.



В соответствии с классическими определениями объект класса **Closed_polyline** многоугольником не является. Как определить класс **Polygon** так, чтобы он правильно отображал связь с классом **Closed_polyline**, не нарушая правил геометрии? Подсказка содержится в предыдущем описании. Класс **Polygon** — это класс **Closed_polyline**, в котором линии не пересекаются. Иначе говоря, мы могли бы подчеркнуть способ образования фигуры из точек и сказать, что класс **Polygon** — это класс **Closed_polyline**, в который невозможно добавить точку **Point**, определяющую отрезок линии, пересекающийся с одним из существующих отрезков объекта **Polygon**.

Эта идея позволяет описать класс **Polygon** следующим образом:

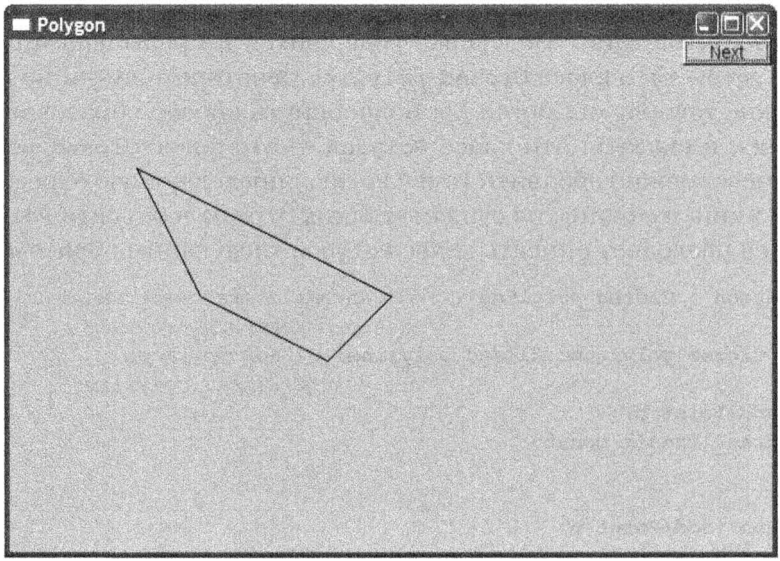
```
struct Polygon : Closed_polyline {// Замкнутая последовательность
                                  // непересекающихся отрезков
    using Closed_polyline::Closed_polyline; // Конструкторы
                                            // Closed_polyline
    void add(Point p);
    void draw_lines() const;
};

void Polygon::add(Point p)
{
    // Проверка того, что новая линия не
    // пересекает существующие (код не показан)
    Closed_polyline::add(p);
}
```

Здесь мы унаследовали определение функции **draw_lines()** от класса **Closed_polyline**, сэкономив усилия и избежав дублирования кода. К сожалению, мы должны проверить каждый вызов функции **add()**. Это приводит нас к неэффективному алгоритму, сложность которого оценивается как N^2, — определение объекта класса **Polygon**, состоящего из N точек, требует $N \cdot (N-1)/2$ вызовов функции **intersect()**. По существу, мы сделали предположение, что класс **Polygon** будет использоваться для создания многоугольников с небольшим количеством точек. Например, для того чтобы создать объект класса **Polygon**, состоящего из 24 точек, потребуется $24 \cdot (24-1)/2 = 276$ вызовов функции **intersect()**. Вероятно, это приемлемая величина, но если бы мы захотели создать многоугольник из 2000 точек, то вынуждены были бы сделать около 2 000 000 вызовов. В таком случае нам надо поискать более эффективный алгоритм, который может потребовать изменения интерфейса.

С помощью конструктора со списком инициализации можно создавать многоугольники следующим образом:

```
Polygon poly = {
    {100,100}, {150,200}, {250,250}, {300,200}
};
```

Очевидно, что этот фрагмент создает объект класса `Polygon`, идентичный (вплоть до последнего пикселя) исходному объекту класса `Closed_polyline`.



Проверка того, что объект класса `Polygon` действительно представляет собой многоугольник, оказывается на удивление запутанной. Проверка пересечений, которая реализована в функции `Polygon::add()`, является наиболее сложной во всей графической библиотеке. Если вас интересуют кропотливые манипуляции координатами, просто взгляните на соответствующий код.

Проблема с классом `Polygon` заключается в том, что его инвариант — "точки образуют многоугольник" — невозможно проверить, пока не будут определены все точки. Иначе говоря, несмотря на строгую рекомендацию, мы не выполняем проверку инварианта в конструкторе класса `Polygon`. Мы рассматривали вариант с удалением `add()` и требованием, чтобы многоугольник полностью определялся списком инициализации как минимум с тремя точками, но это усложняет применение данного класса там, где программа генерирует последовательности точек.

13.9. Класс `Rectangle`

Большинство фигур на экране являются прямоугольниками. Причина этого явления объясняется частично культурными традициями (большинство дверей, окон, картин, книжных шкафов, страниц и тому подобного является прямоугольниками), а частично техническими особенностями (задать координаты прямоугольника проще, чем любой другой фигуры). В любом случае прямоугольник настолько широко распространен, что в

системах графического пользовательского интерфейса он обрабатывается непосредственно, а не как многоугольник, имеющий четыре прямых угла.

```
struct Rectangle : Shape {
    Rectangle(Point xy, int ww, int hh);
    Rectangle(Point x, Point y);
    void draw_lines() const;

    int height() const { return h; }
    int width()  const { return w; }
private:
    int h; // Высота
    int w; // Ширина
};
```

Мы можем задать прямоугольник двумя точками (левой верхней и правой нижней) или одной точкой, шириной и высотой. Конструкторы этого класса могут иметь следующий вид:

```
Rectangle::Rectangle(Point xy, int ww, int hh)
    : w{ww}, h{hh}
{
    if (h<=0 || w<=0)
        error("Ошибка: не положительная сторона");
    add(xy);
}

Rectangle::Rectangle(Point x, Point y)
    :w{y.x-x.x}, h{y.y-x.y}
{
    if (h<=0 || w<=0)
        error("Ошибка: первая точка не верхняя левая");
    add(x);
}
```

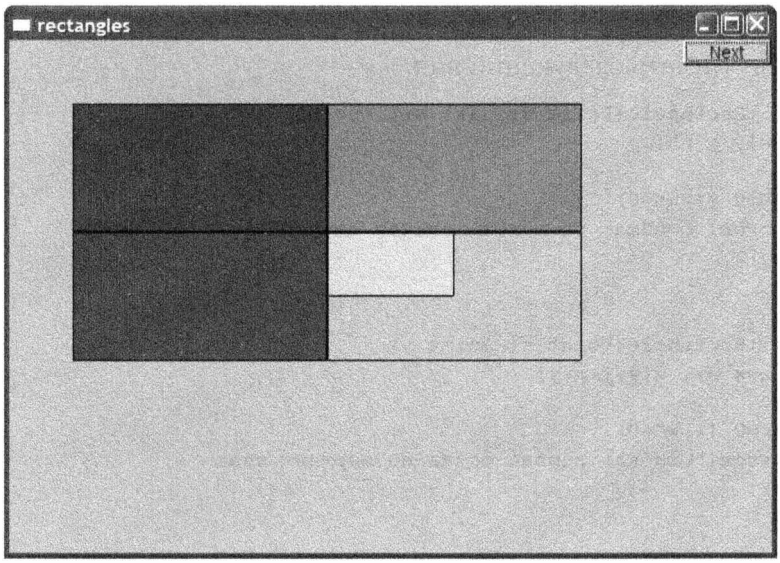
Каждый конструктор соответствующим образом инициализирует члены `h` и `w` (используя синтаксис инициализации члена; см. раздел 9.4.4) и хранит верхнюю левую точку отдельно в базовом классе `Shape` (используя функцию `add()`). Кроме того, в конструкторах содержится простейшая проверка ширины и длины — они не должны быть отрицательными.

Одна из причин, по которым некоторые графические системы рассматривают прямоугольники как отдельные фигуры, заключается в том, что алгоритм определения того, какие пиксели попадают внутрь прямоугольника, намного проще и, следовательно, намного быстрее, чем алгоритмы проверки для других фигур, таких как `Polygon` и `Circle`. По этой причине понятие "заполнение цветом" — т.е. закраска пространства внутри прямоугольника — чаще применяется по отношению к прямоугольникам, чем к другим фигурам. Заполнение цветом можно реализовать в конструкторе или в виде отдельной функции `set_fill_color()`

(предусмотренной в классе **Shape** наряду с другими средствами для работы с цветом).

```
Rectangle rect00{Point{150,100},200,100};
Rectangle rect11{Point{50,50},Point{250,150}};
Rectangle rect12{Point{50,150},Point{250,250}}; // Под rect11
Rectangle rect21{Point{250,50},200,100};        // Правее rect11
Rectangle rect22{Point{250,150},200,100};       // Под rect21

rect00.set_fill_color(Color::yellow);
rect11.set_fill_color(Color::blue);
rect12.set_fill_color(Color::red);
rect21.set_fill_color(Color::green);
```

В итоге получаем следующее изображение.



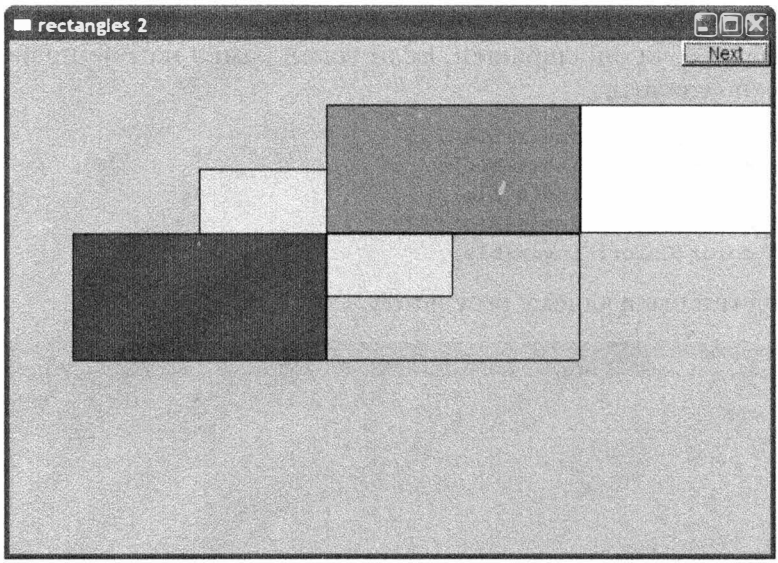
Если заполнение цветом не требуется, то прямоугольник считается прозрачным; вот почему вы видите угол объекта **rect00**.

Фигуры в окне можно передвигать (см. раздел 14.2.3). Рассмотрим пример.

```
rect11.move(400,0);                  // Справа от rect21
rect11.set_fill_color(Color::white);
win12.set_label("rectangles 2");
```

В результате мы получаем следующее изображение.

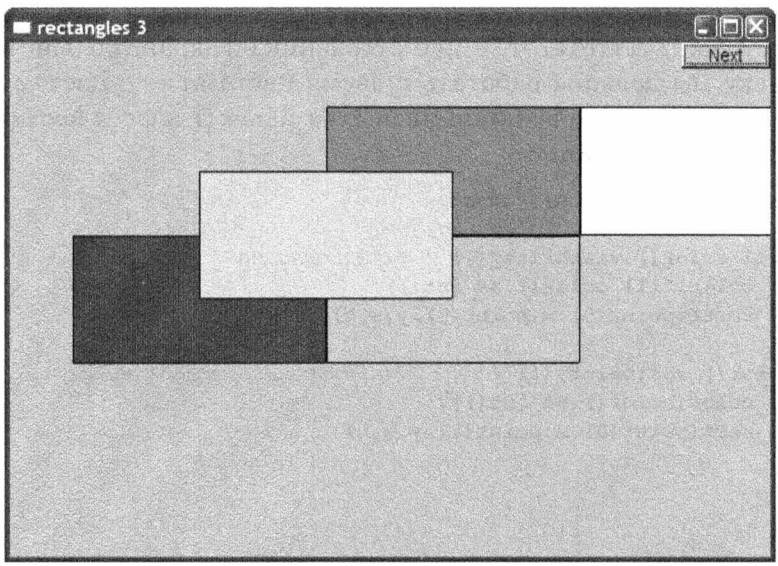
Заметьте, что в окне помещается только часть белого прямоугольника **rect11**. То, что выходит за пределы окна, "обрезается"; иначе говоря, на экране эта часть не отображается.



Обратите внимание на то, как фигуры накладываются одна на другую. Это выглядит так, будто на столе вы кладете один лист бумаги на другой. Первый лист окажется в самом низу. Наш класс **Window** (раздел Д.3) реализует простой способ размещения фигуры поверх другой (используя функцию **Window::put_on_top()**). Так, при использовании фрагмента

```
win12.put_on_top(rect00);
win12.set_label("rectangles 3");
```

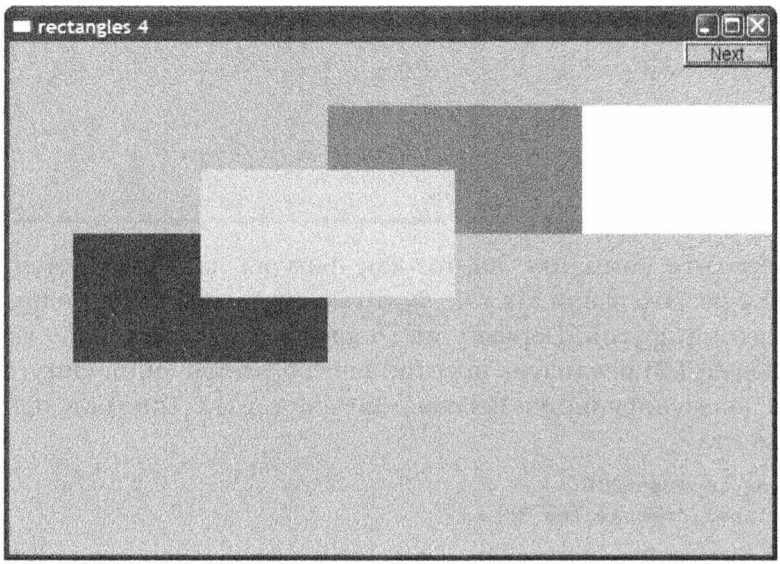
мы получим следующее изображение.



Обратите внимание, что мы можем видеть линии, образующие прямоугольник, даже если он закрашен. Если такая рамка нам не нравится, эти линии можно удалить.

```
rect00.set_color(Color::invisible);
rect11.set_color(Color::invisible);
rect12.set_color(Color::invisible);
rect21.set_color(Color::invisible);
rect22.set_color(Color::invisible);
```

Это приводит к следующему результату.



Поскольку у прямоугольника **rect22** и цвет заполнения, и цвет линий заданы параметром **invisible**, этот прямоугольник на экране больше не виден.

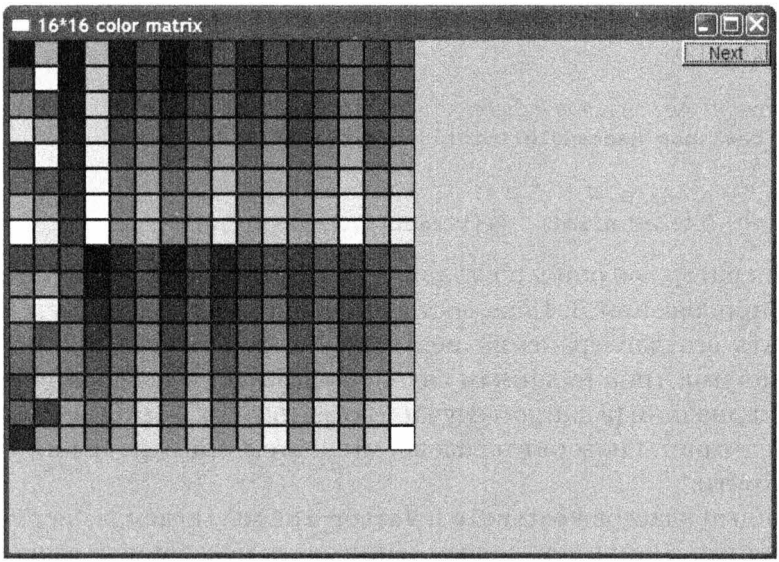
Поскольку мы должны работать с двумя цветами — с цветом линий и с цветом заполнения, — функция-член **draw_lines()** класса **Rectangle** становится немного запутаннее.

```
void Rectangle::draw_lines() const
{
    if (fill_color().visibility()) { // Заполнение
        fl_color(fill_color().as_int());
        fl_rectf(point(0).x,point(0).y,w,h);
    }
    if (color().visibility()) {      // Линии поверх заполнения
        fl_color(color().as_int());
        fl_rect(point(0).x,point(0).y,w,h);
    }
}
```

Как видим, библиотека FLTK содержит функции для рисования как заполненных прямоугольников (`fl_rectf()`), так и пустых (`fl_rect()`). По умолчанию рисуются оба прямоугольника (с линиями рамки поверх заполненного).

13.10. Управление неименованными объектами

До сих пор мы именовали все наши графические объекты. Когда же объектов много, присваивать всем им имена становится нецелесообразно. В качестве примера нарисуем простую цветную диаграмму, состоящую из 256 цветов, предусмотренных в палитре библиотеки; иначе говоря, сделаем 256 цветных квадратов и нарисуем их в матрице 16×16.



Именовать все эти 256 квадратов было бы не только утомительно, но и глупо. Очевидно, что "имя" левого верхнего квадрата в матрице определяется его местоположением (0,0) в матрице, а все остальные квадраты можно точно так же идентифицировать с помощью пар координат (*i*, *j*). Таким образом, в данном примере нам необходим эквивалент матрицы объектов. Сначала мы подумали было о применении вектора `vector<Rectangle>`, но оказалось, что он недостаточно гибок. Например, было бы неплохо иметь коллекцию неименованных объектов (элементов), не все из которых имеют одинаковые типы. Проблему гибкости мы обсудим в разделе 14.3, а здесь продемонстрируем наше решение: векторный тип, который может хранить именованные и неименованные объекты:

```
template<class T> class Vector_ref {
public:
```

```
    // ...
    void push_back(T&);     // Добавляет именованный объект
    void push_back(T*);     // Добавляет неименованный объект

    T& operator[](int i);  // Индексация: доступ для записи
    const T& operator[](int i) const; // и чтения

    int size() const;
};
```

Использование этого класса очень похоже на использование типа **vector** из стандартной библиотеки.

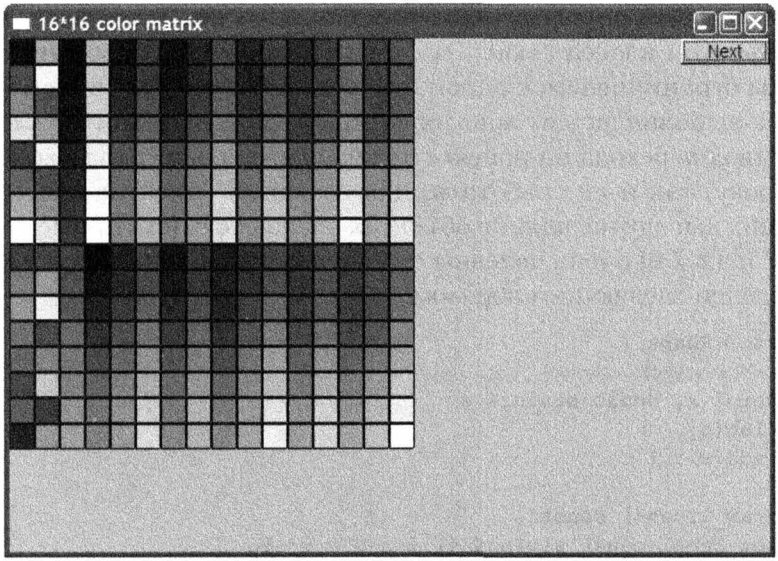
```
Vector_ref<Rectangle> rect;
Rectangle x(Point(100,200),Point(200,300));

// Добавляем именованный объект
rect.push_back(x);

// Добавляем неименованный объект
rect.push_back(new Rectangle(Point(50,60),Point(80,90)));

// Используем объекты
for (int i=0; i<rect.size(); ++i) rect[i].move(10,10);
```

Оператор **new** описан в главе 17, а реализация класса **Vector_ref** — в приложении Д. Пока достаточно знать, что мы можем использовать его для хранения неименованных объектов. За оператором **new** следует имя типа (в данном случае **Rectangle**) и необязательный список инициализации (в данном случае **{Point(50,60),Point(80,90)}**). Опытные программисты поверят на слово, что в этом примере мы не допускаем утечки памяти.

С помощью классов **Rectangle** и **Vector_ref** мы можем экспериментировать с цветами. Например, можем нарисовать показанную выше простую диаграмму, состоящую из 256 цветов.

```
Vector_ref<Rectangle> vr;

for (int i = 0; i<16; ++i)
    for (int j = 0; j<16; ++j) {
        vr.push_back(new Rectangle(Point(i*20,j*20),20,20));
        vr[vr.size()-1].set_fill_color(Color(i*16+j));
        win20.attach(vr[vr.size()-1]);
    }
```

Мы создали объект класса **Vector_ref**, состоящий из 256 объектов класса **Rectangle**, организованных графически в объекте **Window** в виде матрицы 16×16. Мы назначаем объектам класса **Rectangle** цвета 0, 1, 2, 3, 4 и т.д. После этого каждый из созданных объектов этого типа выводится на экран.

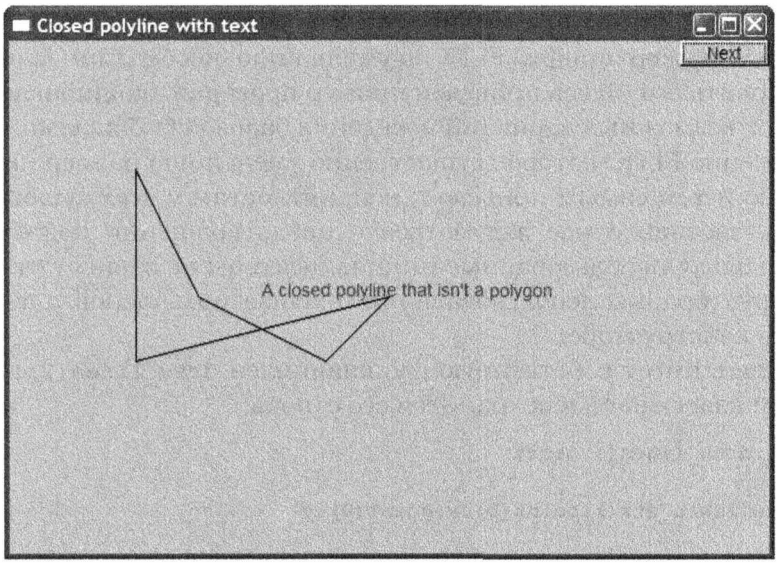


13.11. Класс `Text`

Очевидно, что нам необходимо уметь выводить на экран текст. Например, мы могли бы пометить "странный" объект класса `Closed_polyline` из раздела 13.8

```
Text t(Point(200,200),"A closed polyline that isn't a polygon");
t.set_color(Color::blue);
```

получив в результате следующее изображение.



Объект класса **Text** определяет строку текста, начинающегося с точки, заданной объектом класса **Point**. Эта точка находится в левом нижнем углу текста. Мы ограничиваемся одной строкой, поскольку хотим, чтобы наша программа выполнялась на многих компьютерах. Не пытайтесь вставлять в окно символ перехода на новую строку; в вашем окне это может как дать новую строку, так и не сработать. Для создания объектов класса **string**, подлежащих выводу на экран в объектах класса **Text** (см. примеры в разделах 12.7.7 и 12.7.8) очень полезны строковые потоки (см. раздел 11.4). Тип **Text** определен следующим образом.

```
struct Text : Shape {
    // Точка в левом нижнем углу первой буквы
    Text(Point x, const string& s)
        : lab{s},
        { add(x); }

    void draw_lines() const;
    void set_label(const string& s) { lab = s; }
    string label() const { return lab; }

    void set_font(Font f) { fnt = f; }
    Font font() const { return fnt; }

    void set_font_size(int s) { fnt_sz = s; }
    int font_size() const { return fnt_sz; }
private:
    string lab;   // Текст
    Font fnt   {fl_font()};
    int fnt_sz {(fl_size()<14)?14:fl_size()};
};
```

Если вы хотите, чтобы размер шрифта мог быть меньше 14 пунктов или больше, чем размер шрифта FLTK по умолчанию, необходимо явным образом установить его. Здесь приведен пример проверки, защищающей пользователя от возможных вариаций поведения базовой библиотеки. Имевшееся обновление FLTK, которое существенно уменьшило размер шрифта по умолчанию и тем самым привело к неприятностям у всех существующих программ, заставило нас задуматься о предотвращении подобных проблем. Поскольку использованные инициализаторы не зависят от аргументов конструктора, мы использовали инициализаторы членов, а не инициализацию в конструкторе.

Класс **Text** имеет собственную функцию-член **draw_lines()**, поскольку только сам класс знает, как хранится его строка.

```
void Text::draw_lines() const
{
    fl_draw(lab.c_str(),point(0).x,point(0).y);
}
```

Цвет символов определяется точно так же, как в фигурах, состоящих из линий (например, `Open_polyline` и `Circle`), поэтому можем выбирать новый цвет с помощью функции `set_color()`, а выяснять, какой цвет используется в качестве текущего, — с помощью функции `color()`. Размер и шрифт символов выбираются аналогично. В классе предусмотрено небольшое количество предопределенных шрифтов.

```
class Font { // Шрифт символа
public:
    enum Font_type {
        helvetica             =FL_HELVETICA,
        helvetica_bold        =FL_HELVETICA_BOLD,
        helvetica_italic      =FL_HELVETICA_ITALIC,
        helvetica_bold_italic=FL_HELVETICA_BOLD_ITALIC,
        courier               =FL_COURIER,
        courier_bold          =FL_COURIER_BOLD,
        courier_italic        =FL_COURIER_ITALIC,
        courier_bold_italic   =FL_COURIER_BOLD_ITALIC,
        times                 =FL_TIMES,
        times_bold            =FL_TIMES_BOLD,
        times_italic          =FL_TIMES_ITALIC,
        times_bold_italic     =FL_TIMES_BOLD_ITALIC,
        symbol                =FL_SYMBOL,
        screen                =FL_SCREEN,
        screen_bold           =FL_SCREEN_BOLD,
        zapf_dingbats         =FL_ZAPF_DINGBATS
    };

    Font(Font_type ff) :f{ff} {}
    Font(int ff) :f{ff} {}

    int as_int() const { return f; }
private:
    int f;
};
```

Стиль определения класса `Font` совпадает со стилями определения классов `Color` (см. раздел 13.4) и `Line_style` (см. раздел 13.5).

13.12. Класс `Circle`

Просто для того, чтобы показать, что мир не состоит из одних прямых углов, мы создали классы `Circle` и `Ellipse`. Объект класса `Circle` определяется центром и радиусом.

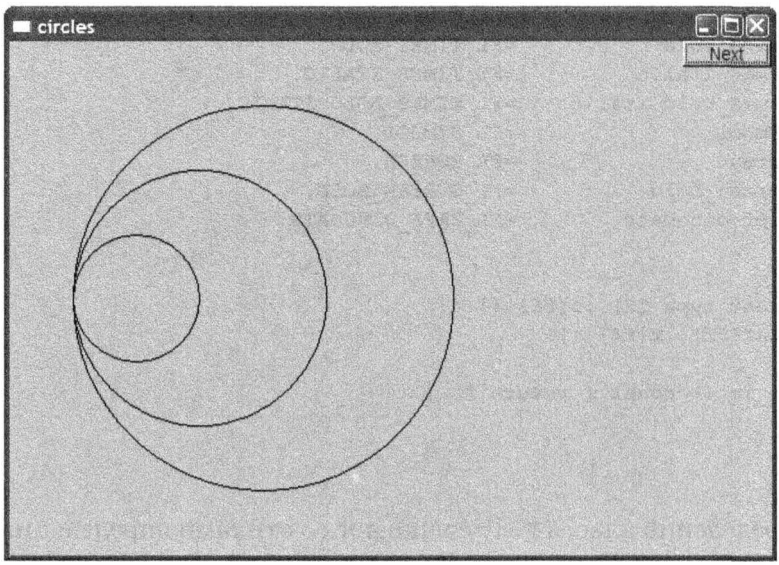
```
struct Circle : Shape {
    Circle(Point p, int rr); // Центр и радиус

    void draw_lines() const;
```

```
    Point center() const ;
    int radius() const { return r; }
    void set_radius(int rr)
    {   // Поддержка центра
        set_point(0,Point{center().x-rr,center().y-rr});
        r=rr;
    }
private:
    int r;
};
```

Использовать класс `Circle` можно следующим образом:

```
Circle c1{Point{100,200},50};
Circle c2{Point{150,200},100};
Circle c3{Point{200,200},150};
```

Эти инструкции рисуют три окружности разных радиусов, центры которых лежат на горизонтальной линии.



Основной особенностью реализации класса `Circle` является то, что в нем хранится не центр, а левая верхняя точка угла квадрата, окаймляющего окружность. Можно было бы хранить и центр окружности, но мы выбрали вариант, позволяющий библиотеке FLTK оптимизировать процесс рисования окружности. Это еще один пример того, как с помощью класса можно создать другое (предположительно, более точное) представление концепции, реализацией которой он является.

```
Circle::Circle(Point p, int rr) // Центр и радиус
    :r(rr)
{
```

```
    add(Point(p.x-r,p.y-r));     // Хранит левый верхний угол
}

Point Circle::center() const
{
    return (point(0).x+r, point(0).y+r);
}

void Circle::draw_lines() const
{
    if (color().visibility())
        fl_arc(point(0).x,point(0).y,r+r,r+r,0,360);
}
```

Обратите внимание на использование функции `fl_arc()`, рисующей окружность. Первые два аргумента задают левый верхний угол, вторые два — ширину и высоту наименьшего прямоугольника, окаймляющего окружность, а последние два аргумента задают начальный и конечный углы дуги. Для того чтобы нарисовать окружность, нужно обойти вокруг ее центра все 360 градусов, но с помощью функции `fl_arc()` можно нарисовать только часть окружности (как и часть эллипса); см. упр. 1.

13.13. Класс `Ellipse`

Эллипс похож на окружность, но он определяется большой и малой осями, а не радиусом. Иначе говоря, для того чтобы определить эллипс, мы должны задать координаты центра, а также расстояние от центра до точки на оси x и расстояние от центра до точки на оси y.

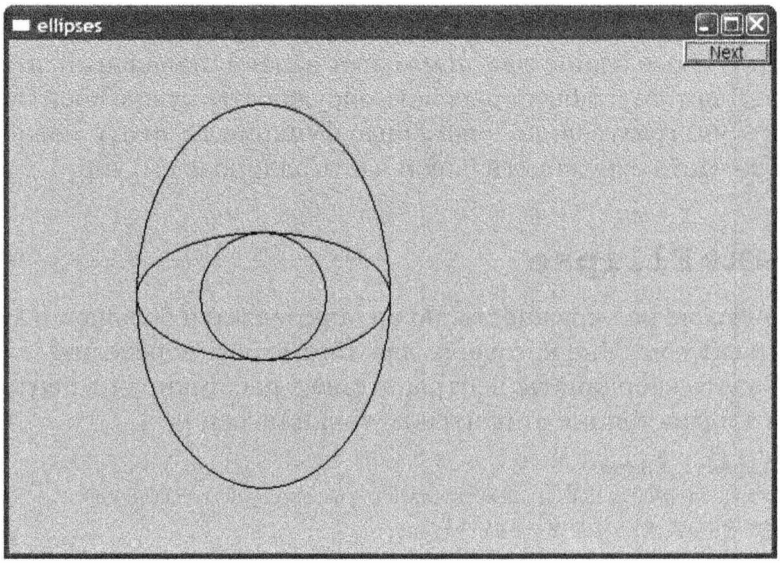
```
struct Ellipse : Shape {
    // Центр, минимальное и максимальное расстояния от центра
    Ellipse(Point p, int w, int h);

    void draw_lines() const;

    Point center() const;
    Point focus1() const;
    Point focus2() const;

    void set_major(int ww)
    {   // Поддержка центра
        set_point(0,Point(center().x-ww,center().y-h);
        w=ww;
    }
    int major() const { return w; }

    void set_minor(int hh)
    {   // Поддержка центра
        set_point(0,Point(center().x-w,center().y-hh));
```

```
        h=hh;
    }
    int minor() const { return h; }
private:
    int w;
    int h;
};
```

Класс **Ellipse** можно использовать следующим образом:

```
Ellipse e1{Point{200,200},50,50};
Ellipse e2{Point{200,200},100,50};
Ellipse e3{Point{200,200},100,150};
```

Этот фрагмент программы рисует три эллипса с общим центром и разными осями.



Объект класса **Ellipse**, для которого выполняется условие **major()== minor()**, выглядит как окружность.

Эллипс можно также задать с помощью двух фокусов и суммы расстояний от точки до фокусов. Имея объект класса **Ellipse**, можем вычислить фокус:

```
Point Ellipse::focus1() const
{
    if (h<=w) // Фокусы на оси x:
        return {center().x+int(sqrt(double(w*w-h*h))),
                center().y};
    else      // Фокусы на оси y:
        return {center().x,
                center().y+int(sqrt(double(h*h-w*w)))};
}
```

Почему класс `Circle` не является наследником класса `Ellipse`? С геометрической точки зрения каждая окружность является эллипсом, но не каждый эллипс является окружностью. В частности, окружность — это эллипс, у которого оба фокуса совпадают. Представьте себе, что мы определили класс `Circle` как разновидность класса `Ellipse`. В этом случае нам пришлось включать в представление дополнительные величины (окружность определяется центром и радиусом; для определения эллипса необходимы центр и пара осей). Мы не приветствуем излишние затраты памяти там, где они не нужны, но основная причина, по которой класс `Circle` не сделан наследником класса `Ellipse`, заключается в том, что мы не можем определить его, не заблокировав каким-то образом функции `set_major()` и `set_minor()`. Кроме того, фигура не была бы окружностью (что легко распознают математики), если бы мы использовали функцию `set_major()`, чтобы обеспечить неравенство `major()!=minor()`, — по крайней мере, после этого фигура перестанет быть окружностью. Нам не нужен объект, который иногда относится к одному типу (когда `major()!=minor()`), а иногда к другому (когда `major()==minor()`). Нам нужен объект (класса `Ellipse`), который иногда выглядит как окружность. С другой стороны, объект класса `Circle` никогда не превратится в эллипс с двумя неравными осями.

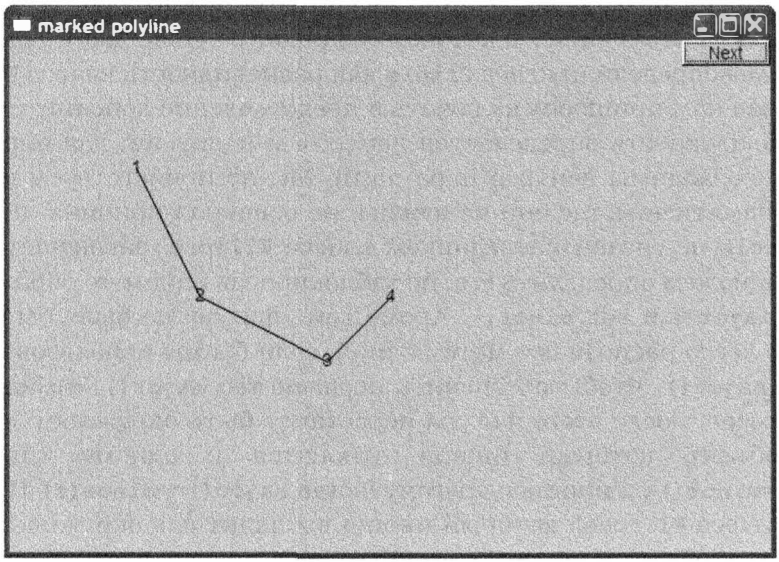
Разрабатывая классы, мы должны быть осторожны: не слишком умничать и не слишком полагаться на интуицию. И наоборот, мы должны быть уверены, что наш класс представляет некое осмысленное понятие, а не является просто коллекцией данных и функций-членов. Механическое объединение фрагментов кода без размышлений об идеях и концепциях, которые они представляют, — это хакерство. Оно приводит к программам, которые невозможно объяснить и поддерживать без участия автора. Если вы не альтруист, то просто помните, что в роли ничего не понимающего пользователя через несколько месяцев можете оказаться вы сами. Кроме того, такие программы труднее отлаживать.

13.14. Класс `Marked_polyline`

Часто возникает необходимость пометить точки графика. График можно изобразить в виде ломаной, поэтому нам нужна ломаная, точки которой имели бы метки. Для этого предназначен класс `Marked_polyline`. Рассмотрим пример.

```
Marked_polyline mpl {"1234"};
mpl.add(Point{100,100});
mpl.add(Point{150,200});
mpl.add(Point{250,250});
mpl.add(Point{300,200});
```

В результате выполнения этого фрагмента программы получим следующий результат.



Определение класса `Marked_polyline` имеет следующий вид:

```
struct Marked_polyline : Open_polyline {
    Marked_polyline(const string& m) :mark{m}
    {
        if (m=="") mark = "*";
    }
    Marked_polyline(const string& m,
                    initializer_list<Point> lst);
    void draw_lines() const;
private:
    string mark;
};
```

Поскольку этот класс является наследником класса `Open_polyline`, мы получаем "бесплатную" обработку объектов класса `Point`, и все, что нам надо сделать, — это обработать метки. В частности, функция `draw_lines()` принимает следующий вид:

```
void Marked_polyline::draw_lines() const
{
     Open_polyline::draw_lines();
     for (int i=0; i<number_of_points(); ++i)
          draw_mark(point(i),mark[i%mark.size()]);
}
```

Вызов функции `Open_polyline::draw_lines()` рисует линии, так что остается просто расставить метки. Эти метки представляют собой строки

символов, которые используются в определенном порядке: команда `mark[i%mark.size()]` выбирает символ, который должен быть использован следующим, циклически перебирая символы, хранящиеся в объекте класса `Marked_polyline`. Оператор `%` означает деление по модулю (взятие остатка). Для вывода буквы в заданной точке функция `draw_lines()` использует вспомогательную функцию меньшего размера `draw_mark()`.

```
void draw_mark(Point xy, char c)
{
     constexpr int dx = 4;
     constexpr int dy = 4;
     string m{1,c};     // Строка с единственным символом c
     fl_draw(m.c_str(),xy.x-dx,xy.y+dy);
}
```

Константы `dx` и `dy` используются для центрирования буквы относительно заданной точки. Объект `m` класса хранит единственный символ `c`.

Конструктор, принимающий список инициализации, просто передает его конструктору со списком инициализации класса `Open_polyline`:

```
Marked_polyline(const string& m, initializer_list<Point> lst)
    :Open_polyline{lst}, mark{m}
{
    if (m=="") mark = "*";
}
```

Проверка на пустоту строки необходима для того, чтобы избежать ситуации, когда функция `draw_lines()` будет пытаться обратиться к несуществующему символу.

При наличии конструктора, принимающего список инициализации, пример можно сократить до

```
Marked_polyline mpl {"1234",{{100,100}, {150,200},
                             {250,250}, {300,200}}};
```

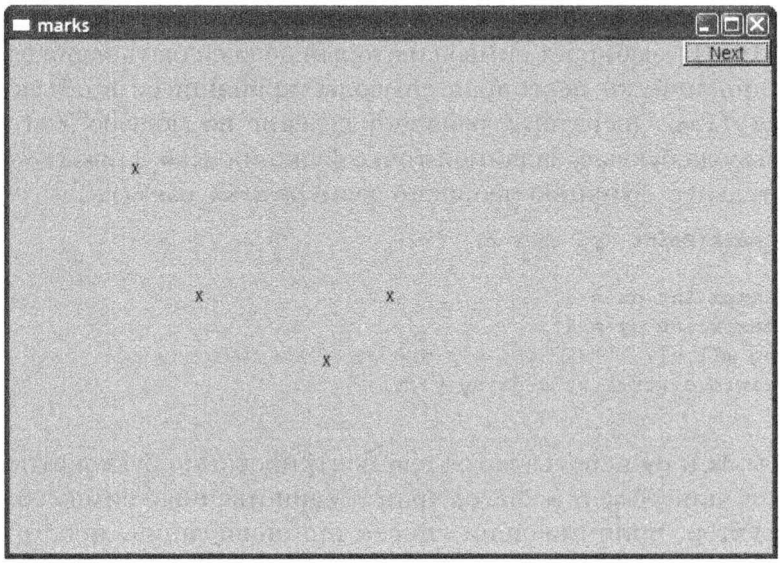
13.15. Класс `Marks`

Иногда необходимо вывести метки отдельно от линий. Для этого предназначен класс `Marks`. Например, мы можем пометить четыре точки, использованные в предыдущих примерах, не соединяя их линиями.

```
Marks pp {"x",{{100,100}, {150,200}, {250,250}, {300,200}}};
```

В итоге будет получено следующее изображение:

Очевидно, что класс `Marks` можно использовать для отображения дискретных событий, изображать которые с помощью ломаной было бы неуместно. В качестве примера можно привести пары (рост, вес), характеризующие группу людей.



Класс **Marks** — это разновидность класса **Marked_polyline** с невидимыми линиями.

```
struct Marks : Marked_polyline {
    Marks(const string& m) : Marked_polyline{m}
    {
        set_color(Color{Color::invisible});
    }
    Marked_polyline(const string& m, initializer_list<Point> lst)
        : Marked_polyline{m,lst}
    {
        set_color(Color{Color::invisible});
    }
};
```

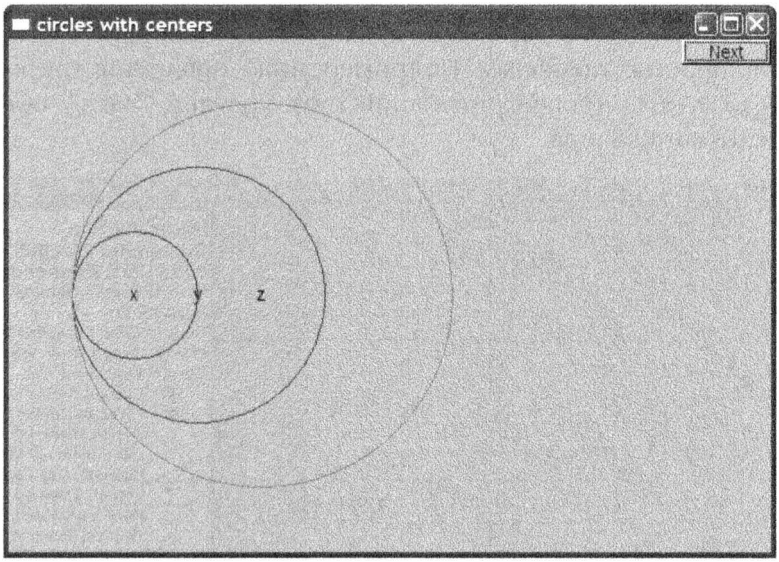
Запись **:Marked_polyline{m}** использована для инициализации части **Marked_polyline** объекта **Marks**. Эта запись представляет собой вариант синтаксиса, используемого для инициализации членов (§9.4.4).

13.16. Класс Mark

Объект класса **Point** задает координаты в объекте класса **Window**. Этот объект не является чем-то, что мы можем рисовать или видеть на экране. Если возникает необходимость пометить отдельную точку, чтобы ее увидеть, можно изобразить ее в виде пары линий, как в разделе 13.2, или с помощью класса **Marks**. Этот вариант несколько "многословен", поэтому у нас есть простая версия класса **Marks**, инициализируемая точкой и символом. Например, мы могли бы пометить центры окружностей, изображенных в разделе 13.12, следующим образом:

```
Mark m1 {Point{100,200},'x'};
Mark m2 {Point{150,200},'y'};
Mark m3 {Point{200,200},'z'};
c1.set_color(Color::blue);
c2.set_color(Color::red);
c3.set_color(Color::green);
```

В итоге мы получаем следующее изображение.



Класс **Mark** — это разновидность класса **Marks**, в котором непосредственно при создании объекта задается начальная точка.

```
struct Mark : Marks {
    Mark(Point xy, char c) : Marks{string{1,c}}
    {
        add(xy);
    }
};
```

Инструкция **string{1,c}** представляет собой конструктор класса **string**, инициализирующий строку, содержащую единственный символ **c**.

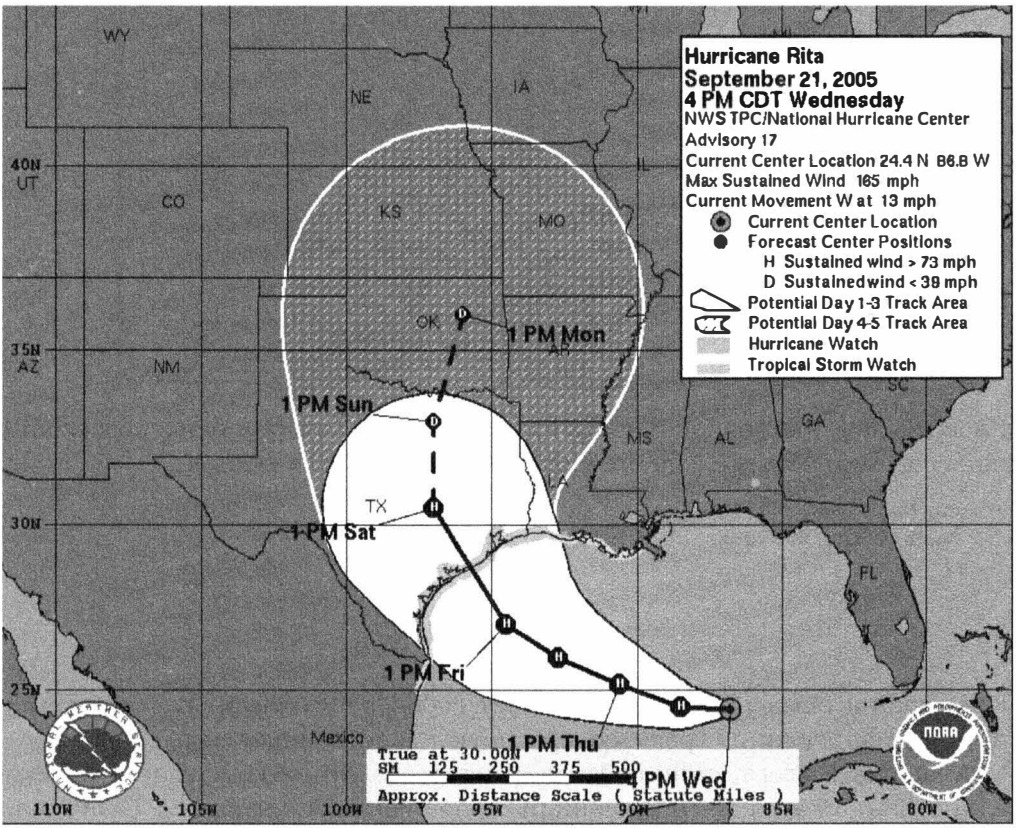
Класс **Mark** всего лишь позволяет легко создать объект класса **Marks** с единственной точкой, помеченной единственным символом. Стоило ли тратить силы, чтобы определять такой класс? Или он является следствием "ложного стремления к усложнениям и недоразумениям"? Однозначного и логичного ответа на этот вопрос нет. Мы много думали над этим и в конце концов решили, что для пользователей этот класс будет полезен, а определить его было совсем нетрудно.

Почему в качестве метки используется символ? Можно было бы нарисовать любую маленькую фигуру, но символы нагляднее и проще. Они часто

позволяют отделить одно множество точек от другого. К тому же такие символы, как **x**, **o**, **+** и *****, обладают центральной симметрией.

13.17. Класс `Image`

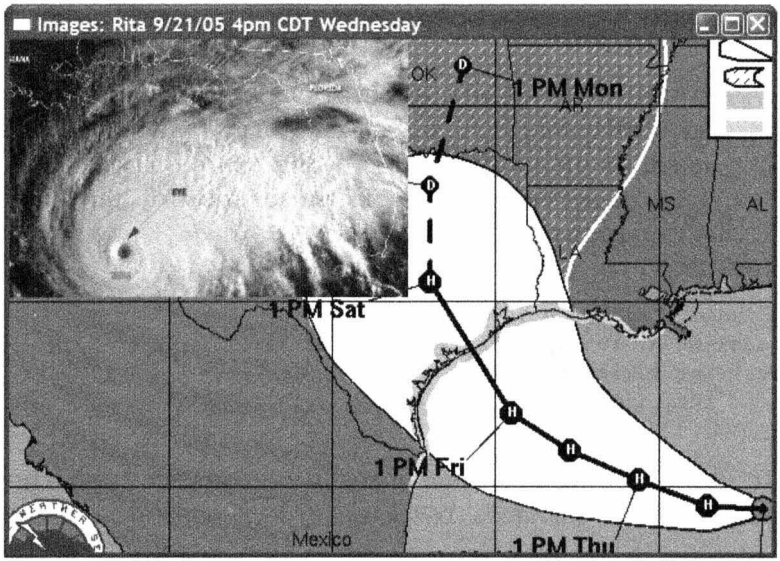
В типичном персональном компьютере хранятся тысячи изображений. Кроме того, миллионы изображений доступны в сети веб. Естественно, мы хотели бы отображать содержимое этих файлов на экране с помощью относительно простых программ. Например, ниже продемонстрирован рисунок (**`rita_path.gif`**), иллюстрирующий путь урагана "Рита", пришедшего из Мексиканского залива.



Мы можем выбрать часть этого изображения и добавить фотографию урагана, сделанную из космоса (**`rita.jpg`**).

```
Image rita {Point{0,0},"rita.jpg"};
Image path {Point{0,0},"rita_path.gif"};
path.set_mask(Point{50,250},600,400); // Выбор области
```

```
win.attach(path);
win.attach(rita);
```



Операция **set_mask()** выбирает часть рисунка, которую следует изобразить на экране. В данном случае мы выбрали изображение размером 600×400 пикселей из файла **rita_path.gif** (загруженного в объект **path**) и показали его в области, левый верхний угол которой имеет координаты (50,250). Выбор части рисунка — довольно распространенный прием, поэтому мы предусмотрели для него отдельную операцию.

Фигуры изображаются одна поверх другой, подобно листам бумаги, в порядке их добавления на экран. По этой причине объект **path** оказался на самом "дне", просто потому, что он был связан с окном до объекта **rita**.

Изображения могут кодироваться в множестве форматов. Здесь мы используем только два наиболее распространенных из них: JPEG и GIF.

```
enum class Suffix {
    enum Encoding { none, jpg, gif };
};
```

В нашей библиотеке графического интерфейса изображение в памяти представляется в виде объекта класса **Image**.

```
struct Image : Shape {
    Image(Point xy, string file_name, Suffix e = Suffix::none);
    ~Image() { delete p; }
    void draw_lines() const;
    void set_mask(Point xy, int ww, int hh)
        { w=ww; h=hh; cx=xy.x; cy=xy.y; }
private:
    int w,h; // Определяем "маскировочное окно" в изображении
```

```
            // относительно позиции (cx,cy)
    int cx,cy;
    Fl_Image* p;
    Text fn;
};
```

Конструктор класса **Image** пытается открыть файл с переданным ему именем и создать рисунок, используя кодировку, указанную в дополнительном аргументе или (как правило) в расширении файла. Если изображение невозможно вывести на экран (например, потому, что файл не найден), класс **Image** выводит на экран объект **Bad_image**. Определение класса **Bad_image** выглядит следующим образом:

```
struct Bad_image : Fl_Image {
    Bad_image(int h, int w) : Fl_Image(h,w,0) {}
    void draw(int x, int y, int,
              int, int, int) { draw_empty(x,y); }
};
```

Работа с изображениями в графической библиотеке довольно сложна, но основная сложность класса **Image** кроется в обработке файла в конструкторе.

```
// Более сложный конструктор, потому что ошибки,
// связанные с графическими файлами, трудно найти
Image::Image(Point xy, string s, Suffix e)
    :w(0), h(0), fn(xy,"")
{
    add(xy);
    if (!can_open(s)) { // Можно ли открыть файл s?
        fn.set_label("Невозможно открыть \""+s+'"');
        p = new Bad_image(30,20); // "Ошибочное изображение"
        return;
    }

    if (e == Suffix::none) e = get_encoding(s);

    switch(e) {             // Проверка кодировки
    case Suffix::jpg:
        p = new Fl_JPEG_Image(s.c_str());
        break;
    case Suffix::gif:
        p = new Fl_GIF_Image(s.c_str());
        break;
    default:                // Неподдерживаемая кодировка
        fn.set_label("Неподдерживаемый тип файла \""+s+'"');
        p = new Bad_image(30,20); // "Ошибочное изображение"
    }
}
```

Расширение файла используется для того, чтобы определить вид объекта, создаваемого для хранения изображения (**Fl_JPEG_Image** или **Fl_GIF_Image**). Этот объект реализации создается с помощью оператора **new** и присваивается указателю. Эти подробности реализации (оператор **new** и указатели рассматриваются в главе 17) связаны с организацией библиотеки FLTK и не имеют для нас большого значения. FLTK использует строки в стиле C, поэтому мы используем **s.c_str()** вместо непосредственного применения строки **s**.

Теперь настало время реализовать функцию **can_open()**, проверяющую, можно ли открыть файл для чтения.

```
bool can_open(const string& s)
    // Проверка, существует ли файл s и
    // можно ли его открыть для чтения
{
    ifstream ff(s.c_str());
    return ff;
}
```

Открыть файл, а затем закрыть его — довольно примитивный способ проверки, позволяющий переносимо отделить ошибки, связанные с невозможностью открыть файл, от ошибок, обусловленных неверным форматированием данных.

Если хотите, можете посмотреть на определение функции **get_encoding()**: она просто анализирует суффикс и ищет соответствие в таблице заранее заданных суффиксов. Эта таблица реализована с помощью стандартного типа **map** (раздел 21.6).

Задание

1. Создайте простое окно **Simple_window** размером 800×1000 пикселей.
2. Разместите сетку размером 8×8 в левой части окна размером 800×800 пикселей (так что каждый квадрат сетки имеет размер 100×100 пикселей).
3. Раскрасьте восемь квадратов, расположенных по диагонали, начиная с левого верхнего угла, в красный цвет (используйте класс **Rectangle**).
4. Подберите изображение размером 200×200 пикселей (в формате JPEG или GIF) и разместите три его копии поверх сетки (каждое изображение покроет четыре квадрата). Если вы не найдете изображения, размеры которого точно равнялись бы 200 пикселям, то, используя функцию **set_mask()**, вырежьте соответствующий фрагмент более крупного изображения. Не закрывайте красные квадраты.

5. Добавьте изображение размером 100×100 пикселей. Перемещайте его с одного квадрата на другой, щелкая на кнопке Next. Для этого поместите вызов функции `wait_for_button()` в цикл, сопроводив его командами, выбирающими новый квадрат для вашего изображения.

Контрольные вопросы

1. Почему мы просто не используем какую-нибудь коммерческую или бесплатную графическую библиотеку?
2. Сколько примерно классов из библиотеки графического интерфейса нам понадобится, чтобы организовать простой вывод графической информации?
3. Какие заголовочные файлы нужны для использования библиотеки графического интерфейса?
4. Какие классы определяют замкнутые фигуры?
5. Почему мы просто не используем класс `Line` для рисования любой фигуры?
6. Что означают аргументы конструктора класса `Point`?
7. Перечислите компоненты класса `Line_style`.
8. Перечислите компоненты класса `Color`.
9. Что такое система RGB?
10. В чем заключается разница между двумя объектами класса `Line` и объектом `Lines`, содержащим две линии?
11. Какие свойства можно задать для любого объекта класса `Shape`?
12. Сколько ребер объекта класса `Closed_polyline` определяются пятью объектами класса `Point`?
13. Что мы увидим на экране, если определим объект класса `Shape`, но не свяжем его с объектом класса `Window`?
14. Чем объект класса `Rectangle` отличается от объекта класса `Polygon` с четырьмя объектами класса `Point` (углами)?
15. Чем объект класса `Polygon` отличается от объекта класса `Closed_polyline`?
16. Что расположено сверху: заполненная цветом область или границы фигуры?
17. Почему мы не определили класс `Triangle` (ведь мы определили класс `Rectangle`)?
18. Как переместить объект класса `Shape` в другое место окна?
19. Как пометить объект класса `Shape` строкой текста?

20. Какие свойства текстовой строки можно задать в классе `Text`?
21. Что такое шрифт и зачем он нужен?
22. Для чего нужен класс `Vector_ref` и как его использовать?
23. В чем заключается разница между классами `Circle` и `Ellipse`?
24. Что произойдет, если мы попытаемся изобразить объект класса `Image` с некоторым именем файла, которое не относится к файлу, содержащему изображение?
25. Как вывести на экран часть изображения?

Термины

`GIF`	кодировка изображения	отрезок
`JPEG`	линия	размер шрифта
`Vector_ref`	ломаная линия	стиль линии
видимый	многоугольник	точка
замкнутая фигура	невидимый	цвет
заполнение	неименованный объект	шрифт
изображение	открытая фигура	эллипс

Упражнения

Для каждого упражнения, в котором требуется определить класс, выведите на экран несколько объектов данного класса и продемонстрируйте, как они работают.

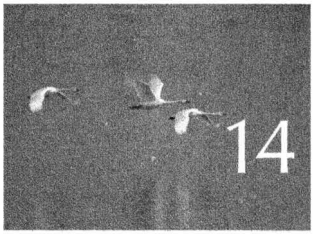
1. Определите класс `Arc`, рисующий часть эллипса. *Указание*: `fl_arc()`.
2. Нарисуйте окно с закругленными углами. Определите класс `Box`, состоящий из четырех линий и четырех дуг.
3. Определите класс `Arrow`, рисующий стрелки.
4. Определите функции `n()`, `s()`, `e()`, `w()`, `center()`, `ne()`, `se()`, `sw()` и `nw()`. Каждая из них должна получать аргумент типа `Rectangle` и возвращать объект типа `Point`. Эти функции должны определять "точки подключения", расположенные на границах и внутри прямоугольника. Например, `nw(r)` — это левый верхний (северо-западный) угол объекта класса `Rectangle` с именем `r`.
5. Определите функции из упр. 4 для классов `Circle` и `Ellipse`. Разместите точки подключения внутри или вне фигур, но не за пределами описанного вокруг них прямоугольника.
6. Напишите программу, рисующую диаграмму классов, подобную изображенной в разделе 12.6. Программировать будет проще, если начать

с определения класса **Box**, объект которого представляет собой прямоугольник с текстовой меткой.

7. Создайте цветную диаграмму RGB (поищите пример в вебе).
8. Определите класс **Regular_hexagon** (правильный шестиугольник). В качестве аргументов конструктора используйте центр и расстояние от центра до угловой точки.
9. Покройте часть окна мозаикой из объектов класса **Regular_hexagon** (используйте не меньше восьми шестиугольников).
10. Определите класс **Regular_polygon**. В качестве аргументов конструктора используйте центр, количество сторон (не меньше трех) и расстояние от центра до угла.
11. Нарисуйте эллипс размером 300×200 пикселей. Нарисуйте ось *x* длиной 400 пикселей и ось *y* длиной 300 пикселей, проходящие через центр эллипса. Пометьте фокусы. Отметьте на эллипсе точку, которая не принадлежит ни одной из осей. Соедините эту точку с фокусами двумя линиями.
12. Нарисуйте окружность. Заставьте метку перемещаться по окружности (пусть она немного сдвигается всякий раз, когда вы щелкаете на кнопке Next).
13. Нарисуйте матрицу цветов из раздела 13.10, но без линий, окаймляющих каждый квадрат.
14. Определите класс прямоугольного треугольника. Составьте восьмиугольник из восьми прямоугольных треугольников разного цвета.
15. Покройте окно мозаикой из маленьких прямоугольных треугольников.
16. Выполните предыдущее упражнение с шестиугольниками.
17. Выполните предыдущее упражнение с шестиугольниками разных цветов.
18. Определите класс **Poly**, представляющий многоугольник, с проверкой в конструкторе того, что его точки образуют многоугольник. *Указание*: вы должны передавать в конструктор координаты точек.
19. Определите класс **Star**. Одним из его параметров должно быть количество точек. Нарисуйте несколько звездочек с разным количеством точек, разноцветными линиями и разными цветами заполнения.

Послесловие

В главе 12 мы показали, как использовать классы. В этой главе мы перешли на уровень вверх по "пищевой цепочке" программистов: теперь в дополнение к роли пользователей классов мы стали и их разработчиками.



14

Проектирование графических классов

Польза, прочность, красота.
— Витрувий (Vitruvius)

Главы, посвященные графике, преследуют двоякую цель: описать полезные инструменты, предназначенные для отображения информации, и одновременно использовать семейство графических классов для иллюстрации общих методов проектирования и реализации программ. В частности, данная глава посвящена некоторым методам проектирования интерфейса и понятию наследования. По ходу изложения мы вынуждены сделать небольшое отступление, посвященное возможностям языка, непосредственно поддерживающим объектно-ориентированное программирование: механизму наследования классов, виртуальным функциям и управлению доступом. Мы считаем, что проектирование классов невозможно обсуждать отдельно от их использования и реализации, поэтому наше обсуждение вопросов проектирования носит весьма конкретный характер. Возможно, было бы лучше назвать эту главу "Проектирование и реализация графических классов".

14.1. Принципы проектирования

Каковы принципы проектирования наших классов графического интерфейса? Сначала надо разобраться в смысле поставленного вопроса. Что такое "принципы проектирования" и почему мы должны говорить о них, вместо того чтобы заняться созданием изящных рисунков?

14.1.1. Типы

Графика — это пример предметной области, и совокупность основных понятий и возможностей программист должен искать именно в ней. Если понятия предметной области представлены в программе нечетко, противоречиво, неполно или просто плохо, то сложность разработки средств графического вывода возрастает. Мы хотим, чтобы наши графические классы упростили работу пользователей.

Идеал программного проектирования — непосредственное отражение понятий предметной области в исходном тексте. Если вы хорошо разбираетесь в предметной области, то легко поймете код, и наоборот. Рассмотрим пример.

- `Window` — окно, открываемое операционной системой.
- `Line` — линия, которую вы видите на экране.
- `Point` — точка в системе координат.
- `Color` — цвет объекта на экране.
- `Shape` — общие свойства всех фигур в нашей модели графики или графического пользовательского интерфейса.

Последнее понятие, `Shape`, отличается от остальных тем, что является обобщенным, т.е. чисто абстрактным. Абстрактную фигуру изобразить невозможно; мы всегда видим на экране конкретную фигуру, будь то линия

или шестиугольник. Это отражается и в определении наших типов: попытка создать объект класса **Shape** будет пресечена компилятором.

Совокупность классов нашего графического интерфейса образует библиотеку; эти классы предназначены для совместного использования в сочетании друг с другом. Они должны послужить образцом при создании других графических фигур и строительных блоков для других классов. Поскольку все классы связаны между собой, мы не можем принимать проектные решения для каждого класса по отдельности. В совокупности наши классы отражают наше представление о том, как работать с графикой. Мы должны гарантировать, что наша точка зрения является достаточно элегантной и логичной. Поскольку размер библиотеки ограничен, а область графических приложений бесконечна, рассчитывать на полноту библиотеки не приходится. Следовательно, мы должны сосредоточиться на простоте и гибкости библиотеки.

На самом деле ни одна библиотека не способна моделировать все аспекты предметной области. Это не только невозможно, но и бессмысленно. Представьте себе библиотеку для отображения географической информации. Хотите ли вы демонстрировать растительность, национальные, государственные или другие политические границы, автомобильные и железные дороги или реки? Надо ли показывать социальные и экономические данные? Отражать ли сезонные колебания температуры и влажности? Показывать ли розу ветров? Следует ли изобразить авиамаршруты? Стоит ли отметить местоположение школ, ресторанов быстрого питания или местных косметических салонов? "Показать все!" Для исчерпывающей географической системы это могло бы быть хорошим ответом, но в нашем распоряжении только один дисплей. Так можно было бы поступить при разработке библиотеки, поддерживающей работу соответствующих географических систем, но вряд ли такая библиотека могла бы одновременно обеспечивать возможность рисовать элементы карт от руки, редактировать фотографии, строить научные диаграммы и отображать элементы управления самолетами.

Итак, как всегда, мы должны решить, что для нас важно. В данном случае мы должны выбрать вид графики и графического пользовательского интерфейса. Попытка сделать сразу все обречена на провал. Хорошая библиотека непосредственно и точно моделирует предметную область с конкретной точки зрения, делая акцент на некоторых аспектах приложения и затеняя остальные.

Классы, которые мы опишем, разработаны для создания простых графических приложений и пользовательских интерфейсов. В основном они предназначены для пользователей, которым необходимо представить в графической форме данные и результаты вычислительных, научных или технических приложений. Используя наши классы, вы сможете создавать собственные. Если этого окажется недостаточно, мы продемонстрируем

множество деталей библиотеки FLTK, которые подскажут вам, как использовать ее (или другую подобную библиотеку) в своих целях. Однако, если вы решите идти этим путем, не спешите и сначала усвойте материал, изложенный в главах 17 и 18. Эти главы содержат информацию об указателях и управлении памятью, которая совершенно необходима для непосредственного использования большинства графических библиотек.

Одним из наших ключевых решений было создание множества небольших классов, содержащих всего по нескольку операций. Например, мы создали классы **Open_polyline**, **Closed_polyline**, **Polygon**, **Rectangle**, **Marked_polyline**, **Marks** и **Mark** вместо одного отдельного класса (который можно было бы назвать "**Polyline**"); в таком классе было бы предусмотрено множество аргументов и операций, позволяющих задавать вид ломаной и даже изменять ее, преобразуя из одной разновидности в другую. Доводя эту идею до абсурда, можно было бы рассматривать фигуру каждого вида как часть единого класса **Shape**. Но мы считаем, что наиболее точно и удобно нашу область графических приложений моделирует использование небольших классов. Отдельный класс, обеспечивающий "все", завалил бы пользователя данными и возможностями, затруднив понимание, усложнив отладку и снизив производительность.

14.1.2. Операции

В каждом классе мы предусмотрели минимум операций. Наш идеал — минимальный интерфейс, позволяющий делать то, что мы хотим. Если нам потребуются дополнительные возможности, мы всегда сможем добавить функции, не являющиеся членами класса, или определить еще один класс.

Мы стремимся к тому, чтобы интерфейсы наших классов имели общий стиль. Например, все функции, выполняющие аналогичные операции в разных классах, называются одинаково, получают аргументы одинаковых типов, и там, где это возможно, их аргументы следуют в одинаковом порядке. Рассмотрим конструкторы: если фигура размещается на экране, в качестве первого аргумента она принимает объект типа **Point**.

```
Line ln(Point(100,200),Point(300,400));
Mark m(Point(100,200),'x'); // Отображает отдельную точку в виде буквы "x"
Circle c(Point(200,200),250);
```

Все функции, работающие с точками, используют класс **Point**. Это очевидно, но многие библиотеки смешивают стили. Например, представим себе функцию, рисующую линию. Мы можем использовать два стиля.

```
void draw_line(Point p1, Point p2);     // От p1 до p2 (наш стиль)
void draw_line(int x1, int y1, int x2, int y2); // От (x1,y1) до (x2,y2)
```

Можно было бы допустить оба стиля, но для обеспечения согласованности, улучшения проверки типов и повышения удобочитаемости будем пользоваться исключительно первым. Последовательное использование класса `Point` позволит также избежать путаницы между парами координат и другими парами целых чисел: шириной и высотой. Рассмотрим пример.

```
draw_rectangle(Point{100,200}, 300, 400); // Наш стиль
draw_rectangle (100,200,300,400);         // Альтернативный
```

При первом вызове функция рисует прямоугольник по заданной точке, ширине и высоте. Это легко угадать. А что можно сказать о втором вызове? Имеется в виду прямоугольник, определенный точками (100,200) и (300,400)? Или прямоугольник, определенный точкой (100,200), шириной 300 и высотой 400? А может быть, программист имел в виду нечто совершенно другое (хотя и разумное)? Последовательно используя класс `Point`, мы можем избежать таких недоразумений.

Иногда, когда функция требует ширину и высоту, они передаются ей именно в таком порядке (как, например, координату *x* всегда указывают до координаты *y*). Последовательное соблюдение таких мелочей удивительно облегчает работу с программой и позволяет избежать ошибок во время ее выполнения.

Логически идентичные операции называются одинаково. Например, каждая функция, которая добавляет точки, линии и так далее к любой фигуре, называется `add()`, а любая функция, рисующая линии, называется `draw_lines()`. Такое единообразие позволяет нам помнить (или вспомнить по некоторым признакам), что делает функция, и облегчает разработку новых классов (по аналогии). Иногда это позволяет даже написать код, работающий со многими разными типами, поскольку операции над этими типами имеют общий шаблон. Такой код называют *обобщенным* (generic); подробно мы рассмотрим его в главах 19–21.

14.1.3. Именование

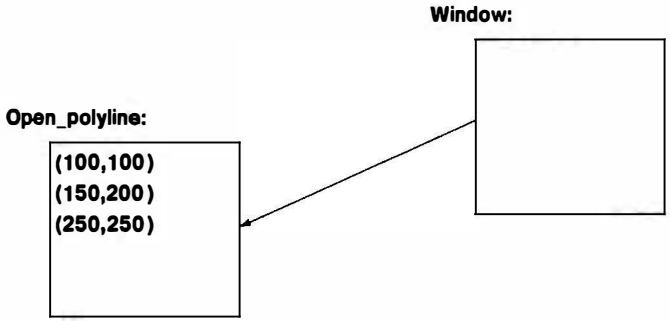
Логически разные операции имеют разные имена. И вновь, несмотря на очевидность, зададимся вопросом "Почему мы «связываем» (attach) объект класса `Shape` с объектом класса `Window`, но «добавляем» (add) объект класса `Line` к объекту класса `Shape`?" В обоих случаях мы "помещаем нечто во что-то", так почему бы не назвать такие операции одинаково? Но нет. За этой схожестью кроется фундаментальная разница. Рассмотрим пример.

```
Open_polyline opl;
opl.add(Point{100,100});
opl.add(Point{150,200});
opl.add(Point{250,250});
```

Здесь мы копируем три точки в объект `opl`. Фигуре `opl` безразлично, что будет с нашими точками после вызова функции `add()`; она хранит собственные копии этих точек. На самом деле мы редко храним копии точек, просто передавая их фигуре. С другой стороны, посмотрим на следующую инструкцию:

```
win.attach(opl);
```

Здесь мы создаем связь между окном `win` и нашей фигурой `opl`; объект `win` не создает копию объекта `opl`, а хранит ссылку на него. Так что мы обязаны обеспечить корректность объекта `opl`, поскольку объект `win` использует его. Таким образом, пока окно `win` использует фигуру `opl`, последняя не должна выйти из области видимости. Мы можем обновить объект `opl`, и в следующий раз при выводе в окне `win` фигуры `opl` изменения отобразятся на экране. Проиллюстрировать разницу между функциями `attach()` и `add()` графически можно следующим образом.



Функция `add()` использует механизм передачи параметров по значению (копии), а функция `attach()` — механизм передачи параметров по ссылке (использует общий объект). Мы могли бы принять решение о копировании графических объектов в объекты класса `Window`, но это была бы совсем другая программная модель, которая определялась бы выбором функции `add()`, а не `attach()`. Мы же решили просто связывать графический объект с объектом класса `Window`. Это решение имеет важные последствия. Например, мы не можем создать объект, связать его, позволить его уничтожить и ожидать, что программа продолжит работать.

```
void f(Simple_window& w)
{
    Rectangle r{Point{100,200},50,30};
    w.attach(r);
}   // Объекта r больше нет!

int main()
{
    Simple_window win{Point{100,100},600,400,"Мое окно"};
```

```
    // ...
    f(win); // Возникают проблемы!
    // ...
    win.wait_for_button();
}
```

Пока мы выходили из функции `f()` и входили в функцию `wait_for_button()`, объект `r` для объекта `win` перестал существовать, и `win` не к чему обращаться и нечего выводить на экран. В главе 17 мы покажем, как создавать в функции объект, который будет сохраняться между ее вызовами, а пока мы должны избегать связывания с объектом, который может "не дотянуть" до вызова функции `wait_for_button()`. Помочь в этом вопросе нам может класс `Vector_ref`, который рассматривается в разделах 13.10 и Д.4.

Обратите внимание, что если бы мы объявили функцию `f()` так, чтобы она получала константную ссылку на объект класса `Window` (как было рекомендовано в разделе 8.5.6), то компилятор предотвратил бы нашу ошибку: мы не можем выполнить вызов `attach(r)` с аргументом типа `const Window`, поскольку функция `attach()` должна внести изменения в объект класса `Window`, чтобы зарегистрировать связь между ним и объектом `r`.

14.1.4. Изменяемость

Основные вопросы, на которые следует ответить, проектируя классы, звучат так: "Кто может модифицировать данные?" и "Как он может это делать?" Мы пытаемся гарантировать, что изменение состояния объекта будет осуществляться только его собственным классом. Ключевую роль в этом играют модификаторы `public` и `private`, но мы продемонстрируем еще более гибкий и тонкий механизм, основанный на модификаторе `protected`. Это означает, что мы не можем просто включить в класс какой-то член, скажем, переменную `label` типа `string`; мы должны также решить, следует ли позволить изменять его после создания объекта, и если следует, то каким образом. Мы должны также решить, может ли код, отличный от кода данного класса, иметь доступ к переменной `label`, и если может, то как. Рассмотрим пример.

```
struct Circle {
    // ...
private:
    int r; // Радиус
};

Circle c{Point{100,200},50};
c.r = -9; // OK? Нет — ошибка компиляции:
          // переменная Circle::r закрыта
```

Как указано в главе 13, мы решили предотвратить прямой доступ к большинству данных-членов класса. Это дает нам возможность проверять "глупые" значения, например отрицательные радиусы у объектов класса **Circle**. Для простоты реализации мы пользуемся этой возможностью весьма ограниченно, поэтому будьте осторожны с передаваемыми значениями. Мы отказались от полной и последовательной проверки, желая уменьшить объем кода и понимая, что если пользователь введет "глупое" значение, то ранее введенные данные от этого не пострадают, просто на экране появится искаженное изображение.

Мы интерпретируем экран (т.е. совокупность объектов класса **Window**) исключительно как устройство вывода. Мы можем выводить новые объекты и удалять старые, но никогда не обращаемся к системе за информацией, которую сами не можем извлечь из структур данных, на основе которых строится изображение.

14.2. Класс Shape

Класс **Shape** отражает общее понятие о том, что может изображаться в объекте класса **Window** на экране.

- Этот класс представляет собой понятие, которое связывает графические объекты с нашей абстракцией **Window**, которая, в свою очередь, обеспечивает связь с операционной системой и физическим экраном.
- Это класс, работающий с цветом и стилем, используемыми при рисовании линий. Для этого он хранит члены классов **Line_style** и **Color** (как для линий, так и для заполнения).
- Этот класс может хранить последовательности объектов класса **Point** и информацию о том, как их рисовать.

Опытные проектировщики отметят, что класс, обладающий этими тремя свойствами, вероятно, имеет недостаточно общий характер. Однако нам нужно решение, для которого простота важнее обобщенности.

Сначала мы представим полный класс, а затем подробно его обсудим.

```
class Shape { // Работает с цветом и стилем, хранит последовательность точек
public:
    void draw() const;                  // Вывод цветных линий
    virtual void move(int dx, int dy); // Перемещение фигуры
                                        // +=dx и +=dy

    void set_color(Color col);
    Color color() const;

    void set_style(Line_style sty);
    Line_style style() const;
```

```
    void set_fill_color(Color col);
    Color fill_color() const;

    Point point(int i) const;          // Доступ только
    int number_of_points() const;      // для чтения

    Shape(const Shape&) = delete;      // Предотвращение
    Shape& operator=(const Shape&)     // копирования
                    = delete;

    virtual ~Shape() { }
protected:
    Shape();
    Shape(initializer_list<Point> lst);// Добавление точек
    virtual void draw_lines() const;   // Вывод линий
    void add(Point p);                 // Добавление p к точкам
    void set_point(int i, Point p);    // points[i]=p;
private:
    vector<Point> points;              // Не используется всеми
                                       // фигурами
    Color lcolor{fl_color()};          // Цвет линий и символов
                                       // (значение по умолчанию)
    Line_style ls{0};
    Color fcolor{Color::invisible};    // Цвет заполнения

};
```

Это относительно сложный класс, разработанный для поддержки множества графических классов и представления общей концепции фигуры на экране. Однако в нем всего четыре данных-члена и пятнадцать функций. Более того, почти все эти функции тривиальны, так что мы можем сосредоточиться на вопросах проектирования. В оставшейся части главы мы пройдемся по всем членам шаг за шагом и объясним их роль в проекте класса.

14.2.1. Абстрактный класс

Сначала рассмотрим конструкторы класса **Shape**:

```
protected:
    Shape();
    Shape(initializer_list<Point> lst);// Добавление точек
```

Эти конструкторы объявлены с модификатором **protected**. Это значит, что их можно использовать непосредственно только в классах, производных от класса **Shape** (используя запись **:Shape**). Иначе говоря, класс **Shape** можно использовать только в качестве базового для других классов, таких как **Line** и **Open_polyline**. Цель модификатора **protected:** — гарантировать, что мы не сможем создать объекты класса **Shape** непосредственно:

```
Shape ss; // Ошибка: невозможно создать объект класса Shape
```

Класс **Shape** спроектирован таким образом, чтобы быть использованным только в роли базового класса. В данном случае ничего страшного не произошло бы, если бы мы позволили создавать объекты класса **Shape** непосредственно, но, ограничив его применение, мы оставили бы возможность модификаций, которые могли бы сделать его неподходящим для непосредственного применения. Кроме того, запретив непосредственное создание объектов класса **Shape**, мы моделируем идею о том, что абстрактной фигуры в природе не существует, а реальными являются лишь конкретные фигуры, такие как объекты класса **Circle** и **Closed_polyline**. Подумайте об этом! Как выглядит абстрактная фигура? Единственный разумный ответ на такой вопрос — встречный вопрос: "Какая фигура?" Понятие о фигуре, воплощенное в классе **Shape**, носит абстрактный характер. Это важное и часто полезное свойство проекта, поэтому мы не хотим никакого компромисса в этом отношении. Позволить пользователям непосредственно создавать объекты класса **Shape** противоречило бы нашему идеалу класса как непосредственного воплощения концепции.

Конструкторы по умолчанию устанавливают значения членов равными их значениям по умолчанию. Здесь снова в качестве базовой использована библиотека FLTK. Однако понятия цвета и стиля, принятые в библиотеке FLTK, непосредственно при использовании не упоминаются. Они являются лишь частью реализации классов **Shape**, **Color** и **Line_style**. Вектор **vector<Points>** по умолчанию является пустым вектором.

Конструктор со списком инициализации также использует инициализаторы по умолчанию, после чего добавляет с помощью функции **add()** элементы своего списка аргументов в **Shape**:

```
Shape::Shape(initializer_list<Point> lst)
{
    for (Point p : lst) add(p);
}
```

Класс является *абстрактным* (abstract), если его можно использовать только в качестве базового класса. Другим — более распространенным — способом сделать класс абстрактным является добавление в него *чисто виртуальной функции* (pure virtual function), которую мы рассмотрим в разделе 14.3.5. Класс, который можно использовать для создания объектов, в противоположность абстрактному классу называется *конкретным* (concrete). Обратите внимание на то, что слова *абстрактный* и *конкретный* часто используются и в повседневной жизни. Представим себе, что мы идем в магазин покупать фотоаппарат. Однако мы не можем просто попросить какой-то фотоаппарат и принести его домой. Какую торговую марку вы предпочитаете? Какую модель фотоаппарата хотите купить? Слово *фотоаппарат* — это обобщение; оно ссылается на абстрактное понятие. Название "Canon EOS 600D" означает конкрет-

ную разновидность фотоаппарата, конкретный экземпляр которого с уникальным серийным номером мы можем получить (в обмен на большую сумму денег). Итак, фотоаппарат — это абстрактный (базовый) класс, "Canon EOS 600D" — конкретный (производный) класс, а реальный фотоаппарат в моей руке (если я его купил) — это объект.

Объявление

```
virtual ~Shape() {}
```

определяет виртуальный деструктор. Мы не будем пока его использовать и рассмотрим позднее, в разделе 17.5.2.

14.2.2. Управление доступом

Класс **Shape** объявляет все данные-члены закрытыми.

```
private:
    vector<Point> points;           // не используется всеми
                                    // фигурами
    Color lcolor{fl_color()};       // Цвет линий и символов
                                    // (значение по умолчанию)
    Line_style ls{0};
    Color fcolor{Color::invisible}; // Цвет заполнения
```

Инициализаторы данных-членов не зависят от аргументов конструкторов, так что я указал их в объявлениях этих членов. Как всегда, значением вектора по умолчанию является пустой вектор, так что для него я ничего явно не указываю. Конструктор будет использовать эти значения по умолчанию.

Поскольку данные-члены класса **Shape** объявлены закрытыми, нам нужно предусмотреть для них функции доступа. Существует несколько возможных стилей решения этой задачи. Мы выбрали простой, удобный и понятный: если у нас есть член, представляющий свойство **X**, то предоставляется пара функций **X()** и **set_X()** для чтения и записи этого свойства соответственно, например:

```
void Shape::set_color(Color col)
{
    lcolor = col;
}

Color Shape::color() const
{
    return lcolor;
}
```

Основной недостаток этого стиля заключается в том, что мы не можем назвать переменную так же, как функцию для ее чтения. Как всегда, мы предпочли выбрать наиболее удобные имена для функций, поскольку они

являются частью открытого интерфейса. Как назвать закрытые переменные — существенно менее важно. Обратите внимание на то, что мы использовали ключевое слово **const**, чтобы подчеркнуть, что функция чтения не может модифицировать члены своего класса **Shape** (см. раздел 9.7.4).

В классе **Shape** хранится вектор **points** объектов класса **Point**, который предназначен для поддержки производных классов. Для добавления элементов типа **Point** в вектор **points** предусмотрена функция **add()**:

```
void Shape::add(Point p) // Защищенная
{
    points.push_back(p);
}
```

Естественно, изначально вектор **points** пуст. Мы решили снабдить класс **Shape** полным функциональным интерфейсом, не позволяя пользователям — даже функциям-членам классов, производных от класса **Shape** — непосредственно обращаться к его данным-членам. Одним людям решение о создании функционального интерфейса представляется тривиальным, поскольку они чувствуют, что недопустимо делать какие-либо данные-члены класса открытыми. Другим наш подход кажется слишком ограниченным, поскольку мы не разрешаем прямой доступ к членам базового класса даже членам производных классов.

Классы, производные от класса **Shape**, например **Circle** и **Polygon**, "понимают", что означают их точки. Базовый класс **Shape** этого "не понимает", он просто хранит эти точки. Следовательно, производные классы должны иметь контроль над тем, как добавляются точки. Рассмотрим пример.

- Классы **Circle** и **Rectangle** не позволяют пользователю добавлять точки, это просто не имеет смысла. Что такое прямоугольник с дополнительной точкой? (См. раздел 12.7.6.)
- Класс **Lines** позволяет добавлять только пары точек (но не отдельные точки; см. раздел 13.3).
- Классы **Open_polyline** и **Marks** позволяют добавлять любое количество точек.
- Класс **Polygon** позволяет добавлять точки только с помощью функции **add()**, проверяющей пересечения (раздел 13.8).

Мы сделали функцию **add()** защищенной с помощью модификатора **protected** (т.е. сделали ее доступной только для производных классов), чтобы гарантировать, что производные классы смогут управлять добавлением точек. Если бы функция **add()** была объявлена как **public** (т.е. добавлять точки мог бы кто угодно) или как **private** (т.е. добавлять точки мог бы только сам класс **Shape**), то точное соответствие функциональных возможностей нашему представлению о фигуре стало бы невозможным.

По аналогичным причинам мы сделали защищенной и функцию **set_point()**. В общем случае только производный класс может "знать", что означают точки и можно ли их изменять, не нарушая инвариант. Например, если у нас есть класс **Regular_hexagon**, объявленный как множество из шести точек, то изменение даже одной точки может породить фигуру, не являющуюся правильным шестиугольником. С другой стороны, если мы изменим одну из точек прямоугольника, то в результате все равно получим прямоугольник. Фактически функция **set_point()** в этом случае оказывается ненужной, поэтому мы включили ее просто для того, чтобы обеспечить выполнение правил чтения и записи каждого атрибута класса **Shape**. Например, если бы мы захотели создать класс **Mutable_rectangle**, то могли бы породить его из класса **Rectangle** и снабдить операциями, изменяющими точки.

Мы объявили вектор **points** как **private**, чтобы защитить его от нежелательных изменений. Для того чтобы он был полезен, мы должны обеспечить к нему доступ.

```
void Shape::set_point(int i, Point p) // Не используется
{
    points[i] = p;
}

Point Shape::point(int i) const
{
    return points[i];
}

int Shape::number_of_points() const
{
    return points.size();
}
```

В функциях-членах производного класса эти функции используются следующим образом:

```
void Lines::draw_lines() const
    // Рисует линии, соединяющие пары точек
{
    for (int i=1; i<number_of_points(); i+=2)
    fl_line(point(i-1).x,point(i-1).y,point(i).x,point(i).y);
}
```

Все эти тривиальные функции доступа могут вызвать у вас обеспокоенность. Эффективны ли они? Не замедляют ли работу программы? Не увеличивают ли они размер генерируемого кода? Нет, так как компилятор всех их делает встраиваемыми (inline). Вызов функции **number_of_points()** занимает столько же байтов памяти и выполняет столько же команд, сколько и непосредственный вызов функции **points.size()**.

Решения, касающиеся управления доступом, очень важны. Мы могли бы создать близкую к минимальной версию класса **Shape**:

```
struct Shape {                  // Близкая к минимальной версия
                                // (слишком простая, не используется)
    Shape();
    Shape(initializer_list<Point>);
    void draw() const;                      // Вывод цветных линий
    virtual void draw_lines() const;        // Вывод линий
    virtual void move(int dx, int dy); // Перемещение фигуры
    virtual ~Shape();

    vector<Point> points; // Всеми фигурами не используется
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

Какие возможности обеспечивают эти двенадцать дополнительных функций-членов и две строки спецификаций доступа (**private:** и **protected:**)? Главный ответ состоит в том, что защита класса от нежелательного изменения позволяет разработчику создавать лучшие классы с меньшими усилиями. Этот же аргумент относится и к инвариантам (см. раздел 9.4.3). Подчеркнем эти преимущества на примере определения классов, производных от класса **Shape**. В более ранних вариантах класса **Shape** мы использовали следующие переменные:

```
Fl_Color lcolor;
int line_style;
```

Оказывается, это очень ограничивает наши возможности (стиль линии, задаваемый переменной типа **int**, не позволяет элегантно поддерживать ширину линии, а класс **Fl_Color** не предусматривает невидимые линии) и приводит к довольно запутанному коду. Если бы эти две переменные были открытыми и использовались в пользовательской программе, то мы могли бы улучшить интерфейсную библиотеку только ценой неработоспособности уже имеющегося кода (поскольку в нем упоминаются имена **lcolor** и **line_style**).

Кроме того, функции доступа часто обеспечивают удобство обозначений. Например, инструкция **s.add(p)** читается и записывается легче, чем **s.points.push_back(p)**.

14.2.3. Рисование фигур

Мы описали почти все, кроме ядра класса **Shape**.

```
void draw() const;                  // Работает с цветом и вызывает
                                    //  функцию draw_lines
virtual void draw_lines() const; // Рисует линии
```

Основная задача класса **Shape** — рисовать фигуры. Мы можем удалить из класса **Shape** все остальные функции и оставить его вообще без собственных данных, не нанеся особого вреда нашей основной концепции (см. раздел 14.4); рисование — это главная задача класса **Shape**. Он выполняет ее с помощью библиотеки FLTK и операционной системы, но с точки зрения пользователя он предоставляет только две функции.

- Функция **draw()** применяет стиль и цвет, а затем вызывает функцию **draw_lines()**.
- Функция **draw_lines()** помещает пиксели на экран.

Функция **draw()** не использует никаких новых методов. Она просто вызывает функции библиотеки FLTK, чтобы задать цвет и стиль фигуры, вызывает функцию **draw_lines()**, чтобы выполнить реальное рисование на экране, а затем пытается восстановить цвет и фигуру, заданные до ее вызова.

```
void Shape::draw() const
{
    Fl_Color oldc = fl_color();
    // Универсального переносимого способа получить
    // текущий стиль не существует
    fl_color(lcolor.as_int());             // Задаем цвет
    fl_line_style(ls.style(),ls.width()); // Задаем стиль
    draw_lines();
    fl_color(oldc);       // Восстанавливаем цвет
    fl_line_style(0);     // Восстанавливаем стиль линии по умолчанию
}
```

К сожалению, в библиотеке FLTK не предусмотрен способ идентификации текущего стиля, поэтому он просто сбрасывается к значениям по умолчанию. Это пример компромисса, на который мы иногда идем, чтобы обеспечить простоту и мобильность программы. Мы не думаем, что эту функциональную возможность имеет смысл реализовать в нашей интерфейсной библиотеке.

Обратите внимание на то, что функция **Shape::draw()** не работает с цветом заливки фигуры и не управляет видимостью линий. Эти свойства обрабатывают отдельные функции **draw_lines()**, которые лучше знают, как их интерпретировать. В принципе, всю обработку цвета и стиля можно было бы делегировать отдельным функциям **draw_lines()**, но для этого пришлось бы повторять много одних и тех же фрагментов кода.

Рассмотрим теперь, как организовать работу с функцией **draw_lines()**. Если немного подумать, то можно прийти к выводу, что функции-члену класса **Shape** было бы трудно рисовать все, что необходимо для создания любой разновидности фигуры. Для этого пришлось бы хранить в объекте класса **Shape** каждый пиксель каждой фигуры. Если мы используем модель вектора **vector<Point>**, то будем вынуждены хра-

нить огромное количество точек. И что еще хуже, экран (т.е. устройство для вывода графических изображений) уже делает это, причем делает это на аппаратном уровне — лучше нас.

Для того чтобы избежать лишней работы и хранения лишней информации, примем другой подход: дадим каждому классу, производному от класса **Shape**, возможность самому определить, что он будет рисовать. Классы **Text**, **Rectangle** и **Circle** могут иметь свои представления о том, что и как выводить на экран, рисуя самих себя. На самом деле это знает большинство классов. В конце концов, эти классы точно "знают", что именно они должны представлять. Например, объект класса **Circle** определяется точкой и радиусом, а не, скажем, множеством небольших отрезков. Генерация необходимых битов для объекта класса **Circle** на основе информации о центре и радиусе там, где это необходимо, и тогда, когда это необходимо, — не слишком сложная и затратная работа. По этой причине в классе **Circle** определяется собственная функция **draw_lines()**, которую мы хотим вызывать вместо функции **draw_lines()** из класса **Shape**. Именно это и означает ключевое слово **virtual** в объявлении функции **Shape::draw_lines()**:

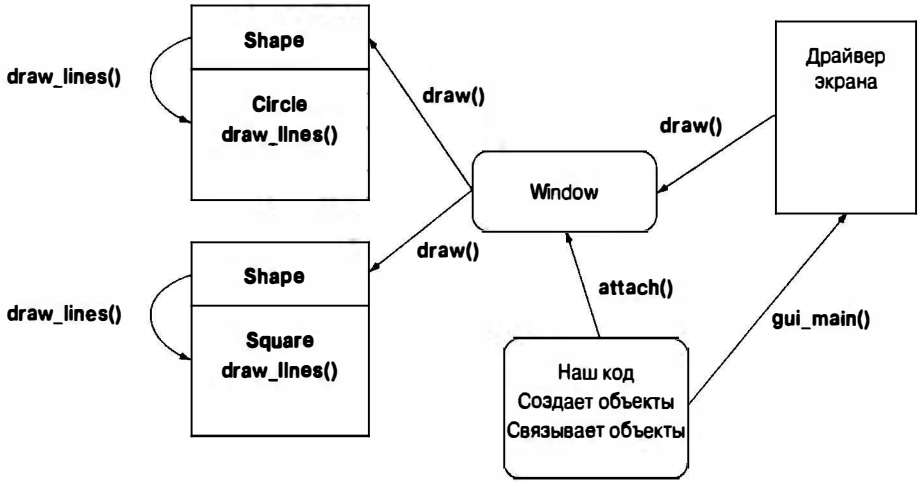
```
struct Shape {
    // ...
    virtual void draw_lines() const;
    // Пусть каждый производный класс при необходимости
    // сам определяет собственную функцию draw_lines()
    // ...
};

struct Circle : Shape {
    // ...
    void draw_lines() const; // "Перекрытие" функции Shape::draw_lines()
    // ...
};
```

Итак, вызов **draw_lines()** из класса **Shape** должен приводить к вызову этой функции-члена из класса **Circle**, если фигурой является объект класса **Circle**, и соответствующей функции-члена из класса **Rectangle**, если фигурой является объект класса **Rectangle**. Вот что означает слово **virtual** в объявлении функции **draw_lines()**: если класс является производным от класса **Shape**, то он должен самостоятельно объявить свою функцию **draw_lines()** (с тем же типом, что и функция **draw_lines()** класса **Shape**), которая будет вызываться вместо функции **draw_lines()** класса **Shape**. В главе 13 показано, как это сделано в классах **Text**, **Circle**, **Closed_polyline** и т.д. Определение функции в производном классе, используемой с помощью интерфейса базового класса, называют *перекрытием* (overriding).

Обратите внимание на то, что, несмотря на свою главную роль в классе **Shape**, функция **draw_lines()** объявлена как **protected**. Это означает,

что она не предназначена для вызова "обобщенным пользователем" — для этого есть функция **draw()**, — а является просто "деталью реализации", используемой функцией **draw()** и классами, производными от класса **Shape**.

На этом завершается описание нашей модели вывода, начатое в разделе 12.2. Система, управляющая экраном, знает о классе **Window**. Класс **Window** знает о классе **Shape** и может вызывать его функцию-член **draw()**. Наконец функция **draw()** вызывает функцию **draw_lines()** конкретной фигуры. Вызов функции **gui_main()** в нашем пользовательском коде запускает драйвер экрана.



Что же делает функция **gui_main()**? До сих пор мы не видели ее в нашей программе; вместо нее мы использовали функцию **wait_for_button()**, которая вызывала драйвер экрана более простым способом.

Функция **move()** класса **Shape** просто перемещает каждую хранимую точку на определенное расстояние относительно текущей позиции.

```
void Shape::move(int dx, int dy) // Перемещает фигуру +=dx и +=dy
{
    for (int i = 0; i<points.size(); ++i) {
        points[i].x+=dx;
        points[i].y+=dy;
    }
}
```

Подобно функции **draw_lines()**, функция **move()** является виртуальной, поскольку производный класс может иметь данные, которые необходимо переместить и о которых может не знать класс **Shape**. В качестве примера можно привести класс **Axis** (см. разделы 12.7.3 и 15.4).

Функция **move()** не является логически необходимой для класса **Shape**; мы ввели ее для удобства и в качестве примера еще одной виртуальной

функции. Каждый вид фигуры, имеющей точки, не хранящиеся в базовом классе **Shape**, должен определить собственную функцию **move()**.

14.2.4. Копирование и изменчивость

Класс **Shape** объявляет копирующий конструктор и копирующее присваивание удаленными:

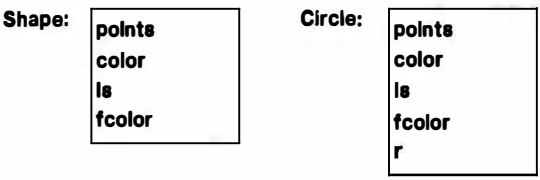
```
Shape(const Shape&) =delete; // Предотвращение копирования
Shape& operator=(const Shape&) =delete;
```

Результат заключается в удалении операций копирования по умолчанию, например:

```
void my_fct(Open_polyline& op, const Circle& c)
{
    Open_polyline op2 = op; // Ошибка: копирующий
                            // конструктор удален
    vector<Shape> v;
    v.push_back(c);         // Ошибка: копирующий
                            // конструктор удален
    // ...
    op = op2;               // Ошибка: присваивание
                            // Shape удалено
}
```

Однако копирование может быть полезным во многих ситуациях! Просто взгляните на функцию **push_back()**; без копирования было бы трудно использовать векторы (функция **push_back()** помещает в вектор *копию* своего аргумента). Почему надо беспокоиться о непредвиденном копировании? Операцию копирования по умолчанию запрещают, если она может привести к проблемам. В качестве примера такой проблемы рассмотрим функ-

цию **my_fct()**. Мы не можем копировать объект класса **Circle** в элемент вектора **v**, содержащий объекты типа **Shape** меньшего размера; объект класса **Circle** имеет радиус, а объект класса **Shape** — нет, так что **sizeof(Shape) < sizeof(Circle)**. Если бы мы допустили операцию **v.push_back(c)**, то объект класса **Circle** был бы "срезан" и любое последующее использование элемента вектора **v** привело бы к краху; операции класса **Circle** предполагают наличие радиуса (члена **r**), который не был бы скопирован.



Конструктор копирования объекта **op2** и оператор присваивания объекту **op** приводят к одной и той же проблеме:

```
Marked_polyline mp{"x"};
Circle c(p,10);
my_fct(mp,c); // Аргумент типа Open_polyline ссылается
              // на Marked_polyline
```

Теперь операции копирования класса **Open_polyline** приведут к срезке объекта **mark**, имеющего тип **string**.

Вообще говоря, иерархии классов и механизм передачи аргументов по ссылке не следует смешивать с копированием по умолчанию. Разрабатывая базовый класс иерархии, заблокируйте копирующий конструктор и операцию копирующего присваивания с помощью конструкции **=delete**, как мы сделали в классе **Shape**.

Срезка (это технический термин) — не единственная причина, по которой следует не допускать копирование. Существует еще несколько концепций, которые лучше представляются без операций копирования. Напомним, что для вывода объекта **Shape** на экран графическая система должна помнить, где он хранится. Вот почему мы связываем объекты класса **Shape** с объектами класса **Window**, а не копируем их. Например, если **Window** хранит копию **Shape**, изменения в оригинальном объекте никак не отразятся в копии. Так, если мы изменим цвет фигуры, то окно не будет об этом уведомлено и будет продолжать выводить на экран копию тем же цветом, что и ранее. В некотором смысле копия всегда хуже оригинала.

Если мы хотим скопировать объекты типа, в котором операции копирования по умолчанию были заблокированы, то можем написать явную функцию, выполняющую это действие. Такая функция копирования часто называется **clone()**. Очевидно, что функцию **clone()** можно написать, только если функций для чтения данных достаточно для реализации копирования, как в случае с классом **Shape**.

14.3. Базовые и производные классы

Посмотрим на базовый и производные классы с технической точки зрения; другими словами, в этом (и только этом) разделе предметом дискуссии будет не программирование, проектирование и графика, а возможности языка программирования. Разрабатывая нашу библиотеку графического интерфейса, мы использовали три ключевых механизма языка.

- *Порождение*. Это способ построения одного класса из другого так, чтобы новый класс можно было использовать вместо исходного. Например, класс **Circle** является производным от класса **Shape**; иначе говоря, класс **Circle** является разновидностью класса **Shape** или

класс **Shape** является базовым по отношению к классу **Circle**. Производный класс (в данном случае **Circle**) получает все члены базового класса (в данном случае **Shape**) в дополнение к собственным. Это свойство часто называют *наследованием* (inheritance), потому что производный класс наследует все члены базового класса. Иногда производный класс называют *подклассом* (subclass), а базовый — *суперклассом* (superclass).

- *Виртуальные функции*. В языке C++ можно определить функцию в базовом классе и функцию в производном классе с точно таким же именем и типами аргументов так, чтобы при вызове пользователем функции базового класса на самом деле вызывалась функция производного класса. Например, когда класс **Window** вызывает функцию **draw_lines()** объекта **Shape**, который на самом деле представляет собой объект класса **Circle**, выполняется именно функция **draw_lines()** класса **Circle**, а не функция **draw_lines()** класса **Shape**. Это свойство часто называют *полиморфизмом времени выполнения* (run-time polymorphism), *динамической диспетчеризацией* (dynamic dispatch) или *диспетчеризацией времени выполнения* (run-time dispatch), потому что вызываемая функция определяется во время выполнения программы на основе типа объекта, для которого она вызывается.

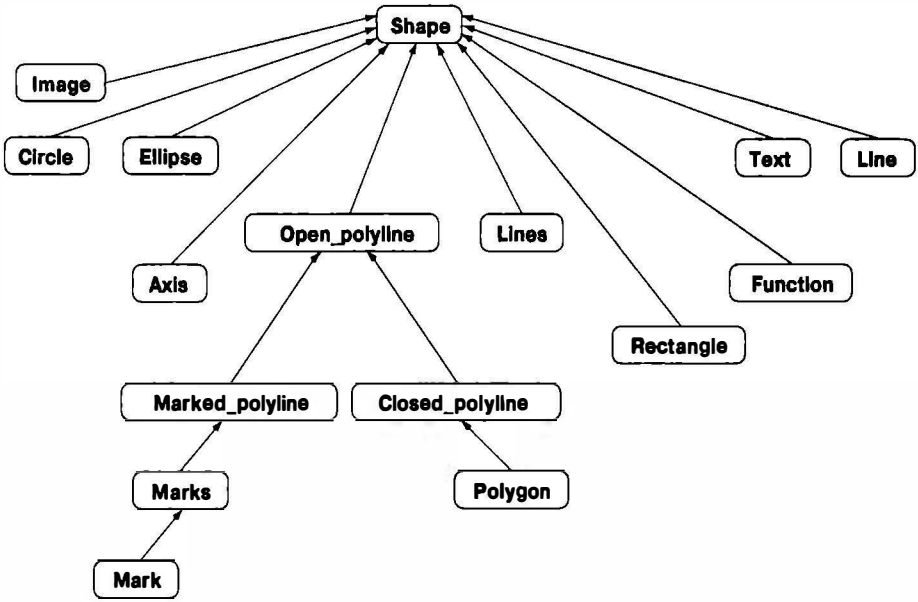
- *Закрытые и защищенные члены*. Мы закрываем детали реализации наших классов, чтоб защитить их от непосредственного использования, которое может затруднить поддержку программы. Это свойство часто называют *инкапсуляцией* (encapsulation).

Наследование, полиморфизм времени выполнения и инкапсуляция — наиболее распространенные характеристики *объектно-ориентированного программирования* (object-oriented programming). Таким образом, язык C++ непосредственно поддерживает объектно-ориентированное программирование наряду с другими стилями программирования. Например, в главах 20 и 21 мы увидим, как язык C++ поддерживает обобщенное программирование. Язык C++ позаимствовал — с явным признанием этого факта — эти ключевые механизмы из языка Simula67, первого языка, непосредственно поддерживавшего объектно-ориентированное программирование (подробнее об этом речь пойдет в главе 22).

Как много технической терминологии! Но что вся она означает? И как на самом деле работают эти механизмы? Давайте сначала нарисуем простую диаграмму наших классов графического интерфейса, показав их отношения наследования.

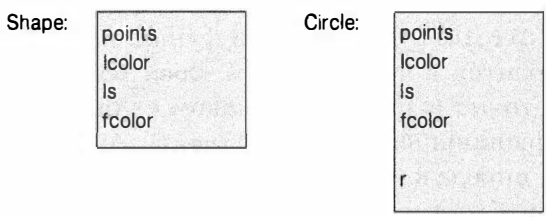
Стрелки направлены от производного класса к базовому. Такие диаграммы помогают визуализировать отношения между классами и часто украшают доски программистов. По сравнению с коммерческими программами эта иерархия классов невелика и содержит всего шестнадцать элементов.

Причем в этой иерархии только класс `Open_polyline` имеет несколько поколений наследников. Очевидно, что наиболее важным является общий базовый класс (`Shape`), несмотря на то, что он представляет абстрактную концепцию и никогда не используется для ее непосредственного воплощения в объекте.



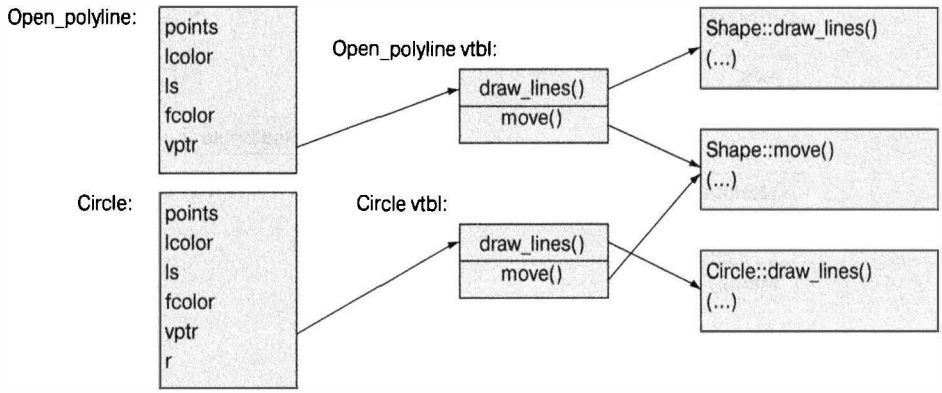
14.3.1. Схема объекта

Как объекты размещаются в памяти? Как было показано в разделе 9.4.1, схема объекта определяется членами класса: данные-члены хранятся в памяти один за другим. Если используется наследование, то данные-члены производного класса просто добавляются после членов базового класса, например:



Объект класса `Circle` имеет все данные-члены класса `Shape` (в конце концов, он является разновидностью класса `Shape`) и может быть использован вместо объекта класса `Shape`. Кроме того, объект класса `Circle` имеет собственный член `r`, который размещается в памяти после унаследованных данных-членов.

Для того чтобы обработать вызов виртуальной функции, нам нужна еще одна порция данных в объекте класса **Shape**: информация о том, какая функция будет на самом деле вызываться при обращении к функции **draw_lines()** класса **Shape**. Обычно для этого в таблицу функций заносится ее адрес. Эта таблица обычно называется **vtbl** (таблица виртуальных функций), а ее адрес часто называют **vptr** (виртуальный указатель, указатель на таблицу виртуальных функций). Указатели обсуждаются в главах 17 и 18; здесь они действуют как ссылки. В конкретных реализациях языка таблица виртуальных функций и виртуальный указатель могут называться иначе. Добавив эти таблицы и указатели к нашему рисунку, получаем следующую диаграмму:



Поскольку функция **draw_lines()** — первая виртуальная функция, она занимает первую ячейку в таблице **vtbl**, за ней следует вторая виртуальная функция **move()**. Класс может иметь сколько угодно виртуальных функций; его таблица **vtbl** может быть сколь угодно большой (по одной ячейке на каждую виртуальную функцию). Теперь, когда мы вызовем функцию **x.draw_lines()**, компилятор сгенерирует вызов функции, найденной в ячейке **draw_lines()** таблицы **vtbl**, соответствующей объекту **x**. По сути код просто следует по стрелкам на диаграмме. Так, если **x** представляет собой объект **Circle**, будет вызвана функция **Circle::draw_lines()**. Если объект **x** относится к типу, скажем, **Open_polyline**, который использует таблицу **vtbl** точно в том виде, в каком ее определил класс **Shape**, то будет вызвана функция **Shape::draw_lines()**. Аналогично, поскольку в классе **Circle** не определена его собственная функция **move()**, при вызове **x.move()**, если объект **x** относится к классу **Circle**, будет выполнена функция **Shape::move()**. По сути код, сгенерированный для вызова виртуальной функции, просто ищет указатель **vptr** и использует его для поиска соответствующей таблицы **vtbl** и вызова функции по указанному в ней адресу. Для этого требуются два обращения к памяти и обычный вызов функции, — быстро и просто.

Класс **Shape** является абстрактным, поэтому на самом деле объект класса **Shape** существовать не может, но класс **Open_polyline** имеет точно такую же структуру, как и "просто **Shape**", поскольку не добавляет никаких данных-членов и не определяет виртуальную функцию. Таблица виртуальных функций **vtbl** определяется для каждого класса с виртуальными функциями, а не для каждого объекта, поэтому таблицы **vtbl** увеличивают размер программы очень незначительно.

Обратите внимание на то, что на рисунке мы не изобразили ни одной невиртуальной функции. В этом не было необходимости, поскольку об этих функциях мы не можем сказать что-то особенное и они не увеличивают размеры объектов своего класса.

Определение функции, имеющей то же имя и те же типы аргументов, что и виртуальная функция в базовом классе (например, **Circle::draw_lines()**), так что функция производного класса записывается в таблицу **vtbl** вместо соответствующей функции базового класса, называется *перекрытием* (overriding). Например, функция **Circle::draw_lines()** перекрывает функцию **Shape::draw_lines()**.

Почему мы говорим о таблицах **vtbl** и схемах размещения в памяти? Нужна ли нам эта информация, чтобы использовать объектно-ориентированное программирование? Нет. Однако многие люди (и мы относимся к их числу) очень хотят знать, как устроены те или иные механизмы, а когда люди чего-то не знают, возникают мифы. Мы встречали людей, которые боялись использовать виртуальные функции, "потому что они снижают производительность и требуют больше памяти". Почему? Насколько? По сравнению с чем? Как оценить эти факторы? Мы объяснили модель реализации виртуальных функций, чтобы вы их не боялись. Если вам нужно вызвать виртуальную функцию (для выбора одной из нескольких альтернатив в ходе выполнения программы), то вы не сможете запрограммировать эту функциональную возможность с помощью другого языкового механизма, который работал бы быстрее или использовал меньше памяти, чем механизм виртуальных функций. Можете сами в этом убедиться.

14.3.2. Порождение классов и определение виртуальных функций

Мы указываем, что класс является производным, упоминая базовый класс после его имени:

```
struct Circle : Shape { /* ... */ };
```

По умолчанию члены структуры, объявляемой с помощью ключевого слова **struct**, являются открытыми (см. раздел 9.3), причем сюда включаются и открытые члены базового класса. Можно написать эквивалентный код следующим образом:

```
class Circle : public Shape { public: /* ... */ };
```

Эти два объявления класса **Circle** совершенно эквивалентны, но вы можете провести множество долгих и бессмысленных споров о том, какой из них лучше. Мы считаем, что время, которое можно затратить на эти споры, лучше посвятить другим темам.

Не забывайте ключевое слово **public**, когда захотите объявить открытые члены класса. Рассмотрим пример.

```
class Circle : Shape { public: /* ... */ }; // Возможно, ошибка
```

В этом случае класс **Shape** является закрытым базовым классом для класса **Circle**, что делает открытые функции-члены класса **Shape** недоступными для класса **Circle**. Вряд ли вы стремились к этому. Хороший компилятор предупредит вас о возможной ошибке. Закрытые базовые классы используются, но их описание выходит за рамки нашей книги.

Виртуальная функция должна объявляться с помощью ключевого слова **virtual** в объявлении класса, но если вы разместили определение функции за пределами класса, то ключевое слово **virtual** указывать не следует.

```
struct Shape {
    // ...
    virtual void draw_lines() const;
    virtual void move();
    // ...
};

virtual void Shape::draw_lines() const { /* ... */ } // Ошибка
void Shape::move() { /* ... */ }                     // ОК
```

14.3.3. Перекрытие

Если вы хотите перекрыть виртуальную функцию, то должны использовать точно такие же имя и тип функции, как и в базовом классе, например:

```
struct Circle : Shape {
    void draw_lines(int) const; // Возможно, ошибка
                                // (аргумент int?)
    void drawlines() const;     // Возможно, ошибка
                                // (опечатка в имени?)
    void draw_lines();          // Возможно, ошибка (нет const?)
    // ...
};
```

В данном случае компилятор увидит три функции, независимые от функции **Shape::draw_lines()** (поскольку они имеют другое имя или другой тип), и перекрытия не будет. Хороший компилятор предупредит программиста о возможных ошибках. В данном случае нет никаких признаков того, что вы действительно собирались перекрывать виртуальную функцию.

Пример функции **draw_lines()** реален, а потому его трудно описать во всех деталях, поэтому ограничимся чисто технической иллюстрацией перекрытия.

```
struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // Невиртуальная
};

struct D : B {
    void f() const { cout << "D::f "; } // Перекрывает B::f
    void g() { cout << "D::g "; }
};

struct DD : D {
    void f() { cout << "DD::f "; }       // Не перекрывает D::f
                                         // (нет const)
    void g() const { cout << "DD::g "; }
};
```

Здесь мы описали небольшую иерархию классов только с одной виртуальной функцией **f()**. Мы можем попробовать использовать ее. В частности, можем попробовать вызвать функцию **f()** и невиртуальную функцию **g()**, не знающие ничего о конкретном типе объекта, кроме того, что он относится к классу **B** (либо к классу, производному от **B**).

```
void call(const B& b)
    // Класс D - разновидность класса B, поэтому функция call()
    // может получать объект класса D. Класс DD - разновид-
    // ность класса D, а класс D - разновидность класса B,
    // поэтому функция call() может получать объект класса DD
{
    b.f();
    b.g();
}

int main()
{
    B b;
    D d;
    DD dd;

    call(b);
    call(d);
    call(dd);

    b.f();
    b.g();

    d.f();
    d.g();
```

```
    dd.f();
    dd.g();
}
```

В результате выполнения этой программы получим следующий вывод:

```
B::f B::g D::f B::g D::f B::g B::f B::g D::f D::g DD::f DD::g
```

Если вы понимаете, почему он именно такой, значит, вы знаете механизмы наследования и виртуальных функций.

Очевидно, что может быть достаточно трудно отслеживать, какие функции производного класса предназначены для перекрытия функций базового класса. К счастью, мы можем получить помощь от компилятора, если явно укажем, что функция предназначена для перекрытия. Для указания того, что данная функция должна перекрывать функцию базового класса, можно использовать ключевое слово **override**.

```
struct B {
    virtual void f() const { cout << "B::f "; }
    void g() const { cout << "B::g "; } // Не виртуальная
};
struct D : B {
    void f() const override   // Перекрывает B::f
        { cout << "D::f "; }
    void g() override         // Ошибка: нет виртуальной
        { cout << "D::g "; }  // функции B::g для перекрытия
};
struct DD : D {
    void f() override         // Ошибка: не перекрывает D::f
        { cout << "DD::f "; } // (нет const)
    void g() const override   // Ошибка: нет виртуальной
        { cout << "DD::g "; } // функции D::g для перекрытия
};
```

Явное использование ключевого слова **override** особенно полезно в больших, сложных иерархиях классов.

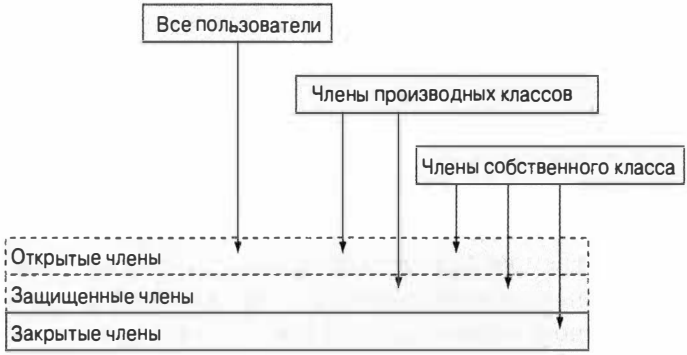
14.3.4. Доступ

Язык C++ реализует простую модель доступа к членам класса. Члены класса могут относиться к следующим категориям.

- *Закрытые* (private). Если член класса объявлен с помощью модификатора **private**, то его имя могут использовать только члены данного класса.
- *Защищенные* (protected). Если член класса объявлен с помощью модификатора **protected**, то его имя могут использовать только члены данного класса или члены классов, производных от него.

- *Открытые* (public). Если член класса объявлен с помощью модификатора **public**, то его имя могут использовать все функции.

Вот как это выглядит графически.



Базовый класс также может иметь атрибут **private**, **protected** или **public**.

- Если базовый класс для класса `D` является **private**, то имена его членов, объявленных как **public** и **protected**, могут использоваться только членами класса `D`.
- Если базовый класс для класса `D` является **protected**, то имена его членов, объявленных как **public** и **protected**, могут использоваться только членами класса `D` и членами классов, производных от класса `D`.
- Если базовый класс для класса `D` является **public**, то имена его членов, объявленных как **public**, могут использоваться любыми функциями.

Эти определения игнорируют понятие дружественной функции или класса и некоторые иные мелкие детали, которые выходят за рамки рассмотрения нашей книги. Если хотите стать языковым крючкотвором, читайте книги Страуструпа *Дизайн и эволюция языка C++* и *Язык программирования C++*, а также стандарт ISO C++. Однако мы не рекомендуем вам становиться крючкотвором (т.е. вникать в мельчайшие детали языковых определений); быть программистом (разработчиком программного обеспечения, инженером, пользователем, назовите как хотите) намного увлекательнее и обычно гораздо полезнее для общества.

14.3.5. Чисто виртуальные функции

Абстрактный класс — это класс, который можно использовать только в качестве базового. Абстрактные классы используются для представления абстрактных концепций; иначе говоря, мы используем абстрактные классы для описания концепций, которые являются обобщением общих ха-

рактеристик связанных между собой сущностей. Описанию *абстрактных концепций* (или *абстракций*, или *обобщений*, или...) посвящены толстенные книги по философии. Однако философское определение абстрактной концепции мало полезно. Примерами являются понятие "животное" (в противоположность конкретному виду животного), "драйвер устройства" (в противоположность драйверу конкретного вида устройств) или "публикация" (в противоположность конкретной книге или журналу). В программах абстрактные классы обычно определяют интерфейсы групп связанных между собой классов (*иерархий классов*).

В разделе 14.2.1 мы видели, как создать абстрактный класс, объявив его конструктор в разделе **protected**. Существует другой, гораздо более распространенный способ создания абстрактного класса: указать, что одна или несколько его виртуальных функций обязаны быть перекрыты в производном классе. Рассмотрим пример.

```
class B {                    // Абстрактный базовый класс
public:
    virtual void f() =0;     // Чисто виртуальная функция
    virtual void g() =0;
};

B b;                         // Ошибка: класс B – абстрактный
```

Интересное обозначение =0 указывает на то, что виртуальные функции **B::f()** и **B::g()** являются "чистыми", т.е. они должны быть перекрыты в некотором производном классе. Поскольку класс **B** содержит чисто виртуальную функцию, мы не можем создать объект этого класса. Перекрытие чисто виртуальных функций устраняет эту "проблему":

```
class D1 : public B {
public:
    void f() override;
    void g() override;
};

D1 d1;                       // OK
```

Обратите внимание, что класс остается абстрактным, пока не будут перекрыты все чисто виртуальные функции:

```
class D2 : public B {
public:
    void f() override;
            // Функция g() не перекрыта
};

D2 d2;      // ошибка: класс D2 – (все еще) абстрактный

class D3 : public D2 {
```

```
public:
    void g() override;
};

D3 d3;      // ok
```

Классы с чисто виртуальными функциями обычно являются чистыми интерфейсами; иначе говоря, они, как правило, не содержат данных-членов (эти данные хранятся в производных классах), а следовательно, не имеют конструкторов (если инициализация данных-членов не нужна, то и необходимость в конструкторах отпадает).

14.4. Преимущества объектно-ориентированного программирования

Когда мы говорим, что класс **Circle** является производным от класса **Shape** или что класс **Circle** является разновидностью класса **Shape**, то делаем это для того, чтобы достичь следующих целей (по отдельности или вместе).

- *Наследование интерфейса.* Функция, ожидающая аргумент класса **Shape** (обычно в качестве аргумента, передаваемого по ссылке), может принять аргумент класса **Circle** (и использовать его с помощью интерфейса класса **Shape**).
- *Наследование реализации.* Определяя класс **Circle** и его функции-члены, мы можем использовать возможности (такие, как данные и функции-члены), предоставляемые классом **Shape**.

Проект, в котором не используется наследование интерфейса (т.е. проект, в котором объект производного класса нельзя использовать вместо объекта его открытого базового класса), следует признать плохим и уязвимым для ошибок. Например, мы могли бы определить класс **Never_do_this**, для которого класс **Shape** является открытым базовым классом. Затем мы могли бы перекрыть функцию **Shape::draw_lines()** функцией, которая не рисует фигуру, а перемещает ее центр на 100 пикселей влево. Этот "проект" фатально неверен, поскольку, несмотря на то, что класс **Never_do_this** может предоставить интерфейс класса **Shape**, его реализация не поддерживает семантику (т.е. поведение), требуемое классом **Shape**. Никогда так не делайте!

Наследование интерфейса получило свое название потому, что его преимущества проявляются в использовании интерфейса, предоставленного базовым классом ("интерфейсом"; в данном случае — **Shape**) без информации о производных классах ("реализациях"; в данном случае — о классах, производных от класса **Shape**).

Преимущества наследования реализации получило свое название потому, что его преимущества проявляются в упрощении реализации производных классов (например, класса **Circle**), которое обеспечивается возможностями, предоставляемыми базовым классом (в данном случае — классом **Shape**).

Обратите внимание на то, что наш графический проект критично зависит от наследования интерфейса: "графический драйвер" вызывает функцию **Shape::draw()**, которая, в свою очередь, вызывает виртуальную функцию **draw_lines()** класса **Shape**, чтобы та выполнила реальную работу по выводу изображения на экран. Ни "графический драйвер", ни класс **Shape** не знают, какие виды фигур существуют. В частности, библиотека FLTK и графические средства операционной системы написаны и скомпилированы за много лет до создания наших графических классов! Мы просто определяем конкретные фигуры и вызываем функцию **attach()**, чтобы связать их с объектами класса **Window** в качестве объектов класса **Shape** (функция **Window::attach()** получает аргумент типа **Shape&**; см. раздел Д.3). Более того, поскольку класс **Shape** не знает о ваших графических классах, вам не нужно перекомпилировать класс **Shape** каждый раз, когда мы хотим определить новый класс графического интерфейса.

Иначе говоря, мы можем добавлять новые фигуры, не модифицируя существующий код. Это — "святой Грааль" проектирования, разработки и сопровождения программного обеспечения: расширение системы без ее модификации. Разумеется, существуют пределы, до которых мы можем расширять систему, не модифицируя существующие классы (например, тот же класс **Shape** предусматривает довольно ограниченный набор операций), и решить все проблемы программирования этот метод не в состоянии (например, в главах 17–19 определяется класс **vector**; наследование здесь мало может помочь). Однако наследование интерфейса — один из мощных методов проектирования и реализации систем, устойчивых к изменениям.

Аналогично многое позволяет сделать наследование реализации, также панацеей не являющееся. Помещая полезные функции в класс **Shape**, мы экономим силы, избегая дублирования кода в производных классах. Это может оказаться существенным фактором при разработке реальных про-

Однако этот эффект достигается ценой того, что любое изменение интерфейса класса **Shape** или любое изменение схемы размещения его данных-членов требует повторной компиляции всех производных классов и их клиентов. Для широко используемых библиотек такая повторная компиляция может оказаться неразрешимой проблемой. Естественно, существуют способы достичь указанных преимуществ и избежать большинства проблем (см. раздел 14.3.5).

Задание

К сожалению, мы не можем сформулировать задание, которое выявило бы понимание общих принципов проектирования, поэтому решили сосредоточиться на свойствах языка, поддерживающих объектно-ориентированное программирование.

1. Определите класс `B1` с виртуальной функцией `vf()` и невиртуальной функцией `f()`. Определите эти функции в классе `B1`. Реализуйте каждую функцию так, чтобы она выводила свое имя (например, "`B1::vf()`"). Сделайте эти функции открытыми. Создайте объект `B1` и вызовите каждую из функций.
2. Определите класс `D1`, производный от класса `B1`, и перекройте в нем функцию `vf()`. Создайте объект класса `D1` и вызовите из него функции `vf()` и `f()`.
3. Определите ссылку на объект класса `B1` (т.е. `B1&`) и инициализируйте ее только что созданным объектом класса `D1`. Вызовите функции `vf()` и `f()` с помощью этой ссылки.
4. Теперь определите функцию `f()` в классе `D1` и повторите пп. 1–3. Объясните результаты.
5. Добавьте в класс `B1` чисто виртуальную функцию `pvf()` и попытайтесь повторить пп. 1–4. Объясните результат.
6. Определите класс `D2`, производный от класса `D1`, и перекройте в нем функцию `pvf()`. Создайте объект класса `D2` и вызовите из него функции `f()`, `vf()` и `pvf()`.
7. Определите класс `B2` с чисто виртуальной функцией `pvf()`. Определите класс `D21` с данным-членом типа `string` и функцией-членом, перекрывающей функцию `pvf()`; функция `D21::pvf()` должна выводить значение упомянутого выше члена типа `string`. Определите класс `D22`, аналогичный классу `D21`, за исключением того, что его член имеет тип `int`. Определите функцию `f()`, получающую аргумент типа `B2&` и вызывающую функцию `pvf()` для этого аргумента. Вызовите функцию `f()` с аргументами `D21` и `D22`.

Контрольные вопросы

1. Что такое предметная область?
2. Как выглядит идеальное именование?
3. Что такое имя?
4. Какие возможности предоставляет класс `Shape`?

5. Чем абстрактный класс отличается от классов, не являющихся абстрактными?
6. Как сделать класс абстрактным?
7. Как управлять доступом?
8. Зачем нужен раздел `private`?
9. Что такое виртуальная функция и чем она отличается от невиртуальных функций?
10. Что такое базовый класс?
11. Как объявляется производный класс?
12. Что мы подразумеваем под схемой объекта?
13. Что можно сделать, чтобы класс было легче тестировать?
14. Что такое диаграмма наследования?
15. В чем заключается разница между защищенными и закрытыми членами класса?
16. К каким членам класса имеют доступ члены производного класса?
17. Чем чисто виртуальная функция отличается от других виртуальных функций?
18. Зачем делать функции-члены виртуальными?
19. Зачем делать функции-члены чисто виртуальными?
20. Что такое перекрытие?
21. Чем наследование интерфейса отличается от наследования реализации?
22. Что такое объектно-ориентированное программирование?

Термины

`private`	диспетчеризация	полиморфизм
`protected`	изменчивость	производный класс
`public`	инкапсуляция	суперкласс
абстрактный класс	наследование	схема объекта
базовый класс	объектно-ориентированный	таблица виртуальных функций
виртуальная функция	перекрытие	управление доступом
вызов виртуальной функции	подкласс	чисто виртуальная функция

Упражнения

1. Определите два класса, `Smiley` (веселый) и `Frowny` (хмурый), производные от класса `Circle` и имеющие два глаза и рот. Затем создайте

классы, производные от классов **Smiley** и **Frowny**, добавляющие к каждому из них соответствующую шляпу.

2. Попытайтесь скопировать объект класса **Shape**. Что произошло?
3. Определите абстрактный класс и попытайтесь определить объект его типа. Что произошло?
4. Определите класс **Immobile_Circle**, который во всем похож на класс **Circle**, но объекты которого не способны перемещаться.
5. Определите класс **Striped_rectangle**, в котором вместо заполнения прямоугольник заштриховывается горизонтальными линиями толщиной в один пиксель (например, рисуя линии через один пиксель). Поэкспериментируйте с толщиной линий и расстоянием между ними, чтобы добиться желаемого эффекта.
6. Определите класс **Striped_circle**, используя приемы из класса **Striped_rectangle**.
7. Определите класс **Striped_closed_polyline**, используя приемы из класса **Striped_rectangle** (для этого придется потрудиться всерьез).
8. Определите класс **Octagon**, реализующий правильный восьмиугольник. Напишите тестовую программу, вызывающую все его функции-члены (как определенные вами, так и унаследованные от класса **Shape**).
9. Определите класс **Group**, служащий контейнером объектов класса **Shape** с соответствующими операциями над членами класса **Group**. *Указание*: **Vector_ref**. Используя класс **Group**, определите класс, рисующий шахматную доску, по которой под управлением программы могут перемещаться шашки.
10. Определите класс **Pseudo_window**, напоминающий класс **Window**. Постарайтесь не прилагать героических усилий. Он должен иметь закругленные углы, метку и управляющие пиктограммы. Возможно, вы сможете добавить какое-нибудь фиктивное содержание, например изображение. На самом деле с этим изображением ничего не надо делать. Допускается (и даже рекомендуется), чтобы это окно располагалось в окне **Simple_window**.
11. Определите класс **Binary_tree**, производный от класса **Shape**. Задайте параметр, задающий количество уровней (**levels==0** означает, что в дереве нет ни одного узла, **levels==1** означает, что в дереве есть один узел, **levels==2** означает, что дерево состоит из вершины и двух узлов, **levels==3** означает, что дерево состоит из вершины и двух дочерних узлов, которые, в свою очередь, имеют по два дочерних узла, и т.д.). Пусть узел изображается маленьким кружочком. Соедините узлы линиями (как это принято).

P.S. В компьютерных науках деревья изображаются растущими вниз от вершины (забавно, что ее часто называют корнем, хотя это и логично).

12. Модифицируйте класс `Binary_tree` так, чтобы он рисовал свои узлы с помощью виртуальной функции. Затем унаследуйте от класса `Binary_tree` новый класс, в котором эта виртуальная функция перекрывается так, что узлы изображаются иначе (например, в виде треугольников).

13. Модифицируйте класс `Binary_tree` так, чтобы он получал параметр (или параметры), указывающие, какой вид линии используется для соединения узлов (например, стрелка, направленная вниз, или красная стрелка, направленная вверх). Обратите внимание, как в этом и последнем упражнениях используются два альтернативных способа, позволяющих сделать иерархию классов более гибкой и полезной.

14. Добавьте в класс `Binary_tree` операцию, добавляющую к узлу текст. Для того чтобы сделать это элегантно, можете модифицировать проект класса `Binary_tree`. Выберите способ идентификации узла; например, для перехода налево, направо, направо, налево и направо вниз по бинарному дереву можете использовать строку `"lrrlr"` (корневой узел может соответствовать как первому символу "`l`", так и первому символу "`r`").

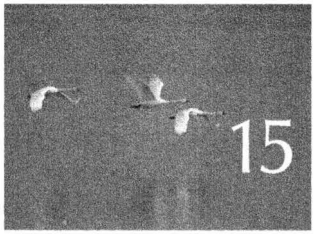
15. Большинство иерархий классов не связано с графикой. Определите класс `Iterator`, содержащий чисто виртуальную функцию `next()`, возвращающую указатель типа `double*` (см. главу 17). Теперь унаследуйте от класса `Iterator` классы `Vector_iterator` и `List_iterator` так, чтобы функция `next()` для класса `Vector_iterator` возвращала указатель на следующий элемент вектора типа `vector<double>`, а для класса `List_iterator` делала то же самое для списка типа `list<double>`. Инициализируйте объект класса `Vector_iterator` вектором `vector<double>`, и первый вызов функции `next()` должен возвращать указатель на первый элемент вектора, если таковой существует. Если такого элемента нет, верните нуль. Проверьте этот класс с помощью функции `void print(Iterator&)`, выводящей на печать элементы вектора типа `vector<double>` и списка типа `list<double>`.

16. Определите класс `Controller`, содержащий четыре виртуальные функции: `on()`, `off()`, `set_level(int)` и `show()`. Породите из класса `Controller` как минимум два класса. Один из них должен быть простым тестовым классом, в котором функция `show()` выводит на печать информацию о том, включен или выключен контроллер, а также его текущий уровень. Второй производный класс должен управлять цветом линий объекта класса `Shape`; точный смысл понятия "уровень" определите сами. Попробуйте найти третий объект для управления с помощью класса `Controller`.

17. Исключения, определенные в стандартной библиотеке языка C++, такие как `exception`, `runtime_error` и `out_of_range` (см. раздел 5.6.3), организованы в виде иерархии классов (с полезной виртуальной функцией `what()`, возвращающей строку, которая предположительно содержит объяснение происшедшей ошибки). Найдите источники информации об иерархии стандартных исключений в языке C++ и нарисуйте диаграмму этой иерархии классов.

Послесловие

Идеалом программирования вовсе не является создание одной программы, которая делает все. Идеал — создание множества классов, точно отражающих наши концепции, работающих вместе и позволяющих нам элегантно создавать приложения, затрачивая минимум усилий (по сравнению со сложностью задачи) при адекватной производительности и уверенности в правильности результатов. Такие программы понятны и удобны в сопровождении, т.е. их коды можно просто объединить, чтобы как можно быстрее выполнить поставленное задание. Классы, инкапсуляция (поддерживаемая с помощью модификаторов `private` и `protected`), наследование (поддерживаемое механизмом вывода классов), а также полиморфизм времени выполнения (поддерживаемый виртуальными функциями) являются одними из наиболее мощных средств структурирования систем.



15

Графическое представление функций и данных

Лучшее — враг хорошего.
— Вольтер (Voltaire)

В любой области приложений, связанной с эмпирическими данными или моделированием процессов, необходимо строить графики. В этой главе обсуждаются основные механизмы построения таких графиков. Как обычно, мы продемонстрируем использование таких механизмов и обсудим их проектирование. В качестве основного примера используются построение графика функции одного аргумента и графическое представление данных, считываемых из файла.

15.1. Введение

По сравнению с профессиональными программными системами, которые вы будете использовать, если визуализация данных станет вашим основным занятием, представленные в этой главе средства довольно примитивны. Наша главная цель — не элегантность вывода, а понимание того, как создается графический вывод и какие приемы программирования при этом используются. Методы проектирования, способы программирования и основные математические инструменты, представленные в главе, намного важнее, чем представленные графические средства. По этой причине мы не рекомендуем вам ограничиваться беглым просмотром фрагментов кода — они содержат намного больше интересной информации, чем простое рисование.

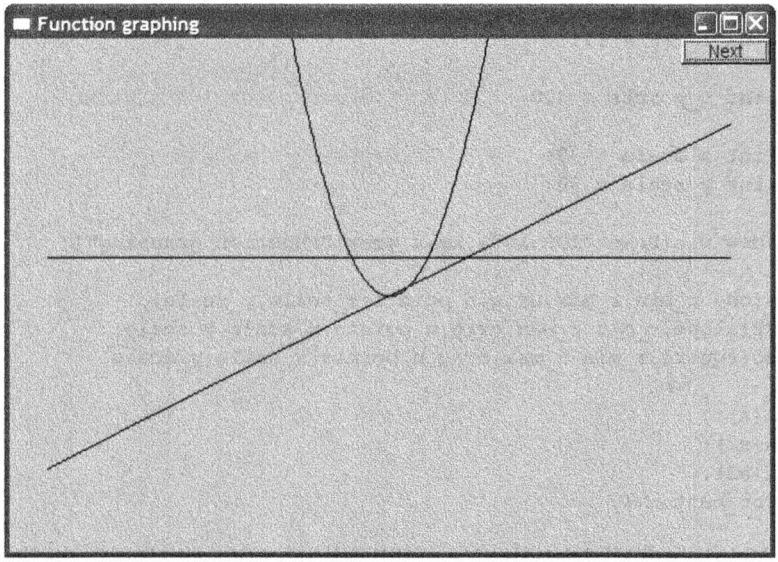
15.2. Графики простых функций

Начнем с рассмотрения примеров того, что мы можем нарисовать и как это реализовать в программе. В частности, рассмотрим используемые классы графического интерфейса. На приведенном рисунке мы видим параболу, горизонтальную и наклонную линии.

На самом деле, поскольку эта глава посвящена графикам функций, данная горизонтальная линия — это не просто горизонтальная линия, а график функции

```
double one(double) { return 1; }
```

Это самая простая функция, которую мы могли себе представить: она имеет один аргумент и всегда возвращает `1`. Поскольку для вычисления результата аргумент не нужен, именовать его необязательно. Для каждого значения `x`, переданного в качестве аргумента функции `one()`, получаем значение `y`, равное `1`; иначе говоря, эта линия определяется равенством `(x,y)==(x,1)` при всех `x`.



Как любой первый пример для начинающих, он выглядит несколько тривиальным, а рассуждения — педантичными, поэтому перейдем к немного более сложному примеру.

```
double slope(double x) { return x/2; }
```

Эта функция порождает наклонную линию. Для каждого аргумента `x` получаем значение `y`, равное `x/2`. Иначе говоря, `(x,y)==(x,x/2)`. Эти две линии пересекаются в точке `(2,1)`.

Теперь можем попытаться сделать что-то более интересное. Запишем квадратичную функцию, которая регулярно будет упоминаться в нашей книге.

```
double square(double x) { return x*x; }
```

Если вы помните школьную математику (и даже если забыли), эта функция определяет параболу, симметричную относительно оси *y*, а ее самая нижняя точка имеет координаты `(0,0)`, т.е. `(x,y)==(x,x*x)`. Итак, самая нижняя точка, в которой парабола пересекает рассмотренную наклонную линию, — это точка `(0,0)`.

Ниже приведен фрагмент кода, который рисует графики трех упомянутых выше функций.

```
constexpr int xmax = 600;        // Размер окна
constexpr int ymax = 400;

constexpr int x_orig = xmax/2;   // Центр окна (0,0)
constexpr int y_orig = ymax/2;
constexpr Point orig{x_orig,y_orig};
```

```
constexpr int r_min = -10;      // Диапазон [-10,11)
constexpr int r_max = 11;

constexpr int n_points = 400;   // Количество точек в диапазоне

constexpr int x_scale = 30;     // Масштабные множители
constexpr int y_scale = 30;

Simple_window win{Point{100,100},xmax,ymax,"Function graphing"};

Function s{one,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s2{slope,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s3{square,r_min,r_max,orig,n_points,x_scale,y_scale};

win.attach(s);
win.attach(s2);
win.attach(s3);
win.wait_for_button();
```

Сначала мы определяем набор именованных констант, чтобы не перегружать нашу программу "магическими константами". Затем создаем окно, определяем функции, связываем их с окном и передаем управление графической системе, которая выполняет реальное рисование на экране.

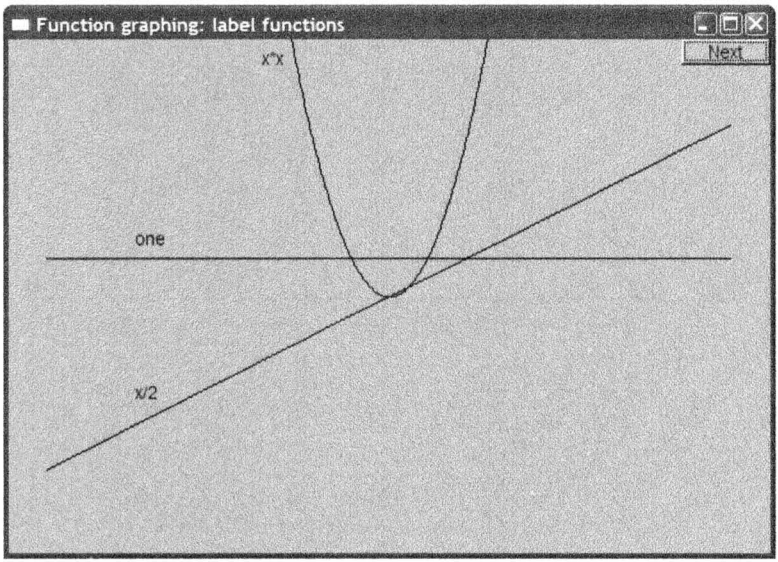
Все это делается по знакомому шаблону, за исключением определений трех объектов класса **Function**: **s**, **s2** и **s3**.

```
Function s{one,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s2{slope,r_min,r_max,orig,n_points,x_scale,y_scale};
Function s3{square,r_min,r_max,orig,n_points,x_scale,y_scale};
```

Каждый объект класса **Function** определяет, как его первый аргумент (функция от одного аргумента типа **double**, возвращающая значение типа **double**) будет нарисован в окне. Второй и третий аргументы задают диапазон изменения переменной **x** (аргумента изображаемой функции). Четвертый аргумент (в данном случае **orig**) сообщает объекту класса **Function**, в каком месте окна расположено начало координат **(0,0)**.

Если вы считаете, что в таком количестве аргументов легко запутаться, мы не станем спорить. В идеале аргументов должно быть как можно меньше, поскольку большое количество аргументов сбивает с толку и открывает возможности для ошибок. Однако пока мы не можем обойтись без них. Смысл последних трех аргументов мы объясним позже, в разделе 15.3. А для начала пометим наши графики.

Мы всегда стараемся сделать графики понятными без дополнительных разъяснений. Люди не всегда читают текст, сопровождающий рисунок, поэтому он часто оказывается бесполезным. Все, что мы изображаем на рисунках, должно помочь читателям понять его. В данном случае мы просто ставим метку у каждого графика. Код для создания меток создает три объекта класса **Text** (см. раздел 13.11).



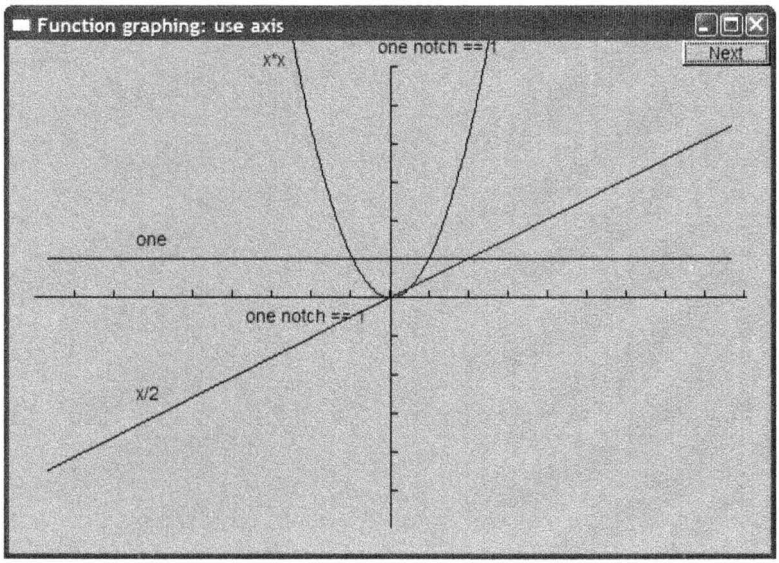

```
Text ts {Point{100,y_orig-40},"one"};
Text ts2 {Point{100,y_orig+y_orig/2-20},"x/2"};
Text ts3 {Point{x_orig-100,20},"x*x"};
win.set_label("Function graphing: label functions");
win.wait_for_button();
```

С этого момента на протяжении всей главы мы будем пропускать повторяющийся код, связывающий фигуру с окном, устанавливающий метку окна и ожидающий щелчка на кнопке Next.

Однако этот рисунок все еще нельзя считать законченным. Мы уже отмечали, что наклонная линия `x/2` пересекает параболу `x*x` в точках `(0,0)` и `(0.5,0.25)`, а график функции `one` пересекает линию `x/2` в точке `(2,1)`, но это известно лишь нам; для того чтобы это стало очевидно читателю, на рисунке следует вывести оси координат.

Код для построения осей содержит объявления двух объектов класса `Axis` (раздел 15.4).

```
constexpr int xlength = xmax-40;      // Оси чуть меньше окна
constexpr int ylength = ymax-40;

Axis x {Axis::x,Point{20,y_orig},
        xlength, xlength/x_scale, "one notch == 1"};
Axis y {Axis::y,Point{x_orig, ylength+20},
        ylength, ylength/y_scale, "one notch == 1"};
```



Использование значения **xlength/x_scale** в качестве параметра, задающего количество делений, позволяет использовать целочисленные отметки 1, 2, 3 и т.д. Выбор точки **(0,0)** в качестве начала координат является общепринятым. Если хотите, чтобы начало координат было не в центре, а, как обычно, в левом нижнем углу окна (раздел 15.6), вы легко сможете сделать это. Кроме того, для того чтобы отличать оси от графиков, можно использовать цвет.

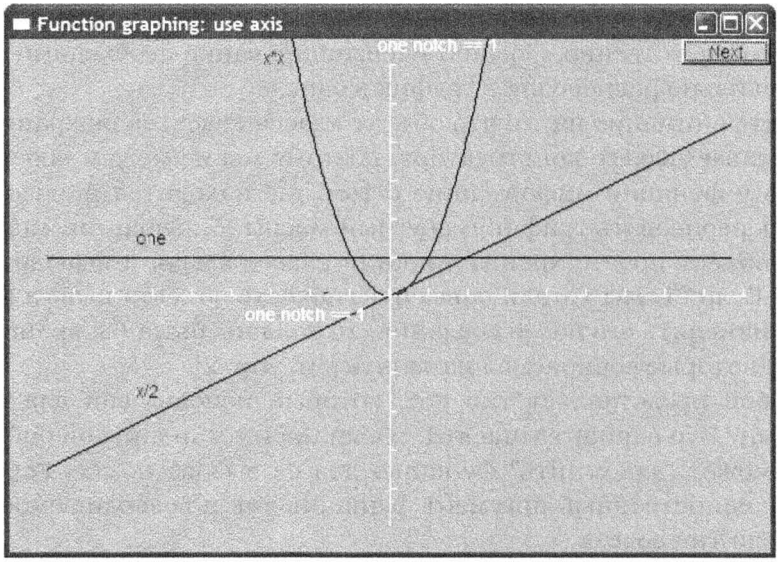
```
x.set_color(Color::red);
y.set_color(Color::red);
```

Так мы получаем результат, показанный ниже.

Такой рисунок вполне приемлем, но по эстетическим соображениям стоило бы оставить немного пустого места вверху — столько же, сколько внизу и по бокам. Кроме того, было бы неплохо отодвинуть метку оси *x* немного влево. Однако мы не будем этого делать, поскольку эстетичность графика можно обсуждать до бесконечности. Одно из профессиональных качеств программиста заключается в том, чтобы знать, когда остановиться и потратить сэкономленное время на что-нибудь более полезное (например, на изучение новых методов или на сон). Помните: "лучшее — враг хорошего".

15.3. Класс `Function`

Определение класса графического интерфейса **Function** приведено ниже.




```
struct Function : Shape {
    // Параметры не сохраняются
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale = 25, double yscale = 25);
};
```

Класс **Function** представляет собой **Shape**, конструктор которого генерирует множество прямолинейных отрезков и хранит их в членах класса **Shape**. Эти отрезки аппроксимируют значения функции **f**, которые вычисляются **count** раз в точках, равномерно распределенных по интервалу [**r1**,**r2**).

```
Function::Function(Fct f, double r1, double r2, Point xy,
                   int count, double xscale, double yscale)
    // Строит график функции f(x) для x из диапазона [r1,r2),
    // используя count отрезков. Начало координат (0,0)
    // располагается в точке xy;
    // координаты x масштабируются множителем xscale;
    // координаты y масштабируются множителем yscale
{
    if (r2-r1<=0) error("Неверный диапазон");
    if (count <=0) error("Не положительное значение count");
    double dist = (r2-r1)/count;
    double r = r1;
    for (int i = 0; i<count; ++i) {
        add(Point(xy.x+int(r*xscale),xy.y-int(f(r)*yscale)));
        r += dist;
    }
}
```

Параметры **xscale** и **yscale** используются для масштабирования координат x и y соответственно. Обычно масштабирование необходимо для того, чтобы правильно расположить график в окне.

Обратите внимание на то, что объект класса **Function** не хранит значения, передаваемые его конструктору, поэтому мы не можем впоследствии запросить у функции информацию о том, где находится начало координат, или перерисовать график с другими масштабирующими множителями. Этот объект просто хранит точки (в классе **Shape**) и выводит график на экран. Если бы мы хотели повысить гибкость объекта класса **Function**, позволяя изменять его после создания, то должны были бы хранить в нем значения, которые собираемся изменять (см. упр. 2).

Что собой представляет тип **Fct**, который использован для аргумента-функции? Это вариант типа **std::function** из стандартной библиотеки, который может "запомнить" функцию для ее вызова позже. **Fct** требует, чтобы как единственный аргумент функции, так и возвращаемое ею значение имели тип **double**.

15.3.1. Аргументы по умолчанию

Обратите внимание на способ инициализации аргументов **xscale** и **yscale** конструктора класса **Function**. Такой способ инициализации называют *аргументами по умолчанию* (default arguments). Их значения используются тогда, когда при вызове значения этих аргументов вообще не указываются.

```
Function s  {one, r_min, r_max,orig, n_points,
             x_scale, y_scale};
Function s2 {slope, r_min, r_max, orig, n_points,
             x_scale};                  // Нет yscale
Function s3 {square, r_min, r_max, orig,
             n_points};                 // Нет xscale, нет yscale
Function s4 {sqrt, r_min, r_max, orig}; // Нет count, нет xscale,
                                        // нет yscale
```

Этот фрагмент кода эквивалентен следующему:

```
Function s  {one, r_min, r_max, orig, n_points,
             x_scale, y_scale};
Function s2 {slope, r_min, r_max,orig, n_points, x_scale, 25};
Function s3 {square, r_min, r_max, orig, n_points, 25, 25};
Function s4 {sqrt, r_min, r_max, orig, 100, 25, 25};
```

Аргументы, заданные по умолчанию, являются альтернативой наличия нескольких перегруженных функций. Вместо определения одного конструктора с тремя аргументами по умолчанию мы могли бы определить четыре конструктора.

```
struct Function : Shape {      // Без аргументов по умолчанию
    Function(Fct f, double r1, double r2, Point orig,
            int count, double xscale, double yscale);
    // Масштаб y по умолчанию:
    Function(Fct f, double r1, double r2, Point orig,
             int count, double xscale);
    // Масштабы x и y по умолчанию:
    Function(Fct f, double r1, double r2, Point orig, int count);
    // Значение count по умолчанию; масштабы x и y по умолчанию:
    Function(Fct f, double r1, double r2, Point orig);
};
```

Для определения четырех конструкторов необходимо проделать существенно большую работу, при этом в определениях конструкторов природа значений, заданных по умолчанию, скрыта, в отличие от случая их явного задания в объявлении функции. Аргументы по умолчанию часто используются при объявлении конструкторов, но они могут быть полезными для любых функций. Определять аргументы по умолчанию можно лишь для завершающих аргументов, например:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale, double yscale); // Ошибка
};
```

Если некоторый аргумент имеет значение по умолчанию, то таковые же должны иметь и все последующие аргументы:

```
struct Function : Shape {
    Function(Fct f, double r1, double r2, Point orig ,
        int count = 100, double xscale=25, double yscale=25);
};
```

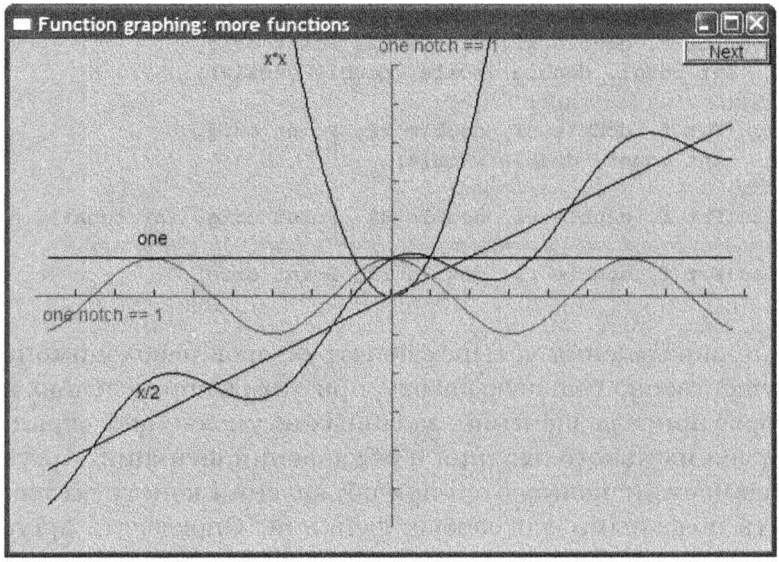
Иногда угадать удачные значения по умолчанию легко. Например, для строки хорошим выбором значения по умолчанию будет пустая строка, а для вектора — пустой вектор. В других ситуациях, например, для класса **Function**, правильно выбрать значения по умолчанию значительно сложнее: для этого приходится применять метод проб и ошибок. Помните, что вы не обязаны задавать значения по умолчанию и, если вам трудно это сделать, просто предоставьте задание значения аргумента пользователю.

15.3.2. Другие примеры

Мы добавили еще несколько функций — косинус (**cos**) из стандартной библиотеки и (просто для того, чтобы продемонстрировать, как создать сложную функцию) косинус с наклоном **x/2**.

```
double sloping_cos(double x) { return cos(x)+slope(x); }
```

Результат показан ниже.



Соответствующий фрагмент кода имеет вид

```
Function s4 {cos,r_min,r_max,orig,400,30,30};
s4.set_color(Color::blue);
Function s5 {sloping_cos, r_min,r_max,orig,400,30,30};
x.label.move(-160,0);
x.notches.set_color(Color::dark_red);
```

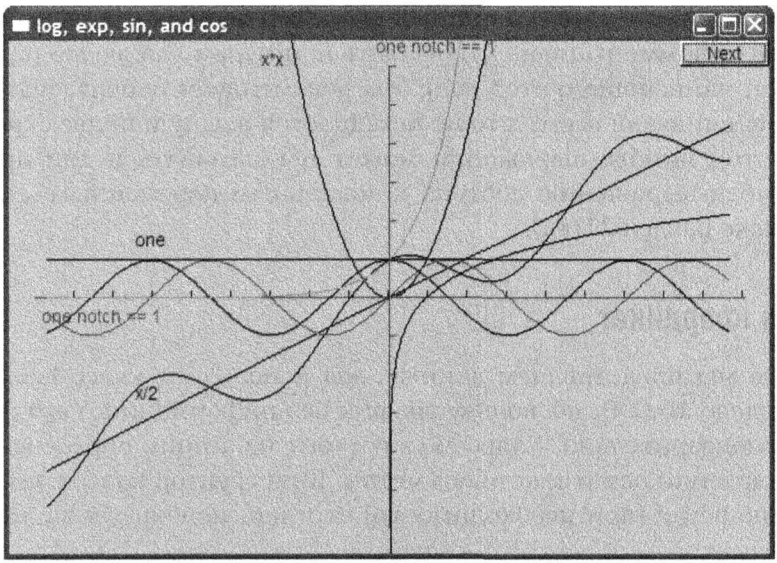
Кроме сложения этих двух функций, мы также сместили метку оси *x* и (просто для иллюстрации) немного изменили цвет шкалы.

В заключение построим графики логарифма, экспоненты, синуса и косинуса.

```
Function f1 {log,0.000001,r_max,orig,200,30,30}; // ln()
Function f2 {sin,r_min,r_max,orig,200,30,30};    // sin()
f2.set_color(Color::blue);
Function f3 {cos,r_min,r_max,orig,200,30,30};    // cos()
Function f4 {exp,r_min,r_max,orig,200,30,30};    // exp()
```

Поскольку значение `log(0)` не определено (с математической точки зрения оно равно бесконечности), мы начали диапазон для функции `log` с небольшого положительного числа. Вот как теперь выглядит наше окно:
Вместо приписывания меток этим графикам мы использовали цвет.

Стандартные математические функции, такие как `cos()`, `sin()` и `sqrt()`, объявлены в заголовочном файле заголовке `<cmath>` стандартной библиотеки. Список стандартных математических функций имеется в разделах 24.8 и Б.9.2.



15.3.3. Лямбда-выражения

Определять функцию только для того, чтобы передать ее в качестве аргумента в другую функцию — занятие утомительное. Поэтому C++ предлагает запись для определения чего-то, что действует как функция, прямо в позиции аргумента. Например, мы можем определить фигуру `sloping_cos` следующим образом:

```
Function s5 {[](double x) { return cos(x)+slope(x); },
             r_min,r_max,orig,400,30,30};
```

Запись `[](double x){return cos(x)+slope(x);}` представляет собой лямбда-выражение, т.е. неименованную функцию, определенную непосредственно там, где она требуется в качестве аргумента. Выражение `[]` называется *лямбда-инициатором* (lambda introducer). После инициатора лямбда-выражение указывает, какие аргументы ему требуются (список аргументов) и какие действия следует выполнять (тело функции). Возвращаемый тип может быть выведен из тела лямбда-выражения. В данном случае возвращаемый тип — `double`, поскольку это тип выражения `cos(x)+slope(x)`. Если мы хотим, то можем указать возвращаемый тип явно:

```
Function s5 {[](double x) -> double { return cos(x)+slope(x); },
                r_min,r_max,orig,400,30,30};
```

Указание возвращаемого типа в лямбда-выражении требуется очень редко. Основная причина этого в том, что лямбда-выражение должно быть как можно проще, чтобы не быть источником ошибок и путаницы. Если

фрагмент кода делает что-то значительное, он должен иметь имя и, вероятно, требует комментариев, чтобы быть понятным для людей (кроме программиста, написавшего этот код). Мы рекомендуем использовать именованную функцию для всего, что не помещается в одну или две строки.

Инициатор лямбда-выражения может использоваться для предоставления лямбда-выражению доступа к локальным переменным; см. раздел 15.5, а также раздел 21.4.3.

15.4. Оси координат

Там, где мы представляем данные, мы используем класс **Axis** (например, в разделе 15.6.4), поскольку график без информации о его масштабе выглядит подозрительно. Класс **Axis** состоит из линии, определенного количества делений оси и текстовой метки. Конструктор класса **Axis** вычисляет линию оси и (при необходимости) отрезки, используемые в качестве делений.

```
struct Axis : Shape {
    enum Orientation { x, y, z };
    Axis(Orientation d, Point xy, int length,
        int number_of_notches=0, string label = "");

    void draw_lines()  const  override;
    void move(int dx, int dy) override;
    void set_color(Color c);

    Text label;
    Lines notches;
};
```

Объекты **label** и **notches** остаются открытыми, поэтому пользователи могут ими манипулировать, например, приписывая делениям цвет, отличающийся от цвета линии, или перемещая объект **label** с помощью функции **move()** в более удобное место. Объект класса **Axis** — это пример объекта, состоящего из нескольких полунезависимых объектов.

Конструктор класса **Axis** размещает линии и добавляет на них деления, если значение **number_of_notches** больше нуля.

```
Axis::Axis(Orientation d, Point xy, int length,
           int n, string lab) :label(Point{0,0},lab)
{
    if (length<0) error("Неверная длина оси");
    switch (d){
    case Axis::x:
    {
        Shape::add(xy);       // Линия оси
        Shape::add(Point{xy.x+length,xy.y});
```

```
        if (0<n) {          // Добавление делений
            int dist = length/n;
            int x = xy.x+dist;
            for (int i = 0; i<n; ++i) {
                notches.add(Point{x,xy.y},Point{x,xy.y-5});
                x += dist;
            }
        }
        label.move(length/3,xy.y+20);        // Метка под линией
        break;
    }
    case Axis::y:
    {
        Shape::add(xy);       // Ось y, идущая вверх
        Shape::add(Point{xy.x,xy.y-length});
        if (0<n) {            // Добавление делений
            int dist = length/n;
            int y = xy.y-dist;
            for (int i = 0; i<n; ++i) {
                notches.add(Point{xy.x,y},Point{xy.x+5,y});
                y -= dist;
            }
        }
        label.move(xy.x-10,xy.y-length-10); // Метка вверху
        break;
    }
    case Axis::z:
        error("Ось z не реализована");
    }
}
```

По сравнению с большинством реальных программ этот конструктор очень прост, но мы рекомендуем внимательно изучить его, поскольку он не настолько тривиален, как кажется, и иллюстрирует несколько полезных приемов. Обратите внимание на то, как мы храним линию в части класса **Shape**, унаследованной классом **Axis** (используя функцию **Shape::add()**), хотя деления хранятся в отдельном объекте (**notches**). Это позволяет нам манипулировать линией и делениями оси независимо друг от друга; например, мы можем раскрасить их в разные цвета. Аналогично метка помещена в фиксированное положение, но, поскольку она является независимым объектом, мы всегда можем переместить ее в другое место. Для удобства и снижения вероятности внесения ошибок мы используем перечисление **Orientation**.

Поскольку класс **Axis** состоит из трех частей, мы должны предусмотреть функции для манипулирования объектом класса **Axis** в целом, например:

```
void Axis::draw_lines() const
{
    Shape::draw_lines();
```

```
    notches.draw(); // Цвет делений может отличаться
                    // от цвета линии
    label.draw();   // Цвет метки может отличаться
                    // от цвета линии
}
```

Для вывода объектов **notches** и **label** мы используем функцию **draw()**, а не **draw_lines()**, чтобы иметь возможность использовать информацию о цвете, которая в них хранится. Основная же линия хранится в самом **Axis::Shape** и использует информацию о цвете, хранящуюся там же.

Мы можем задать цвет линии, делений и метки по отдельности, но с точки зрения красоты стиля этого лучше не делать, поэтому мы предоставляем функцию, которая устанавливает все три цвета одинаковыми:

```
void Axis::set_color(Color c)
{
    Shape::set_color(c);
    notches.set_color(c);
    label.set_color(c);
}
```

Аналогично функция **Axis::move()** перемещает все части объекта класса **Axis** одновременно:

```
void Axis::move(int dx, int dy)
{
    Shape::move(dx,dy);
    notches.move(dx,dy);
    label.move(dx,dy);
}
```

15.5. Аппроксимация

Рассмотрим еще один небольшой пример построения графика функции: "анимируем" вычисление экспоненты. Наша цель — дать вам прочувствовать математические функции (если вы еще их не прочувствовали), продемонстрировать применение графиков для иллюстрации вычислений, показать фрагменты кода и, в заключение, предупредить о типичных проблемах, связанных с вычислениями.

Один из способов вычисления экспоненциальной функции представляет собой суммирование степенного ряда

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{К}$$

Чем больше членов ряда мы вычислим, тем точнее будет значение e^x; иначе говоря, чем больше членов ряда мы вычисляем, тем больше правильных цифр результата окажутся математически верными. В программе мы

суммируем ряд и строим график его частичных сумм. Знак восклицания в формуле, как обычно, обозначает факториал, т.е. мы строим графики функций в следующем порядке:

```
exp0(x) = 0              // Нет членов
exp1(x) = 1              // Один член
exp2(x) = 1+x            // Два члена; pow(x,1)/fac(1)==x
exp3(x) = 1+x+pow(x,2)/fac(2)
exp4(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)
exp5(x) = 1+x+pow(x,2)/fac(2)+pow(x,3)/fac(3)+pow(x,4)/fac(4)
...
```

Каждая функция немного точнее приближает e^x, чем предыдущая. Здесь **pow(x,n)** — стандартная библиотечная функция, возвращающая значение x^n. В стандартной библиотеке нет функции, вычисляющей факториал, поэтому мы должны определить ее самостоятельно.

```
int fac(int n) // factorial(n); n!
{
    int r = 1;
    while (n>1) {
        r*=n;
        --n;
    }
    return r;
}
```

Альтернативная реализация функции **fac()** описана в упр. 1. Имея функцию **fac()**, можем вычислить n-й член ряда:

```
double term(double x, int n) {  // n-й член ряда
    return pow(x,n)/fac(n);
}
```

Имея функцию **term()**, несложно вычислить экспоненты с точностью до **n** членов:

```
double expe(double x, int n) // Сумма n членов для значения x
{
    double sum = 0;
    for (int i=0; i<n; ++i) sum+=term(x,i);
    return sum;
}
```

Воспользуемся сказанным для построения нескольких графиков. Сначала проведем оси и "правильную" экспоненту (**exp()** из стандартной библиотеки), чтобы затем видеть, насколько близки к ней наши приближения функциями **expe()**:

```
Function real_exp {exp,r_min,r_max,orig,200,x_scale,y_scale};
real_exp.set_color(Color::blue);
```

Но как же воспользоваться функцией `expe()`? С точки зрения программиста трудность заключается в том, что наш класс `Function` получает имя функции от одного аргумента, а функция `expe()` имеет их два. В языке C++, в том виде, в котором мы были знакомы с ним до сих пор, нет возможности элегантного решения этой задачи. Однако такую возможность дают нам лямбда-выражения:

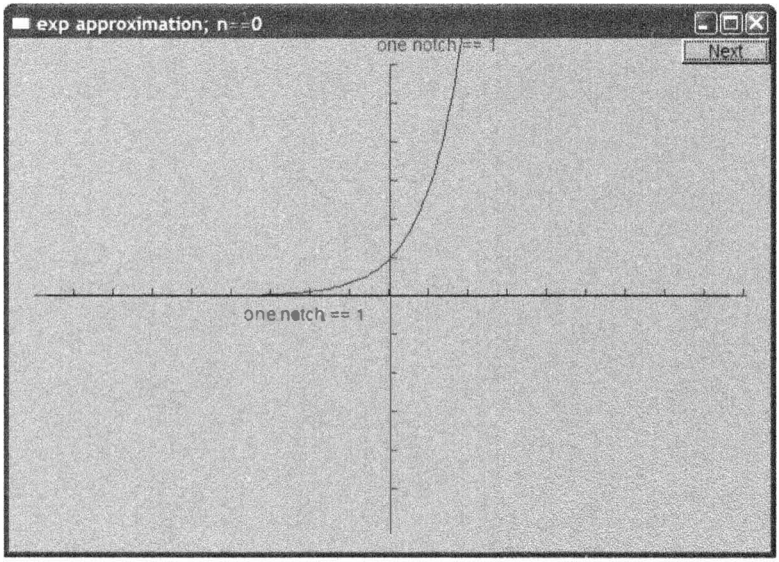
```
for (int n = 0; n<50; ++n) {
    ostringstream ss;
    ss << "exp approximation; n==" << n ;
    win.set_label(ss.str());
    // Очередное приближение:
    Function e {[n](double x) { return expe(x,n); },
            r_min,r_max,orig,200,x_scale,y_scale};
    win.attach(e);
    win.wait_for_button();
    win.detach(e);
}
```

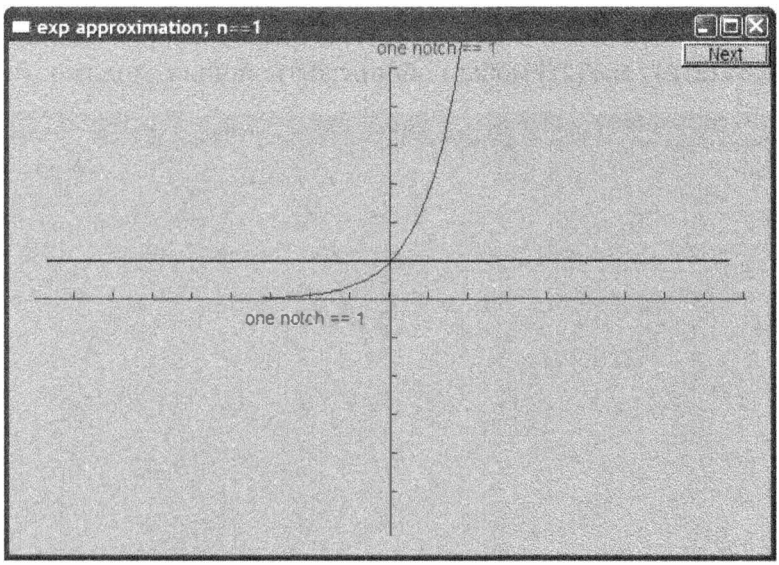
Лямбда-инициатор `[n]` гласит, что лямбда-выражение может обращаться к локальной переменной `n`. Таким образом, при создании объектов `Function` вызов `expe(x,n)` получает соответствующее значение `n` при каждом вызове внутри конструктора `Function`.

Обратите внимание на последний вызов `detach(e)` в этом цикле. Область видимости объекта `e` класса `Function` ограничена телом цикла `for`. Каждый раз входя в этот блок, мы создаем новый объект `e` класса `Function`, а при выходе из блока объект `e` уничтожается, а затем заменяется новым. Объект класса `Window` не должен помнить о старом объекте `e`, потому что он будет уничтожен. Вызов `detach(e)` гарантирует, что объект класса `Window` не попытается нарисовать разрушенный объект.

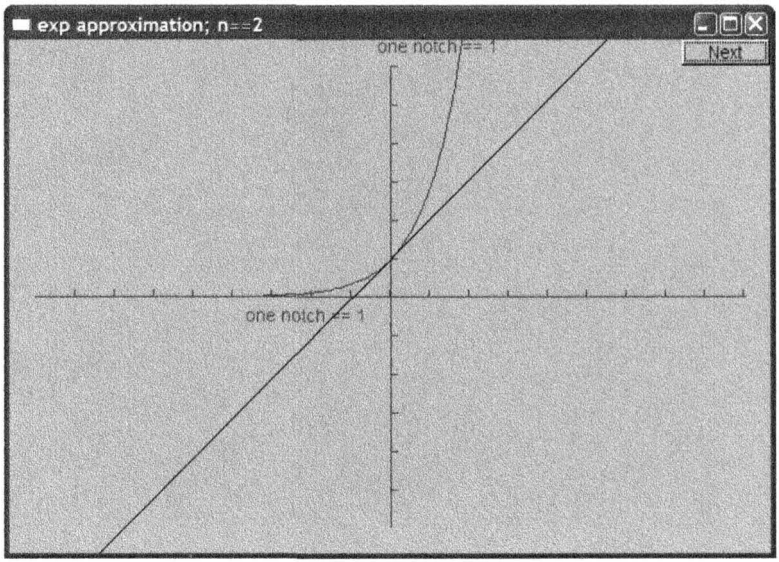
На первом этапе мы получаем окно, в котором нарисованы оси и "настоящая" экспонента синего цвета.

Как мы видим, значение `exp(0)` равно 1, поэтому наш синий график "настоящей" экспоненты пересекает ось y в точке `(0,1)`. Если присмотреться повнимательнее, то видно, что на рисунке нарисовано также первое приближение (`exp0(x)==0`) — черным цветом поверх оси x. Щелкая на кнопке Next, мы получаем приближение, содержащее один член степенного ряда. Обратите внимание на то, что мы указываем количество членов ряда, использованного для приближения экспоненты, в метке окна.

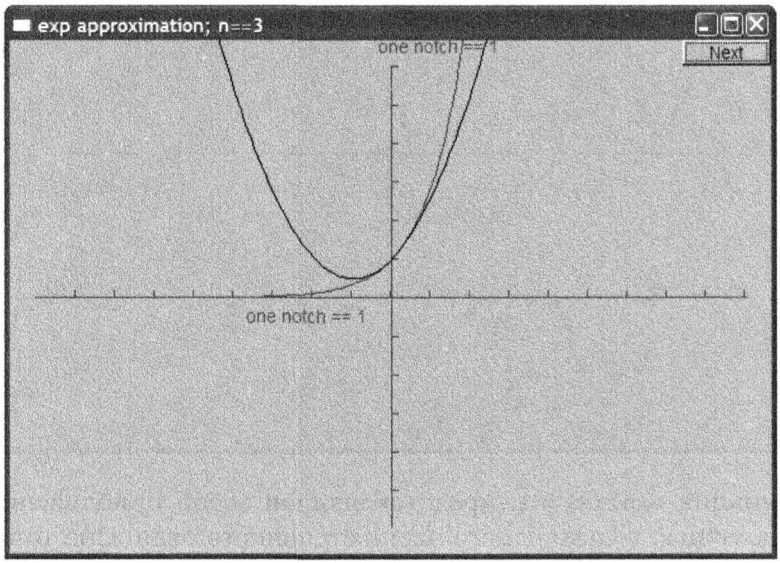




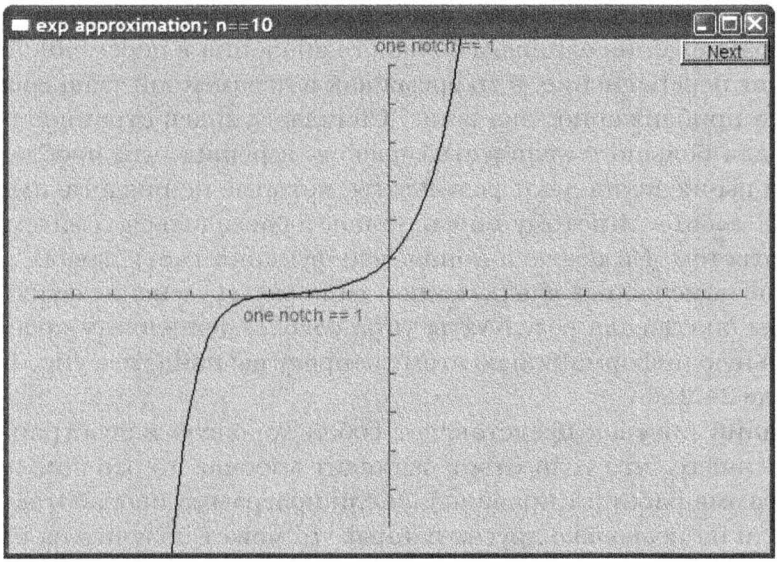
Это функция `exp1(x)==1`, представляющая собой приближение экспоненты с помощью только одного члена степенного ряда. Она точно совпадает с экспонентой в точке `(0,1)`, но мы можем построить и более точное приближение.



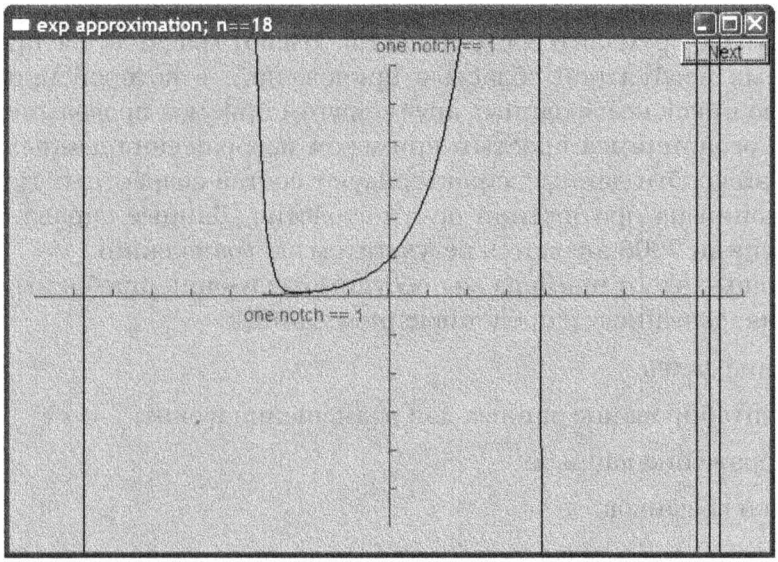
Используя два члена разложения (`1+x`), получаем диагональную прямую, пересекающую ось *y* в точке `(0,1)`. С помощью трех членов разложения (`1+x+pow(x,2)/fac(2)`) можем обнаружить начало сходимости.



Десять членов приближения дают очень хорошее приближение, особенно для значений *x*, превышающих –3.



На первый взгляд, мы могли бы получать все более точные аппроксимации, постоянно увеличивая количество членов степенного ряда. Однако существует предел, и после тринадцати членов происходит нечто странное: аппроксимация ухудшается, а после вычисления восемнадцати членов на рисунке появляются вертикальные линии.



Вспомните, что арифметика чисел с плавающей точкой — это не чистая математика. Числа с плавающей точкой просто являются приближением действительных чисел, поскольку для их пред-

ставления можно использовать лишь ограниченное количество битов. При попытке размещения слишком большого значения в переменной типа `int` произойдет переполнение, в то время как в переменной типа `double` будет храниться приближенное значение. Сначала, увидев странное поведение графика для большого количества членов, я решил, что проблема в том, что наши вычисления дают результаты, которые не представимы с помощью типа `double`, и потому они начинают расходиться с математически верным ответом. Но позже я понял, что функция `fac()` давала значения, которые не помещались в переменной типа `int`. Изменение функции `fac()` так, чтобы она давала результаты типа `double`, решило эту проблему. Дополнительную информацию по этому вопросу вы найдете в упр. 11 главы 5 и в разделе 24.2.

Последний рисунок представляет собой хорошую иллюстрацию принципа, гласящего, что если ответ выглядит хорошо, то это еще не значит, что программа работает правильно. Если программа проработает немного дольше или на несколько других данных, то может получиться настоящая ерунда, как и произошло в данном примере.

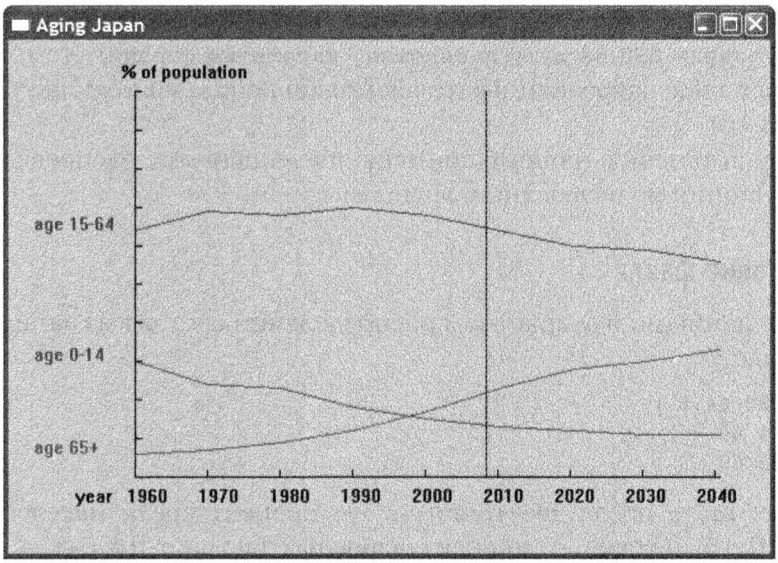
15.6. Графическое представление данных

Изображение данных требует большой подготовки и опыта. Хорошо представленные данные сочетают технические и художественные факторы и могут существенно облегчить анализ сложных явлений. В то же время эти обстоятельства делают графическое представление данных необъятной областью приложений, в которой применяется множество никак не связанных друг с другом приемов программирования. Здесь мы ограничимся простым примером изображения данных, считанных из файла. Эти данные характеризуют состав возрастных групп населения Японии на протяжении почти столетия. Данные справа от вертикальной линии 2008 являются результатом экстраполяции.

С помощью этого примера мы обсудим следующие проблемы программирования, связанные с представлением данных:

- чтение файла;
- масштабирование данных для размещения в окне;
- отображение данных;
- метки графиков.

Мы не будем вдаваться в художественные аспекты этой проблемы. В принципе, мы строим "график для идиотов", а не для художественной галереи. Очевидно, что, когда это потребуется, вы сможете построить его намного более красиво.



Имея набор данных, мы должны подумать о том, как их получше изобразить на экране. Для простоты ограничимся только данными, которые легко изобразить на плоскости, ведь именно такие данные образуют огромный массив приложений, с которыми работает большинство людей. Обратите внимание на то, что гистограммы, секторные диаграммы и другие популярные виды диаграмм на самом деле просто красиво отображают двумерные данные. Трехмерные данные часто обрабатываются с помощью серий двумерных изображений или размещения нескольких двумерных графиков в одном окне (как в приведенном выше примере с возрастными группами населения Японии) или путем пометки отдельных точек с информацией. Если мы хотим выйти за рамки описанных решений, либо нам нужны новые графические классы, либо следует адаптировать другую графическую библиотеку.

Итак, наши данные в основном представляют собой пары точек, такие как **(год, количество детей)**. Если у нас есть больше данных, например **(год, количество детей, количество взрослых, количество стариков)**, то мы должны просто решить, какую пару или пары значений мы хотим выводить. В нашем примере мы выводим пары **(год, количество детей)**, **(год, количество взрослых)** и **(год, количество стариков)**.

Имеется много способов интерпретации пар **(x,y)**. Решая, как изобразить такие данные, важно понять, можно ли их представить в виде функции одного значения от другого. Например, для пары **(year,steel_production)** разумно предположить, что производство стали (**steel_production**) является функцией, зависящей от года (**year**), и изобразить данные в виде непрерывной линии. Для изображения таких данных хорошо подходит класс **Open_polyline** (см. раздел 13.6). Если переменная **y**

не должна рассматриваться как функция от переменной **x** (как, например, в паре **(уровень ВВП на душу населения, население страны)**, то для их изображения в виде разрозненных точек можно использовать класс **Marks** (см. раздел 13.15).

Вернемся теперь к нашему примеру, посвященному распределению населения Японии по возрастным группам.

15.6.1. Чтение файла

Файл с данными о возрастном распределении состоит из записей следующего вида:

```
( 1960 : 30 64 6 )
(1970 : 24 69 7 )
(1980 : 23 68  9 )
```

Первое число после двоеточия — это процент среди населения детей (возраст 0–15), второе — процент взрослых (возраст 15–64), а третье — процент пожилых людей (возраст 65 и выше). Наша задача — прочитать эти данные из файла. Обратите внимание на то, что форматирование данных носит несколько нестрогий характер. Как обычно, мы должны уделить внимание таким деталям.

Для того чтобы упростить задачу, сначала определим тип **Distribution** для хранения данных и оператор ввода этих данных.

```
struct Distribution {
    int year, young, middle, old;
};

istream& operator>>(istream& is, Distribution& d)
    // Предполагаемый формат: ( год : дети взрослые старики )
{
    char ch1 = 0;
    char ch2 = 0;
    char ch3 = 0;
    Distribution dd;

    if (is >> ch1 >> dd.year
           >> ch2 >> dd.young >> dd.middle >> dd.old
           >> ch3) {
        if (ch1!= '(' || ch2!=':' || ch3!=')') {
            is.clear(ios_base::failbit);
            return is;
        }
    }
    else
        return is;
    d = dd;
```

```
    return is;
}
```

Этот код является результатом непосредственного воплощения идей, изложенных в главе 10. Если какие-то места этого кода вам не ясны, пожалуйста, перечитайте эту главу. Мы не обязаны определять тип `Distribution` и оператор `>>`. Однако это упрощает код по сравнению с методом решения "в лоб", основанным на принципе "просто прочитать данные и построить график". Наше использование класса `Distribution` разделяет код на логические части, что облегчает его анализ и отладку. Не бойтесь вводить типы просто для того, "чтобы сделать код яснее". Мы определяем классы, чтобы программа точнее соответствовала нашему представлению об основных концепциях предметной области. В этом случае даже "небольшие" концепции, использованные в исходном тексте очень локально (такие, как, например, линия, представляющая распределение возрастов по годам), могут оказаться очень полезными.

Имея тип `Distribution`, можно записать цикл чтения данных следующим образом.

```
string file_name = "japanese-age-data.txt";
ifstream ifs{file_name};
if (!ifs) error("Невозможно открыть файл ",file_name);

// ...

for(Distribution d; ifs>>d; ) {
     if (d.year < base_year || end_year < d.year)
          error("Год вне диапазона");
     if (d.young + d.middle + d.old != 100)
          error("Проценты не согласуются");
     // ...
}
```

Иначе говоря, мы пытаемся открыть файл `japanese-age-data.txt` и выходим из программы, если это не удается. Идея *не* "прошивать" явно имя файла в программе часто оказывается удачной, но в данном случае мы пишем простой пример и не хотим прилагать лишние усилия, более подходящие для коммерческих, не разовых приложений. С другой стороны, мы присваиваем имя файла `japanese-age-data.txt` именованной переменной типа `string`, поэтому при необходимости его легко изменить.

Цикл чтения проверяет диапазон чисел и согласованность данных. Это основные правила проверки таких данных. Поскольку оператор `>>` сам проверяет формат каждого элемента данных, в цикле чтения больше нет никаких проверок.

15.6.2. Общая схема

Что мы хотим увидеть на экране? Этот ответ можно найти в начале раздела 15.6. На первый взгляд, для изображения данных нужны три объекта класса `Open_polyline` — по одному на каждую возрастную группу. Каждый график должен быть помечен. Для этого мы решили в левой части окна записать "название" каждой линии. В данном случае этот выбор кажется удачнее распространенного альтернативного решения разместить метку на самой линии. Кроме того, для того чтобы отличать графики друг от друга, для каждого графика (и его метки) мы используем разные цвета.

Мы хотим пометить ось *x*, указав годы. Вертикальная линия, проходящая через отметку 2008, означает год, после которого данные являются результатом экстраполяции.

В качестве названия изображения мы решили использовать метку окна.

Сделать так, чтобы графический код был правильным и хорошо выглядел, — довольно сложная задача. Основная причина заключается в том, что нам придется выполнить множество кропотливых вычислений, связанных с определением размеров и смещений. Для их упрощения мы начали с определения символических констант, определяющих способ использования экрана.

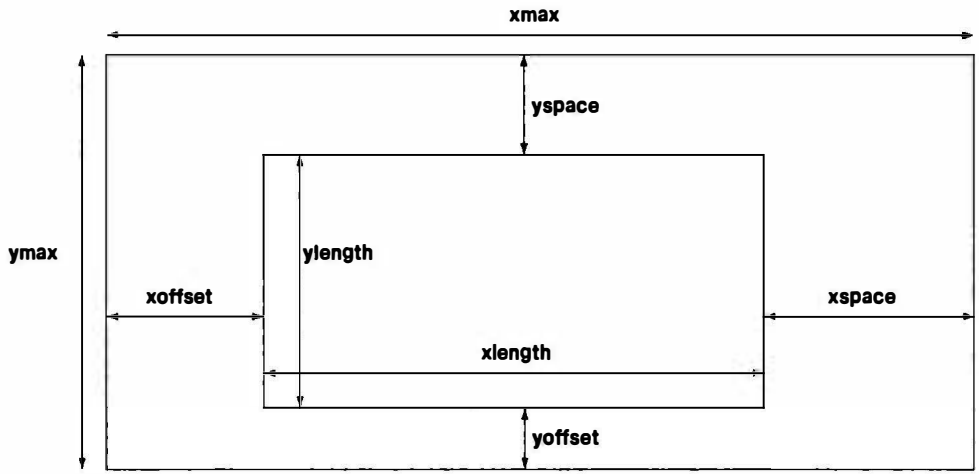
```
constexpr int xmax    = 600; // Размер окна
constexpr int ymax    = 400;
                             // Расстояния:
constexpr int xoffset = 100; //   от левого края до оси y
constexpr int yoffset =  60; //   от нижнего края до оси x

constexpr int xspace  =  40; // Пространство за осями
constexpr int yspace  =  40;
                             // Длины осей
constexpr int xlength = xmax-xoffset-xspace;
constexpr int ylength = ymax-yoffset-yspace;
```

Эти величины определяют прямоугольную область (окно) и вложенный в него прямоугольник (определенный осями).

Без такого схематического наброска размещения элементов экрана в нашем окне с помощью символических констант мы бы очень быстро запутались, и вывод на экран был бы безнадежно испорчен.

15.6.3. Масштабирование данных

Теперь мы должны определить, как изобразить данные в описанной области. Для этого мы масштабируем данные так, чтобы они помещались в прямоугольнике, определенном осями координат. Масштабирование осуществляется с помощью масштабных множителей, представляющих собой отношение диапазонов изменения данных и меток на осях.




```
constexpr int base_year = 1960;
constexpr int end_year = 2040;

constexpr double xscale = double(xlength)/(end_year-base_year);
constexpr double yscale = double(ylength)/100;
```

Мы объявили наши масштабирующие множители (**xscale** и **yscale**) как числа с плавающей точкой — иначе в наших вычислениях возникли бы серьезные ошибки, связанные с округлением. Для того чтобы избежать целочисленного деления, перед делением мы преобразовываем наши длины в тип **double** (см. раздел 4.3.3).

Теперь можно поместить точки на ось *x*, вычитая их базовое значение (1960), масштабируя с помощью множителя **xscale** и добавляя смещение **xoffset**. Значение *y* обрабатывается аналогично. Эти операции тривиальны, но кропотливы и скучны. Для того чтобы упростить код и минимизировать вероятность ошибок (а значит, минимизировать и отладку), мы определили небольшой класс, который выполняет эти вычисления.

```
class Scale {       // Класс для преобразования координат
    int cbase;      // Координатная база
    int vbase;      // База значений
    double scale;
public:
    Scale(int b, int vb, double s):cbase(b), vbase(vb), scale(s)
    { }
    int operator()(int v) const
    { return cbase + (v-vbase)*scale; } // См. раздел 21.4
};
```

Мы хотим создать класс, поскольку вычисления зависят от трех константных значений, которые мы хотели бы сделать не обязательно повторяющимися. Тогда мы можем определить следующие объекты:

```
Scale xs{xoffset,base_year,xscale};
Scale ys{ymax-yoffset,0,-yscale};
```

Обратите внимание на то, что мы сделали масштабирующий множитель `ys` отрицательным, чтобы отразить тот факт, что координаты *y* возрастают в направлении вниз, хотя мы привыкли, что обычно они возрастают в направлении вверх. Теперь можно использовать `xs` для преобразования лет в координату *x*, а `ys` — для преобразования процентов в координату *y*.

15.6.4. Построение графика

Итак, у нас есть все необходимое для создания элегантной программы. Начнем с создания окна и размещения осей.

```
Window win {Point{100,100},xmax,ymax,"Aging Japan"};

Axis x {Axis::x, Point{xoffset,ymax-yoffset}, xlength,
        (end_year-base_year)/10,
        "year 1960 1970 1980 1990 "
        "2000 2010 2020 2030 2040"};
x.label.move(-100,0);

Axis y {Axis::y, Point{xoffset,ymax-yoffset}, ylength, 10,
        "% of population"};

Line current_year {Point{xs(2008),ys(0)},Point{xs(2008),ys(100)}};
current_year.set_style(Line_style::dash);
```

Оси пересекаются в точке `Point{xoffset,ymax-yoffset}`, представляющей значение `(1960,0)`. Обратите внимание на то, как размещены деления, отражающие данные. На оси *y* отложено десять делений, каждое из которых соответствует десяти процентам населения. На оси *x* каждое деление соответствует десяти годам. Точное количество делений вычисляется по значениям переменных `base_year` и `end_year`, поэтому, если мы изменим диапазон, оси автоматически будут вычислены заново. Это одно из преимуществ отсутствия "магических констант" в коде. Метка на оси *x* нарушает это правило, потому что размещать метки, пока числа на окажутся в правильных позициях, бесполезно. Возможно, лучше было бы задать набор индивидуальных меток для каждого деления.

Пожалуйста, обратите внимание на любопытное форматирование этой метки, представляющей собой строку. Мы использовали два смежных строковых литерала.

```
"year 1960 1970 1980 1990 "
"2000 2010 2020 2030 2040"
```

Компилятор соединяет такие строки в одну, поэтому эта запись эквивалентна следующей строке:

```
"year 1960 1970 1980 1990 2000 2010 2020 2030 2040"
```

Этот трюк может оказаться полезным при размещении длинных строк, поскольку он позволяет сохранить удобочитаемость текста.

Объект `current_year` соответствует вертикальной линии, разделяющей реальные и прогнозируемые данные. Обратите внимание на то, как используются для правильного размещения и масштабирования этой линии объекты `xs` и `ys`.

Построив оси, мы можем обработать данные. Определим три объекта класса `Open_polyline` и заполним их в цикле чтения.

```
Open_polyline children;
Open_polyline adults;
Open_polyline aged;

for (Distribution d; ifs>>d; ) {
    if (d.year < base_year || end_year < d.year)
        error("Год вне диапазона");
    if (d.young + d.middle + d.old != 100)
        error("Проценты не согласуются");
    const int x = xs{d.year};
    children.add(Point{x,ys(d.young)});
    adults.add(Point{x,ys(d.middle)});
    aged.add(Point{x,ys(d.old)});
}
```

Использование объектов `xs` и `ys` делает задачу масштабирования и размещения данных тривиальной. "Небольшие классы", такие как `Scale`, могут оказаться очень важными для упрощения кода и устранения лишних повторов — тем самым они повышают удобочитаемость и увеличивают шансы создания правильной программы.

Для того чтобы графики были более ясными, мы пометили их и раскрасили в разные цвета.

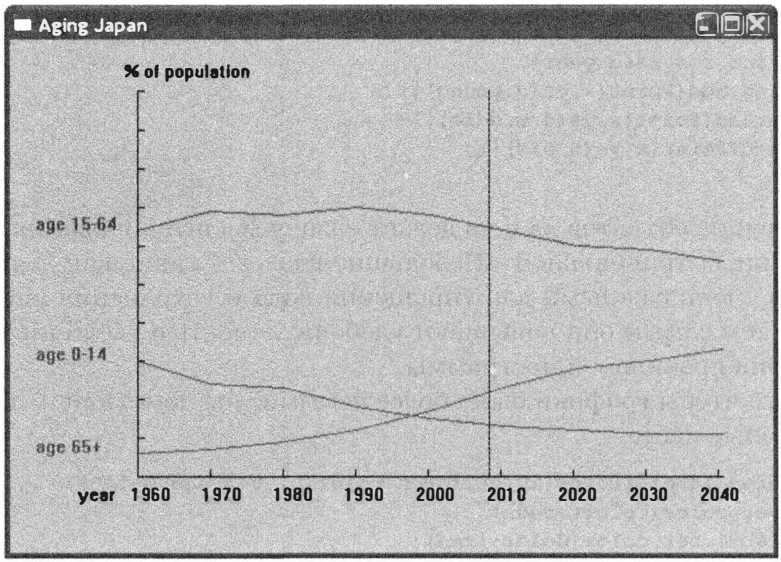
```
Text children_label {Point{20,children.point(0).y},"age 0-14"};
children.set_color(Color::red);
children_label.set_color(Color::red);

Text adults_label {Point{20,adults.point(0).y},"age 15-64"};
adults.set_color(Color::blue);
adults_label.set_color(Color::blue);

Text aged_label {Point{20,aged.point(0).y},"age 65+"};
aged.set_color(Color::dark_green);
aged_label.set_color(Color::dark_green);
```

В заключение нам нужно связать разные объекты класса `Shape` с объектом класса `Window` и передать управление системе графического пользовательского интерфейса (см. раздел 14.2.3).

```
win.attach(children);
win.attach(adults);
win.attach(aged);

win.attach(children_label);
win.attach(adults_label);
win.attach(aged_label);

win.attach(x);
win.attach(y);
win.attach(current_year);

gui_main();
```

Весь код можно было бы поместить в функцию `main()`, но мы предпочитаем использовать вспомогательные классы `Scale` и `Distribution`, а также оператор ввода, определенный в классе `Distribution`.

Если вы забыли, что мы делаем, посмотрите на рисунок еще раз.



Задание

Задание, связанное с построением графиков.

1. Создайте пустое окно 600×600 с меткой "Графики функций".
2. Обратите внимание, что вам надо создать проект, свойства которого указаны на веб-сайте в примечании к руководству по инсталляции библиотеки `FLTK`.
3. Поместите файлы `Graph.cpp` и `Window.cpp` в свой проект.

4. Добавьте оси *x* и *y* длиной по 400 пикселей с метками "1 == 20 пикселей" и делениями через каждые 20 пикселей. Оси должны пересекаться в точке (300,300).
5. Сделайте обе оси красными.

В дальнейшем используйте отдельный объект класса `Shape` для построения каждой из перечисленных ниже функций.

1. Постройте график функции `double one(double x){return 1;}` в диапазоне [–10,11] с началом координат (0,0) в точке (300,300), используя 400 точек и не выполняя масштабирования (в окне).
2. Измените рисунок, применив масштабирование с коэффициентом 20 как по оси *x*, так и по оси *y*.
3. В дальнейшем используйте этот диапазон и коэффициенты масштабирования при построении всех графиков.
4. Добавьте график функции `double slope(double x){return x/2;}`.
5. Пометьте наклонную линию с помощью объекта класса `Text` со значением `"x/2"` в точке, расположенной прямо над левым нижним углом окна.
6. Добавьте график функции `double square(double x){return x*x;}`.
7. Добавьте график косинуса (не создавая новую функцию).
8. Сделайте график косинуса синим.
9. Напишите функцию `sloping_cos()`, суммирующую косинус и функцию `slope()` (как определено выше) и добавьте в окно ее график.

Задание, связанное с определением класса.

1. Определите класс `struct Person`, содержащий член `name` типа `string` и член `age` типа `int`.
2. Определите переменную класса `Person`, инициализируйте ее значением "Goofy" и 63 и выведите на экран (`cout`).
3. Определите оператор ввода (`>>`) и вывода (`<<`) для класса `Person`; считайте объект класса `Person` с клавиатуры (`cin`) и выведите его на экран (`cout`).
4. Напишите конструктор класса `Person`, инициализирующий члены `name` и `age`.
5. Сделайте представление класса `Person` закрытым и добавьте константные функции-члены `name()` и `age()`, предназначенные для чтения имени и возраста.
6. Модифицируйте операторы `>>` и `<<` для заново определенного класса `Person`.

7. Модифицируйте конструктор, чтобы он выполнял проверку, что переменная `age` лежит в диапазоне [0,150), а переменная `name` не содержит символы `; : " ' [ ] * & ^ % $ # @ !`. В случае ошибки используйте функцию `error()`. Протестируйте программу.
8. Считайте последовательность объектов класса `Person` с устройства ввода (`cin`) в вектор типа `vector<Person>`; выведите его на экран (`cout`). Проверьте правильность ввода.
9. Измените представление класса `Person` так, чтобы вместо члена `name` использовались члены `first_name` и `second_name`. Отсутствие при вводе обоих этих членов должно считаться ошибкой. Измените операторы `>>` и `<<`. Протестируйте программу.

Контрольные вопросы

1. Что такое функция одного аргумента?
2. Когда для представления данных используется непрерывная линия, а когда дискретные точки?
3. Какая функция определяет наклон? Напишите ее математическую формулу.
4. Что такое парабола?
5. Как создать ось x? Как создать ось y?
6. Что такое аргумент, заданный по умолчанию, и зачем он нужен?
7. Как составить сложную функцию?
8. Как при построении графиков используются цвет и метки?
9. Что представляет собой приближение функции с помощью ряда?
10. Зачем разрабатывать эскиз графика перед разработкой кода для его построения?
11. Как масштабировать график?
12. Как масштабировать входные данные без многократных попыток и ошибок?
13. Зачем форматировать входные данные? Не лучше ли рассматривать файл, просто заполненный числами?
14. Как вы разрабатываете общий эскиз графика? Как этот эскиз отражается в вашей программе?

Термины

аппроксимация
аргумент по умолчанию
лямбда-выражение
масштабирование
функция
эскиз экрана

Упражнения

1. Рассмотрим еще один способ определения функции, вычисляющей факториал.

   ```
   int fac(int n) { return n>1 ? n*fac(n-1) : 1; } // n!
   ```

   При вычислении, например, `fac(4)`, эта функция сначала выясняет, что, поскольку `4>1`, ответ равен `4*fac(3)`, что, очевидно, равно `4*3*fac(2)` или, рассуждая аналогично, `4*3*2*fac(1)`, т.е. `4*3*2*1`. Попытайтесь понять, как работает эта функция. Функция, вызывающая саму себя, называется *рекурсивной* (recursive). Альтернативная реализация, описанная в разделе 15.5, называется *итеративная* (iterative), потому что в ней используется итерация по значениям (в цикле `while`). Убедитесь, что рекурсивная функция `fac()` работает и выдает те же результаты, что и итеративная функция `fac()` при вычислении факториала чисел 0, 1, 2, 3, 4 и так далее до 20. Какую реализацию функции `fac()` вы предпочитаете и почему?

2. Определите класс `Fct`, который почти совпадает с классом `Function`, за исключением того, что он хранит аргументы, переданные конструктору. Включите в класс `Fct` операции "восстановления" параметров, чтобы мы могли повторять вычисления с разными диапазонами, функциями и т.д.

3. Модифицируйте класс `Fct` из предыдущего упражнения так, чтобы в нем был дополнительный аргумент, позволяющий контролировать точность. Сделайте тип этого аргумента шаблонным параметром, чтобы повысить гибкость класса.

4. Постройте графики функций `sin()`, `cos()`, `sin(x)+cos(x)` и `sin(x)**sin(x)+cos(x)*cos(x)` на одном рисунке. Изобразите оси и метки.

5. "Анимируйте" (как в разделе 15.5) ряд `1-1/3+1/5-1/7+1/9-1/11+ ...` Он называется рядом Лейбница (Leibniz) и сходится к числу $\pi/4$.

6. Разработайте и реализуйте класс для построения гистограммы. Его исходные данные должны храниться в векторе типа `vector<double>`, в котором записаны N значений и каждое значение представляется "столбиком", т.е. прямоугольником с соответствующей высотой.

7. Разработайте класс для построения гистограмм, позволяющий расставлять метки на рисунке в целом и на отдельных столбиках в частности. Предусмотрите использование цвета.

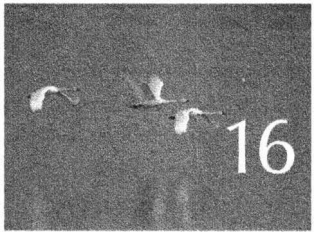
8. Ниже перечислено множество пар, составленных из роста и количества людей указанного роста (с точностью до пяти сантиметров): (170,7), (175,9), (180,23), (185,17), (190,6), (195,1). Как изобразить эти данные? Если вы не нашли лучшего решения, постройте гистограмму.

Помните об осях и метках. Запишите данные в файл и считайте их оттуда.

9. Найдите другой набор данных о росте людей и изобразите их с помощью программы, созданной при выполнении предыдущего упражнения. Например, найдите в вебе распределение роста людей в США или попросите своих друзей измерить свой рост. В идеале вы не должны изменять свою программу, чтобы приспособить ее к новому набору данных. Для этого следует применить масштабирование данных. Считывание меток также позволит минимизировать количество изменений, если вы захотите повторно использовать программу.
10. Какие виды данных неудобно представлять с помощью графиков или гистограмм? Найдите пример и опишите способ представления таких данных (например, в виде коллекции помеченных точек).
11. Найдите среднюю температуру для каждого месяца в нескольких городах (например, Кембридж, Англия и Кембридж, Массачусетс; в мире есть много городов с названием "Кембридж") и изобразите их на одном рисунке. Как всегда, помните об осях, метках, цвете и т.д.

Послесловие

Графическое представление данных очень важно. Мы лучше воспринимаем хорошо нарисованный график, чем совокупность чисел, на основе которых он построен. Когда нужно построить график, большинство людей используют чужой код — подходящую библиотеку. Как устроены такие библиотеки и что делать, если их нет под рукой? На каких идеях основаны "простые графические инструменты"? Теперь вы знаете: это не магия и не нейрохирургия. Мы рассмотрели только двумерные изображения; трехмерные графические изображения также весьма полезны в науке, технике, маркетинге и даже еще более интересны, чем двумерные. Займитесь ими как-нибудь!



16 Графические пользовательские интерфейсы

Вычисления — это уже не только компьютеры. Это образ жизни.
— Николас Негропонте (Nicholas Negroponte)

Графический пользовательский интерфейс (graphical user interface — GUI) позволяет пользователю взаимодействовать с программой, щелкая на кнопках, выбирая пункты меню, вводя данные разными способами и отображая текстовые и графические элементы на экране. Именно этим мы пользуемся во время работы со своими компьютерами и веб-сайтами. В данной главе излагаются основы написания кода, определяющего приложение с графическим пользовательским интерфейсом и управляющего им. В частности, мы покажем, как написать программу, взаимодействующую с элементами экрана с помощью функций обратного вызова. Возможности нашего графического пользовательского интерфейса "надстроены" над средствами системы. Низкоуровневые средства и интерфейсы описаны в приложении Д, где используются инструменты и методы, рассмотренные в главах 17 и 18. Здесь мы сосредоточимся лишь на их использовании.

16.1. Альтернативы пользовательского интерфейса

Каждая программа имеет интерфейс пользователя. Программы, работающие на небольшом устройстве, как правило, ограничиваются вводом данных с помощью щелчка на кнопках, а для вывода используют мигающую подсветку. Другие компьютеры соединены с внешним миром только проводами. В этой главе мы рассмотрим наиболее распространенный случай, когда наша программа взаимодействует с пользователем, который смотрит на экран и пользуется клавиатурой и манипулятором (например, мышью). В этом случае у программиста есть три возможности.

- *Использовать для ввода и вывода консоль.* Это хороший выбор для технической/профессиональной работы, когда ввод имеет простую текстовую форму, а данные несложные (например, имена файлов или числа). Если вывод является текстовым, его можно вывести на экран или записать в файл. Для решения такой задачи удобно использовать потоки `iostream` из стандартной библиотеки C++ (см. главы 10 и 11). Если же результаты необходимо вывести в графическом виде, можно использовать графическую библиотеку (см. главы 12–15), не изменяя существенным образом свой стиль программирования.
- *Использовать библиотеку графического пользовательского интерфейса.* Именно это мы делаем, когда хотим, чтобы взаимодействие пользователя с программой осуществлялось с помощью манипуляций объектами на экране (указание мышью, щелчки, перетаскивание и опускание, наведение указателя и т.д.). Часто (но не всегда) этот стиль связан с интенсивным отображением графической информации на экране. Каждый пользователь современных компьютеров может привести такие примеры. Любой программист, желающий соответствовать "духу" приложений Windows/Mac, должен использовать взаимодействие в стиле графического пользовательского интерфейса.

- *Использовать интерфейс веб-браузера.* В этом случае потребуются язык разметки (markup language), такой как HTML, а также язык сценариев (scripting language). Эта тема выходит за рамки нашей книги, но для приложений с удаленным доступом именно такой выбор часто оказывается самым удачным. В этом случае взаимодействие программы с экраном носит текстовый характер (с использованием потоков символов). Браузер представляет собой приложение графического пользовательского интерфейса, которое переводит текст в графические элементы, транслирует щелчки мышью и другие действия пользователя в текстовые данные и отправляет их обратно программе.

Многие люди считают использование графического пользовательского интерфейса сущностью современного программирования, а взаимодействие с объектами на экране — его основной целью. Мы с этим не согласны: графический пользовательский интерфейс — это разновидность ввода-вывода, а отделение основной логики приложения от системы ввода-вывода является одним из идеалов программного обеспечения. При любой возможности мы предпочитаем провести четкую границу между основной логикой программы и той ее частью, которая осуществляет ввод и вывод. Такое отделение позволяет изменять способ взаимодействия с пользователем, переносить программу на другие операционные системы и, что еще боле важно, размышлять о логике программы и способах ее взаимодействия с пользователем независимо друг от друга.

Тем не менее графический пользовательский интерфейс важен и интересен в разных аспектах. В данной главе исследуются как способы интеграции графических элементов в наши приложения, так и способы защиты основных принципов создания интерфейса от влияния субъективных вкусов.

16.2. Кнопка Next

Как мы предоставляем пользователю кнопку Next, которая использовалась для управления графическими примерами в главах 12–15? В этих примерах графика в окне выводилась после щелчка на ней. Очевидно, что это простая форма программирования графического пользовательского интерфейса. Фактически она настолько проста, что некоторые могут сказать, что это не настоящий графический пользовательский интерфейс. Однако посмотрим, как это было сделано, поскольку это приведет нас прямо к той разновидности программирования, которую все признают программированием графического пользовательского интерфейса.

Наш код в главах 12–15 был устроен примерно так:

```
// Создаем объекты и/или манипулируем ими,
// выводим их в окно win
win.wait_for_button();

// Создаем объекты и/или манипулируем ими,
// выводим их в окно win
win.wait_for_button();

// Создаем объекты и/или манипулируем ими,
// выводим их в окно win
win.wait_for_button();
```

Каждый раз, достигая вызова функции `wait_for_button()`, мы могли видеть наши объекты на экране, пока не выполнялся щелчок на кнопке, чтобы получить результаты работы другой части программы. С точки зрения программной логики этот код ничем не отличается от программы, записывающей строки текста на экране (в окне консоли), а затем останавливающейся и ожидающей ввода данных с клавиатуры:

```
// Определяем переменные и/или вычисляем значения,
// генерируем вывод программы
cin >> var; // Ожидаем ввод

// Определяем переменные и/или вычисляем значения,
// генерируем вывод программы
cin >> var; // Ожидаем ввод

// Определяем переменные и/или вычисляем значения,
// генерируем вывод программы
cin >> var; // Ожидаем ввод
```

С точки зрения реализации эти два вида программ совершенно отличны один от другого. Когда программа выполняет инструкцию `cin >> var`, она останавливается и ждет, пока "система" не вернет символы, которые ввел пользователь. Однако система (графический пользовательский интерфейс), управляющая экраном и отслеживающая вашу работу с мышью, следует другой модели: она определяет, где находится указатель мыши и что пользователь с нею делает (щелкает и т.д.). Если ваша программа ожидает каких-то действий, то она должна делать следующее.

- Указать интерфейсу, за чем надо следить (например, за событием "кто-то щелкнул на кнопке Next").
- Указать, что делать, когда произошло ожидаемое событие.
- Ожидать, пока графический пользовательский интерфейс обнаружит интересующее программу действие.

Новый интересный аспект заключается в том, что система графического пользовательского интерфейса не просто возвращает управление вашей программе; она разрабатывается так, чтобы по-разному реагировать на

разные действия пользователя, такие как щелчок мышью на одной из многих кнопок, изменение размера окна, перерисовка окна после закрытия вложенного окна и открытие выпадающих меню.

Для начала мы просто хотим сказать "пожалуйста, сообщите, когда кто-то щелкнет на кнопке"; иначе говоря, "пожалуйста, продолжите выполнение моей программы, когда кто-то щелкнет кнопкой мыши в тот момент, когда ее указатель будет находиться в прямоугольной области, представляющей собой изображение моей кнопки". Это простейшее действие, которое можно себе представить. Однако эта операция не предоставляется "системой" — ее необходимо написать самому. Как это сделать — первый вопрос, который мы рассмотрим, приступая к изучению программирования графического пользовательского интерфейса.

16.3. Простое окно

В основном "система" (т.е. комбинация библиотеки графического пользовательского интерфейса и операционной системы) непрерывно отслеживает положение указателя мыши и состояние ее кнопок. Программа может проявить интерес к определенной области экрана и попросить систему вызвать функцию, когда произойдет что-нибудь "интересное". В частности, мы можем попросить систему вызвать одну из наших функций обратного вызова (callback functions), когда пользователь щелкнет на кнопке. Для этого необходимо сделать следующее.

- Определить кнопку.
- Отобразить ее на экране.
- Определить функцию, которую должен вызвать графический пользовательский интерфейс.
- Сообщить графическому пользовательскому интерфейсу о данной кнопке и функции.
- Подождать, когда графический пользовательский интерфейс вызовет нашу функцию.

Давайте сделаем это. Кнопка — это часть объекта класса **Window**, поэтому (в файле **Simple_window.h**) мы определяем класс **Simple_window**, содержащий член **next_button**.

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);

    void wait_for_button(); // Простой цикл событий
private:
    Button next_button;     // Кнопка Next
    bool button_pushed;     // Деталь реализации
```

```
    static void cb_next(Address, Address); // Обратный вызов
                                           // для кнопки next_button
    void next();                           // Действие, выполняемое после
                                           // щелчка на кнопке next_button
};
```

Очевидно, что класс **Simple_window** является производным от класса **Window** из библиотеки **Graph_lib**. Все наши окна должны быть объектами класса, непосредственно или косвенно порожденными от класса **Graph_lib::Window**, поскольку именно этот класс (с помощью библиотеки FLTK) связывает наше понятие окна с его системной реализацией. Детали реализации класса **Window** описаны в разделе Д.3.

Наша кнопка инициализируется в конструкторе класса **Simple_window**.

```
Simple_window::Simple_window(Point xy, int w, int h, const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-70,0}, 70, 20, "Next", cb_next},
    button_pushed{false}
{
    attach(next_button);
}
```

Нет ничего удивительного в том, что класс **Simple_window** передает положение своего объекта (**xy**), размер (**w**, **h**) и заголовок (**title**) классу **Window** из библиотеки **Graph_lib** для дальнейшей обработки. Далее конструктор инициализирует член **next_button** координатами (**Point(x_max()-70,0)**; это где-то в области верхнего правого угла), размером (70, 20), меткой (**"Next"**) и функцией обратного вызова (**cb_next**). Первые четыре параметра совпадают с параметрами, которые мы использовали при описании класса **Window**: мы задаем положение прямоугольника на экране и указываем его метку.

Наконец мы вызываем функцию **attach()** и связываем член **next_button** с классом **Simple_window**; иначе говоря, сообщаем окну, что оно должно отобразить кнопку в указанном месте и сделать так, чтобы графический пользовательский интерфейс узнал о ней.

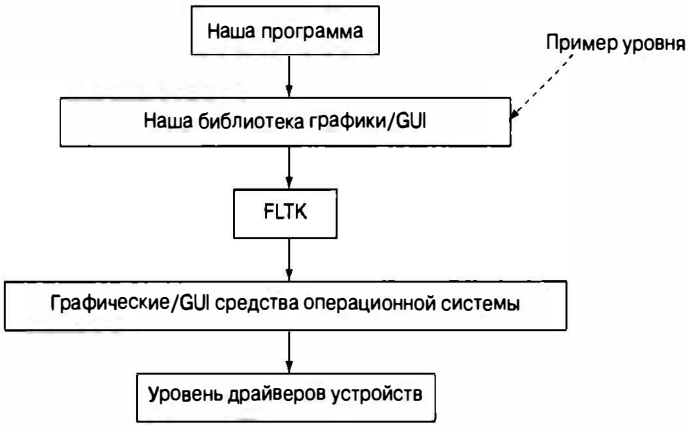
Член **button_pushed** — это несколько запутывающая дело деталь реализации; мы используем его для того, чтобы отслеживать щелчки на кнопке после последнего выполнения функции **next()**. Фактически здесь все является деталями реализации, а следовательно, должно быть объявлено как **private**. Игнорируя детали реализации, мы получим

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);
    void wait_for_button(); // Простой цикл событий
    // ...
};
```

Другими словами, пользователь может создать окно и ожидать, пока произойдет щелчок на кнопке.

16.3.1. Функции обратного вызова

Функция `cb_next()` — новая и интересная деталь. Именно она должна быть вызвана системой графического пользовательского интерфейса, когда будет зарегистрирован щелчок на кнопке. Поскольку мы передаем такие функции системе графического пользовательского интерфейса, чтобы та вызвала их для нас, их часто называют *функциями обратного вызова* (callback function). Этот факт отображается в префиксе функции `cb_next()` (`cb_` — "callback"). Такое имя выбирается просто для того, чтобы мы помнили о предназначении этой функции, — ни язык, ни библиотека этого не требуют. Очевидно, что мы выбрали имя `cb_next` потому, что эта функция должна быть вызвана для кнопки Next. Определение функции `cb_next` выглядит уродливым куском "шаблона". Перед демонстрацией ее кода посмотрим, что происходит.



Наша программа работает поверх нескольких "уровней" кода. Она использует нашу библиотеку графики, которую мы реализовали с помощью библиотеки FLTK, которая, в свою очередь, реализована на основе возможностей операционной системы. В системе есть еще больше уровней и подуровней. Каким-то образом щелчок мышью, идентифицированный драйвером мыши, становится причиной вызова функции `cb_next()`. Мы передаем адрес функции `cb_next()` и адрес нашего объекта класса `Simple_window` вниз через уровни программного обеспечения; затем какой-то код "где-то внизу" вызывает функцию `cb_next()`, когда выполняется щелчок на кнопке Next.

Система графического пользовательского интерфейса (и операционная система) может использоваться программами, написанными на разных языках, поэтому они не могут навязывать всем пользователям стиль языка C++. В частности, ей ничего не известно о наших классах `Simple_window`

и **Button**. Фактически она вообще ничего не знает о классах и их членах. Тип, требуемый для обратного вызова функции, выбирается так, чтобы его можно было использовать на самых низких уровнях программирования, включая язык C и ассемблер. Функция обратного вызова не возвращает значений и принимает в качестве аргументов два адреса. Мы можем объявить функцию-член так, чтобы она подчинялась этим требованиям:

```
static void cb_next(Address, Address); // Обратный вызов
                                       // для next_button
```

Здесь ключевое слово **static** гарантирует, что функция **cb_next()** может быть вызвана как обычная функция, т.е. не как функция-член, вызываемая через конкретный объект. Если бы функцию-член могла вызывать сама операционная система, было бы намного лучше. Однако интерфейс обратного вызова должен использоваться программами, написанными на разных языках, поэтому мы используем статическую функцию-член. Аргументы **Address** указывают, что функция **cb_next()** получает аргументы, адресующие "что-то в памяти". Ссылки, существующие в языке C++, во многих языках неизвестны, поэтому мы не можем их использовать. Компилятор не знает, какие типы имеют эти аргументы, расположенные "где-то". Здесь мы снижаемся на уровень аппаратного обеспечения и не можем получать привычную помощь от языка программирования. Система вызовет функцию обратного вызова, первый аргумент которой должен представлять собой адрес некоторого элемента графического пользовательского интерфейса (объекта класса **Widget**), для которого был сделан обратный вызов. Мы не хотим использовать этот первый аргумент, поэтому и не именуем его. Второй аргумент — это адрес окна, содержащего данный объект класса **Widget**; для функции **cb_next()** аргументом является объект класса **Simple_window**. Эту информацию можно использовать следующим образом:

```
void Simple_window::cb_next(Address, Address pw)
// Вызов Simple_window::next() для окна по адресу pw
{
    reference_to<Simple_window>(pw).next();
}
```

Вызов функции **reference_to<Simple_window>(pw)** сообщает компьютеру, что адрес, хранящийся в переменной **pw**, должен рассматриваться в качестве адреса объекта класса **Simple_window**; иначе говоря, мы можем использовать значение **reference_to<Simple_window>(pw)** как ссылку на объект класса **Simple_window**. В главах 17 и 18 мы еще вернемся к вопросам адресации памяти. Определение функции **reference_to** (кстати, совершенно тривиальное) мы покажем в разделе Д.1. А пока мы просто рады наконец-то получить привычную ссылку на наш объект класса **Simple_window** и обеспечить непосредственный доступ к нашим данным и функциям,

которые собирались использовать, и поскорее выбираемся из этого системно-зависимого кода, вызывая нашу функцию-член `next()`.

Мы могли внести в функцию `cb_next()` весь код, который должен быть выполнен, но мы, как и большинство хороших программистов, разрабатывающих графические пользовательские интерфейсы, предпочитаем отделять запутанный низкоуровневый код от нашего красивого пользовательского кода, а потому обрабатываем обратный вызов с помощью двух функций.

- Функция `cb_next()` превращает системные соглашения об обратных вызовах в вызов обычной функции-члена `next()`.
- Функция `next()` делает то, что мы хотели (ничего не зная о запутанном механизме обратного вызова).

Мы используем здесь две функции, руководствуясь общим принципом, гласящим, что "функция должна выполнять отдельное логическое действие": функция `cb_next()` скрывает низкоуровневую системно-зависимую часть программы, а функция `next()` выполняет требуемое действие. В ситуациях, когда необходим обратный вызов (из "системы") в одном из окон, мы всегда определяем пару таких функций; например, см. разделы 16.5–16.7. Перед тем как идти дальше, повторим сказанное.

- Мы определяем класс `Simple_window`.
- Конструктор класса `Simple_window` регистрирует кнопку `next_button` в системе графического пользовательского интерфейса.
- Когда пользователь щелкает на изображении объекта `next_button` на экране, графический пользовательский интерфейс вызывает функцию `cb_next()`.
- Функция `cb_next()` преобразует низкоуровневую системную информацию в вызов функции-члена `next()` нашего окна.
- После щелчка на кнопке функция `next()` выполняет требуемое действие.

Это довольно сложный способ вызвать функцию. Однако помните, что мы работаем с базовым механизмом, обеспечивающим взаимодействие мыши (или другого аппаратного устройства) с программой. В частности, следует иметь в виду следующие обстоятельства.

- Как правило, на компьютере одновременно выполняется много программ.
- Программа создается намного позже операционной системы.
- Программа создается намного позже библиотеки графического пользовательского интерфейса.

- Программа может быть написана на языке, отличающемся от того, который используется в операционной системе.
- Описанный метод охватывает все виды взаимодействий (а не только щелчок на кнопке).
- Окно может иметь много кнопок, а программа может иметь много окон.

Однако, разобравшись, как вызывается функция **next()**, мы поймем, как обрабатывается каждое действие в программе с графическим пользовательским интерфейсом.

16.3.2. Цикл ожидания

Итак, что же должна делать функция **next()** класса **Simple_window** после каждого щелчка на кнопке в данном (простейшем) случае? В принципе, мы хотели бы, чтобы эта операция останавливала выполнение нашей программы в некоторой точке, давая возможность увидеть, что было сделано к этому моменту. Кроме того, мы хотим, чтобы функция **next()** возобновляла работу нашей программы после паузы.

```
// Создаем некоторые объекты и/или манипулируем ими,
// выводим их в окне

win.wait_for_button(); // Работа программы продолжается с этой точки

// Создаем некоторые объекты и/или манипулируем ими
```

На самом деле это просто. Сначала определим функцию **wait_for_button()**.

```
void Simple_window::wait_for_button()
    // Модифицированный цикл событий:
    // обрабатываем все события (по умолчанию)
    // и выходим из цикла, когда переменная
    // button_pushed принимает значение true.
{
    while (!button_pushed) Fl::wait();
    button_pushed = false;
    Fl::redraw();
}
```

Как и большинство систем графического интерфейса, библиотека FLTK содержит функцию, приостанавливающую работу программы, пока не произойдет какое-то событие. Версия этой функции в библиотеке FLTK называется **wait()**. На самом деле функция **wait()** делает много полезного, чтобы наша программа могла правильно продолжить работу, когда произойдет ожидаемое событие. Например, при работе под управлением операционной системы Microsoft Windows программа должна

перерисовать окно, которое было перемещено или ранее перекрыто другим окном. Кроме того, объект класса **Window** должен обрабатывать изменение размеров окна. Функция **Fl::wait()** выполняет все эти действия так, как это предусмотрено по умолчанию. Каждый раз, когда функция **wait()** обрабатывает какое-то событие, она возвращает управление, чтобы наша программа могла выполнить какие-то свои действия.

Таким образом, когда кто-то щелкает на кнопке Next, функция **wait()** вызывает функцию **cb_next()** и возвращает управление (нашему циклу ожидания). Для обработки события в функции **wait_for_button()** функция **next()** должна просто установить значение булевой переменной **button_pushed** равным **true**. Это просто:

```
void Simple_window::next()
{
    button_pushed = true;
}
```

Конечно же, мы должны где-то определить эту переменную **button_pushed**.

```
bool button_pushed; // Инициализируется в конструкторе
                    // значением false
```

После ожидания функция **wait_for_button()** должна восстановить прежнее значение переменной **button_pushed** и вызвать функцию **redraw()**, чтобы все внесенные изменения отобразились на экране. Именно так мы и поступили.

16.3.3. Лямбда-выражения в качестве функций обратного вызова

Итак, для каждого действия в классе **Widget** мы определяем две функции: одну — для отображения системного понятия функции обратного вызова, а вторую — для выполнения необходимых действий:

```
struct Simple_window : Graph_lib::Window {
    Simple_window(Point xy, int w, int h, const string& title);
    void wait_for_button(); // Простой цикл событий
private:
    Button next_button;     // Кнопка "Next"
    bool button_pushed;     // Деталь реализации
    static void cb_next(Address, Address); // Функция обратного
                                           // вызова для next_button
    void next();            // Действия при нажатии next_button
};
```

Воспользовавшись лямбда-выражениями (§15.3.3), мы можем обойтись без явного объявления отображающей функции **cb_next()**. Вместо нее мы определяем отображение в конструкторе **Simple_window**:

```
Simple_window::Simple_window(Point xy, int w, int h,
                             const string& title)
```

```
    :Window{xy,w,h,title},
     next_button{Point{x_max()-70,0}, 70, 20, "Next",
        [ ] (Address, Address pw)
            {
                reference_to<Simple_window>(pw).next();
            }
        },
    button_pushed{false}
    {
        attach(next_button);
    }
```

16.4. Класс Button и другие разновидности Widget

Определим класс, описывающий кнопку.

```
struct Button : Widget {
    Button(Point xy, int w, int h, const string& label,
           Callback cb);
    void attach(Window&);
};
```

Класс **Button** является производным от класса **Widget** с координатами **xy**, размерами **w** и **h**, текстовой меткой **label** и функцией обратного вызова **cb**. В принципе, все, что появляется на экране в результате какого-то действия (например, обратного вызова), связано с объектом класса **Widget**.

16.4.1. Класс Widget

Виджет (widget) — это технический термин. У него есть более информативный, но менее эффектный синоним — *управляющий элемент* (control). Такой управляющий элемент используется для определения разновидностей взаимодействия с программой через графический пользовательский интерфейс. Интерфейсный класс **Widget** имеет следующий вид:

```
class Widget {
    // Класс Widget - это дескриптор для класса Fl_widget,
    // но он не является классом Fl_widget;
    // мы стараемся, чтобы наши интерфейсные классы
    // отличались от FLTK
public:
    Widget(Point xy, int w, int h, const string& s,
           Callback cb);

    virtual void move(int dx,int dy);
    virtual void hide();
    virtual void show();
    virtual void attach(Window&) = 0;
```

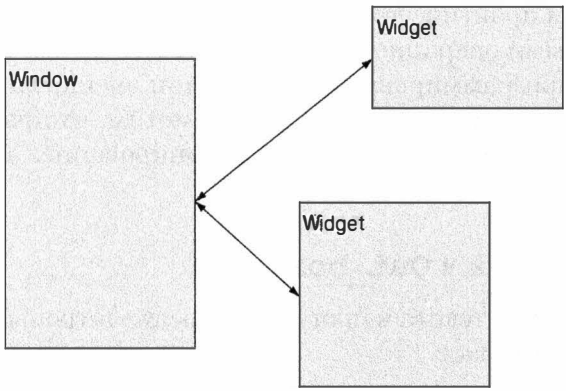
```
    Point loc;
    int width;
    int height;
    string label;
    Callback do_it;

protected:
    Window* own;     // Каждый объект класса Widget
                     // принадлежит некоторому Window
    Fl_Widget* pw;   // Связь с Widget из библиотеки FLTK
};
```

Класс **Widget** имеет две интересные функции, которые можно применить в классе **Button** (а также в любом другом классе, производном от класса **Widget**, например **Menu**; см. раздел 16.7).

- Функция **hide()** делает объект класса **Widget** невидимым.
- Функция **show()** делает объект класса **Widget** снова видимым.

Изначально объект класса **Widget** является видимым.

Как и в случае класса **Shape**, мы можем с помощью функции **move()** перемещать объект класса **Widget** в окне, а перед тем как его использовать, должны связать этот объект с окном, вызвав функцию **attach()**. Обратите внимание на то, что функция **attach()** объявлена как чисто виртуальная (см. раздел 14.3.5): каждый класс, производный от класса **Widget**, должен самостоятельно определить, что означает его связывание с объектом класса **Window**. Фактически в функции **attach()** создаются системные управляющие элементы. Функция **attach()** вызывается из объекта класса **Window** как часть реализации его собственной функции **attach()**. В принципе, связывание окна и элемента управления — это очень тонкое дело, в котором каждая из сторон выполняет свое задание. В результате окно знает о существовании своих управляющих элементов, а каждый управляющий элемент знает о своем окне.



Обратите внимание, что объект класса **Window** не знает о том, с какой разновидностью класса **Widget** он имеет дело. Как описано в разделе 14.4, объектно-ориентированное программирование гарантирует объектам класса **Window** возможность взаимодействовать с любыми разновидностями класса **Widget**. Аналогично классу **Widget** также неизвестно, с какой разновидностью класса **Window** он работает.

Мы проявили небольшую неаккуратность, оставив открытыми данные-члены. Члены **own** и **pw** предназначены исключительно для реализации производных классов, поэтому мы объявили их как **protected**.

Определения класса **Widget** и его конкретных разновидностей (**Button**, **Menu** и т.д.) содержатся в файле **GUI.h**.

16.4.2. Класс Button

Класс **Button** — это простейший класс **Widget**, с которым мы работаем. Все, что он делает, — это обратный вызов после щелчка на кнопке.

```
class Button : public Widget {
public:
    Button(Point xy, int ww, int hh, const string& s,
           Callback cb)
        :Widget{xy,ww,hh,s,cb} {}

    void attach(Window& win);
};
```

Только и всего. Функция **attach()** содержит (относительно) грязный код библиотеки FLTK. Мы отложили объяснения до приложения Д (пожалуйста, не читайте его, не усвоив главы 17 и 18). А пока заметим, что определение простого подкласса **Widget** не представляет особого труда.

Мы не касаемся довольно сложного и запутанного вопроса, связанного с внешним видом кнопки (и других управляющих элементов) на экране. Проблема в том, что выбор внешнего вида элементов управления окном практически бесконечен, причем некоторые стили диктуются конкретными операционными системами. Кроме того, с точки зрения технологии программирования в описании внешнего вида кнопок нет ничего нового. Если вы расстроились, то заметим, что размещение фигуры поверх кнопки не влияет на ее функционирование, а как нарисовать фигуру, вы уже знаете.

16.4.3. Классы In_box и Out_box

Для ввода и вывода текста в программе предусмотрены два класса, производные от класса **Widget**.

```
struct In_box : Widget {
    In_box(Point xy, int w, int h, const string& s)
```

```
        :Widget{xy,w,h,s,0} { }
    int get_int();
    string get_string();

     void attach(Window& win);
};

struct Out_box : Widget {
    Out_box(Point xy, int w, int h, const string& s)
        :Widget{xy,w,h,s,0} { }
    void put(int);
    void put(const string&);

    void attach(Window& win);
};
```

Объект класса `In_box` может принимать текст, введенный в него, а прочесть этот текст можно с помощью функции `get_string()` как строку или как целое число — с помощью функции `get_int()`. Если вы хотите убедиться, что текст был введен, прочитайте его с помощью функции `get_string()` и убедитесь, что она возвращает непустую строку.

```
string s = some_inbox.get_string();
if (s =="") {
    // Текст не был введен
}
```

Объект класса `Out_box` используется для выдачи сообщений, адресованных пользователю. По аналогии с классом `In_box` мы можем с помощью функции `put()` ввести в него либо целые числа, либо строки. Примеры использования классов `In_box` и `Out_box` приведены в разделе 16.5.

Мы могли бы предусмотреть функции `get_floating_point()`, `get_complex()` и так далее, но не сделали этого, поскольку вы можете поместить полученную строку в поток `stringstream` и форматировать ввод так, как захотите (см. раздел 11.4).

16.4.4. Класс Menu

Мы предлагаем очень простое меню.

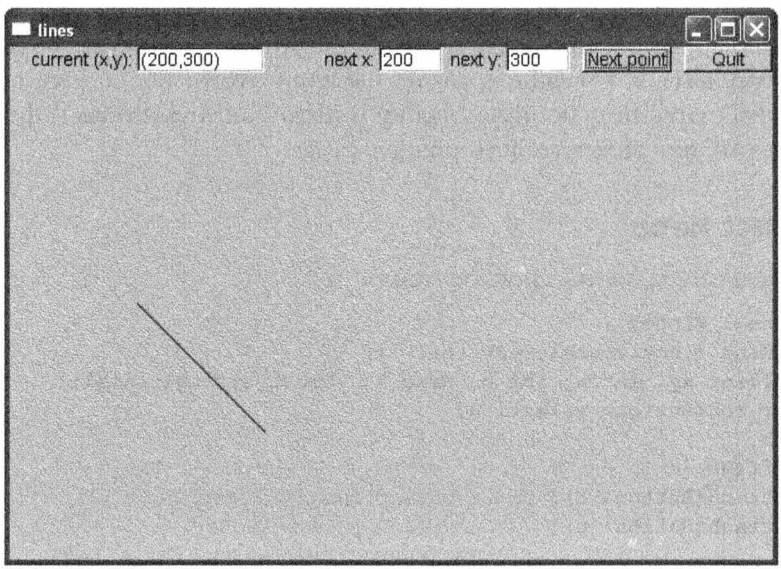
```
struct Menu : Widget {
    enum Kind { horizontal, vertical };
    Menu(Point xy, int w, int h, Kind kk, const string& label);
    Vector_ref<Button> selection;
    Kind k;
    int offset;
    int attach(Button& b); // Связывает кнопку с меню
    int attach(Button* p); // Связывает с меню кнопку,
                           // созданную с помощью оператора new
```

```
    void show()             // Показывает все кнопки
    {
        for(Button& b: selection) b.show();
    }
    void hide();                // Скрывает все кнопки
    void move(int dx, int dy); // Перемещает все кнопки

    void attach(Window& win);  // Связывает все кнопки с win
};
```

По существу, объект класса **Menu** представляет собой вектор кнопок. Как обычно, **Point xy** задает координаты левого верхнего угла. Ширина и высота используются для изменения размера кнопки при ее добавлении в меню. Примеры можно найти в разделах 16.5 и 16.7. Каждая кнопка меню (пункт меню) — это независимый объект класса **Widget**, переданный объекту класса **Menu** в качестве аргумента функции **attach()**. В свою очередь, класс **Menu** содержит функцию **attach()**, связывающую все свои кнопки с окном. Объект класса **Menu** отслеживает все свои кнопки с помощью класса **Vector_ref** (разделы 13.10 и Д.4). Если хотите создать всплывающее меню (pop-up menu), то можете сделать это самостоятельно (подробнее об этом рассказывается в разделе 16.7).

16.5. Пример

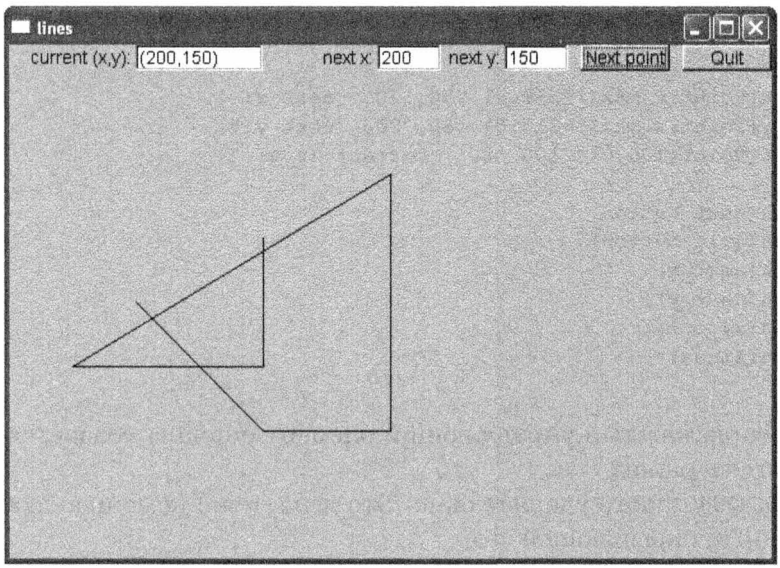
Для того чтобы лучше ознакомиться с основными возможностями графического пользовательского интерфейса, рассмотрим окно простого приложения, в котором есть ввод, вывод и немного графики.



Эта программа позволяет пользователю вывести последовательность отрезков (незамкнутая ломаная; см. раздел 13.6), задаваемую последовательностью пар координат. Идея заключается в том, что пользователь многократно вводит координаты (*x*, *y*) в поля ввода next x и next y; после ввода очередной пары пользователь щелкает на кнопке Next point.

Изначально поле ввода current (x, y) пустое, а программа ожидает, пока пользователь введет первую пару координат. После этого введенная пара координат появляется в поле ввода current (x, y), а ввод каждой новой пары координат приводит к появлению на экране нового отрезка, идущего от текущей точки (координаты которой отображаются в поле ввода current (x, y)) до только что введенной пары (*x*, *y*), а сама точка (*x*, *y*) становится новой текущей точкой.

Так рисуется незамкнутая ломаная. Когда пользователь устанет, он щелкнет на кнопке Quit для завершения работы. Логика программы очень проста, а сама программа использует несколько полезных средств графического пользовательского интерфейса: ввод и вывод текста, рисование линий и несколько кнопок. Окно, показанное выше, демонстрирует результат после ввода двух пар координат. После семи шагов на экране получается рисунок наподобие следующего.



Определим класс для рисования таких окон. Он довольно прост.

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h,
                 const string& title );
    Open_polyline lines;
private:
    Button next_button;  // Добавляет пару (next_x,next_y) в объект lines
```

```
    Button quit_button;
    In_box next_x;
    In_box next_y;
    Out_box xy_out;

    void next();
    void quit();
};
```

Ломаная линия представлена объектом класса `Open_polyline`. Кнопки и поля ввода-вывода объявлены как объекты классов `Button`, `In_box` и `Out_box`, и для каждой кнопки предусмотрена функция-член, реализующая связанное с ней действие. Мы решили воспользоваться лямбда-выражениями, чтобы избежать написания лишних функций обратного вызова.

Конструктор класса `Lines_window` инициализирует все его члены.

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-150,0}, 70, 20, "Next point",
        [ ] (Address, Address pw){
            reference_to<Lines_window>(pw).next();
        },
    quit_button{Point{x_max()-70,0}, 70, 20, "Quit",
        [ ] (Address, Address pw){
            reference_to<Lines_window>(pw).quit();
        },
    next_x{Point{x_max()-310,0}, 50, 20, "next x:"},
    next_y{Point{x_max()-210,0}, 50, 20, "next y:"},
    xy_out{Point{100,0}, 100, 20, "current (x,y):"}
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);
}
```

Иначе говоря, каждый управляющий элемент сначала создается, а потом связывается с окном.

Кнопка Quit просто удаляет окно. Это делается с помощью любопытной идиомы FLTK, скрывающей его.

```
void Lines_window::quit()
{
    hide(); // Идиома FLTK для удаления окна
}
```

Вся реальная работа выполняется кнопкой Next point: она читает пару координат, обновляет объект `Open_polyline`, текущую точку и перерисовывает окно.

```
void Lines_window::next()
{
    int x = next_x.get_int();
    int y = next_y.get_int();
    lines.add(Point{x,y});

    // Обновление текущей точки
    ostringstream ss;
    ss << '(' << x << ',' << y << ')';
    xy_out.put(ss.str());

    redraw();
}
```

Все совершенно очевидно. Функция `get_int()` позволяет получить целочисленные координаты из объектов класса `In_box`; поток `ostringstream` форматирует строки для вывода в объект класса `Out_box`; функция-член `str()` позволяет получить строку из потока `ostringstream`. Последняя функция, `redraw()`, необходима для представления результатов пользователю; старое изображение остается на экране, пока не будет вызвана функция `redraw()` из класса `Window`.

Что же нового и необычного в этой программе? Рассмотрим ее функцию `main()`.

```
#include "GUI.h"

int main()
try {
    Lines_window win{Point{100,100},600,400,"lines"};
    return gui_main();
}
catch(exception& e) {
    cerr << "Исключение: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Какое-то исключение\n";
    return 2;
}
```

Но ведь здесь, по сути, ничего нет! Тело функции `main()` содержит лишь определение нашего окна `win` и вызов функции `gui_main()`. Ни других функций, ни операторов `if` или `switch`, ни цикла — ничего из того, что мы видели в исходных текстах в главах 6 и 7, — только определение переменной и вызов функции `gui_main()`, которая сама является вызовом функции `run()` из библиотеки FLTK. Изучая программу далее, мы видим, что функция `run()` представляет собой просто бесконечный цикл

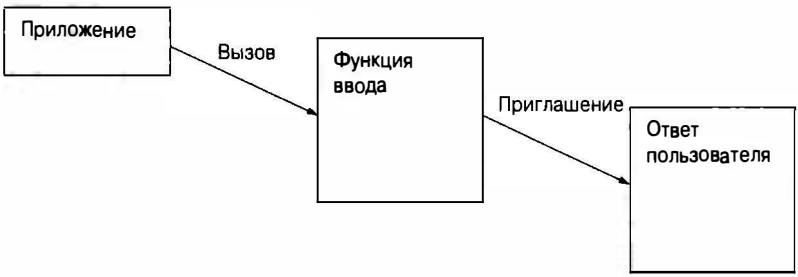
```
while(wait());
```

За исключением некоторых деталей реализации, описание которых вынесено в приложение Д, мы просмотрели весь код, запускающий программу рисования линий. Мы увидели всю логику этой программы. Что же произошло?

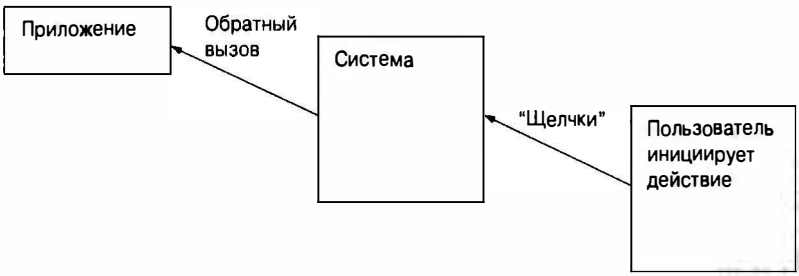
16.6. Инверсия управления

А произошло вот что: мы передали управление порядком от самой программы виджетам: теперь программа возобновляет свою работу каждый раз, когда пользователь активирует какой-нибудь из них. Например, щелкните на кнопке, и заработает функция обратного вызова. После возврата из нее программа "замирает", ожидая, пока пользователь сделает что-нибудь еще. По существу, функция `wait()` просит систему следить за виджетами и выполнять соответствующие обратные вызовы. Теоретически функция `wait()` могла бы сообщать программисту, какой виджет требует внимания, и предоставить вызов соответствующей функции самому программисту. Однако в библиотеке FLTK и в большинстве других систем графического пользовательского интерфейса функция `wait()` просто выполняет соответствующий обратный вызов, освобождая программиста от необходимости писать код для выбора этой функции.

"Обычная" программа организована следующим образом.



Программа графического пользовательского интерфейса организована иначе.



Одна из сложностей такой "инверсии управления" проявляется в том, что порядок выполнения программы теперь полностью определяется действиями пользователя. Это усложняет как организацию, так и отладку программы. Трудно себе представить, что сделает пользователь, но еще труднее представить себе возможные результаты случайной последовательности обратных вызовов. Это превращает систематическое тестирование в ночной кошмар (подробнее об этом — в главе 26). Соответствующие методы выходят за рамки рассмотрения нашей книги, но мы просим читателей быть особенно осторожными, работая с кодом, управляемым пользователями с помощью обратных вызовов. Кроме очевидных проблем с потоком управления, существуют проблемы, связанные с видимостью и отслеживанием связей между элементами управления окном и данными. Для того чтобы минимизировать трудности, очень важно поддерживать часть программы, отвечающей за графический пользовательский интерфейс, как можно более простой, и создавать ее постепенно, тестируя на каждом шаге. Работая с такой программой, почти всегда необходимо рисовать небольшие диаграммы объектов и их взаимодействия между собой.

Как взаимодействуют части программы, активируемые разными обратными вызовами? Простейший способ такого взаимодействия заключается в том, чтобы функции оперировали данными, хранящимися в окне, как в примере из раздела 16.5. В нем функция **next()** класса **Lines_window**, вызываемая щелчком на кнопке Next point, считывала данные из объектов класса **In_box** (**next_x** и **next_y**), а затем обновляла переменную-член **lines** и объект класса **Out_box** (**xy_out**). Очевидно, что функция, запущенная обратным вызовом, может делать все что угодно: открывать файлы, связываться с вебом и т.д. Однако пока что мы рассмотрим простой случай, когда данные хранятся в окне.

16.7. Добавление меню

Рассмотрим вопросы управления и взаимодействия, возникающие при "инверсии управления", добавляя меню в нашу программу рисования ломаной линии. Для начала просто предоставим меню, позволяющее пользователю выбирать цвет всех отрезков в переменной-члене **lines**. Добавим меню **color_menu** и его обратные вызовы.

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);

    Open_polyline lines;
    Menu color_menu;

    // Обратный вызов для красной кнопки:
```

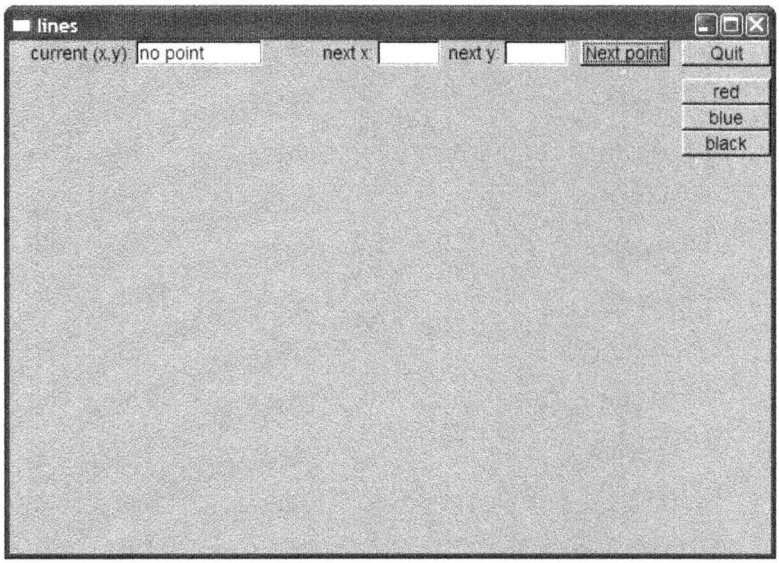
```
    static void cb_red(Address, Address);
    // Обратный вызов для синей кнопки:
    static void cb_blue(Address, Address);
    // Обратный вызов для черной кнопки:
    static void cb_black(Address, Address);

    // Действия:
    void red_pressed()   { change(Color::red);   }
    void blue_pressed()  { change(Color::blue);  }
    void black_pressed() { change(Color::black); }

    void change(Color c) { lines.set_color(c);   }

    // ... все, как и раньше ...
};
```

Создание всех таких практически идентичных функций обратного вызова и функций "действия" — довольно утомительное занятие. Однако оно не вызывает никаких глобальных затруднений, а описание более простых в смысле вводимого текста средств выходит за рамки нашей книги. Если вы хотите, то можете избавиться от функций обратного вызова **cb_** с помощью лямбда-выражений. После щелчка на кнопке меню цвет линий изменяется на требуемый.

Определив член **color_menu**, мы должны его инициализировать.

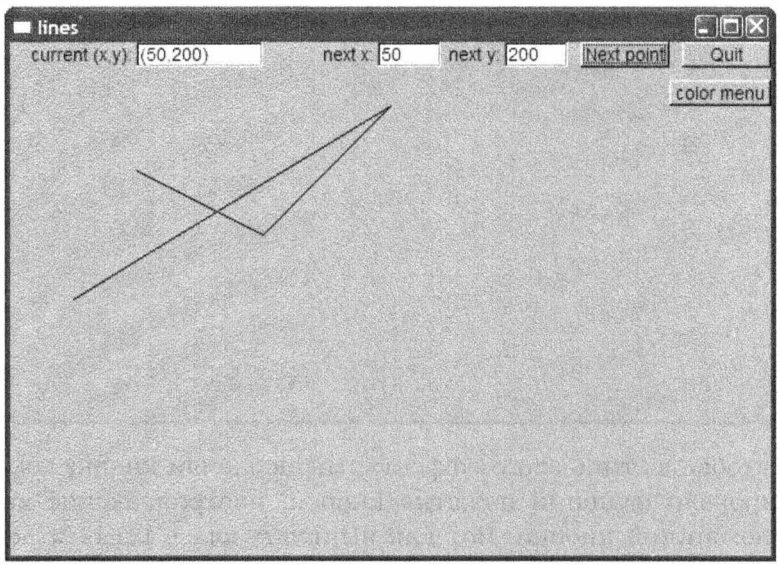
```
Lines_window::Lines_window(Point xy, int w, int h,
                           const string&title)
    :Window{xy,w,h,title},
    // ... Как и раньше ...
    color_menu{Point{x_max()-70,40},70,20,
               Menu::vertical,"color"}
{
     // ... Как и раньше ...
     color_menu.attach(new Button{Point{0,0},0,0,
                                  "red",cb_red});
     color_menu.attach(new Button{Point{0,0},0,0,
                                  "blue",cb_blue});
     color_menu.attach(new Button{Point{0,0},0,0,
                                  "black",cb_black});
     attach(color_menu);
}
```

Кнопки создаются динамически, связываются с меню (с помощью функции **attach()**) и при необходимости могут быть удалены и/или изменены. Функция **Menu::attach()** настраивает размер и местоположение кнопки, а также связывает ее с окном. Это все. Теперь мы увидим на экране следующее.

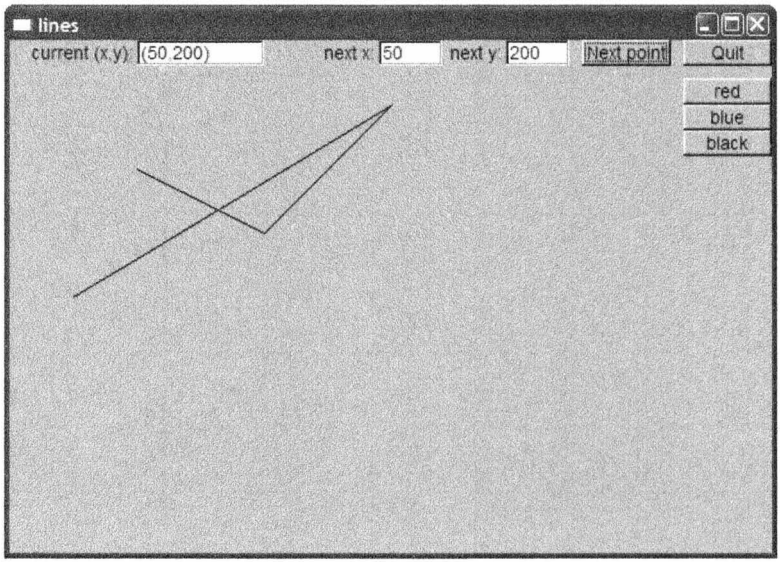


Поэкспериментировав с этой программой, мы решили, что нам необходимо выпадающее меню, т.е. мы не хотим, чтобы на экране находились все кнопки меню, за исключением момента, когда мы собираемся им воспользоваться. Мы решили добавить кнопку color menu. Когда пользователь щелкает на ней, появляется меню выбора цвета, которое скрывается после того, как цвет выбран, а на экране снова появляется кнопка color menu.

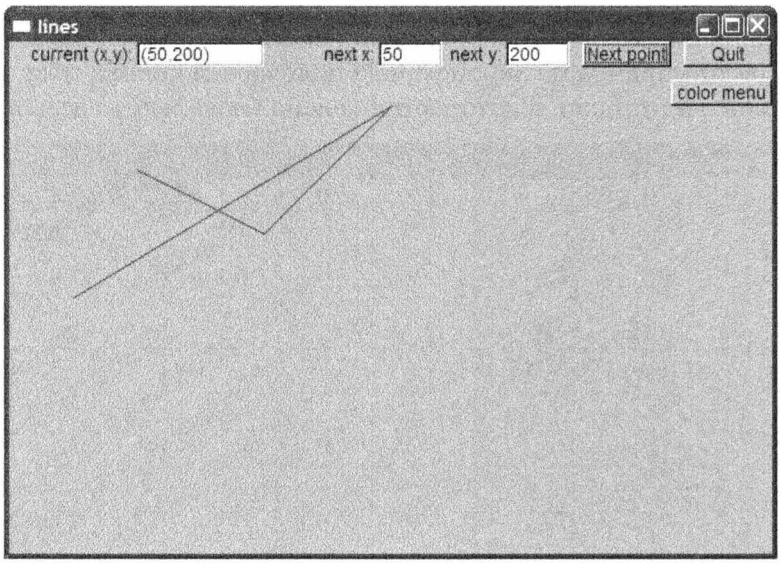
Вот как выглядит окно, в которое добавлено несколько отрезков.



В окне мы видим новую кнопку color menu и несколько черных линий. После щелчка на кнопке color menu на экране появится меню выбора цвета.



Обратите внимание, что кнопка color menu исчезла. Она не нужна, пока открыто меню. Щелкнем на кнопке blue, и цвет ломаной линии изменится на синий, а кнопка color menu вновь появится на экране.



Для того чтобы достичь этого эффекта, мы добавили кнопку color menu и модифицировали функции нажатия кнопок, настраивающие видимость меню и упомянутой кнопки. Вот как выглядит класс `Lines_window` после всех этих модификаций.

```
struct Lines_window : Window {
    Lines_window(Point xy, int w, int h, const string& title);
```

```
private:
    // Данные:
    Open_polyline lines;

    // Виджеты:
    Button next_button; // Добавляет (next_x,next_y) к lines
    Button quit_button; // Завершает работу программы
    In_box next_x;
    In_box next_y;
    Out_box xy_out;
    Menu color_menu;
    Button menu_button;

    void change(Color c) { lines.set_color(c); }

    void hide_menu() { color_menu.hide(); menu_button.show(); }

    // Действия, инициируемые обратными вызовами:
    void red_pressed()  {change(Color::red); hide_menu(); }
    void blue_pressed() {change(Color::blue); hide_menu(); }
    void black_pressed(){change(Color::black); hide_menu(); }
    void menu_pressed() {menu_button.hide(); color_menu.show();}
    void next();
    void quit();

    // Функции обратного вызова:
    static void cb_red(Address, Address);
    static void cb_blue(Address, Address);
    static void cb_black(Address, Address);
    static void cb_menu(Address, Address);
    static void cb_next(Address, Address);
    static void cb_quit(Address, Address);
};
```

Обратите внимание на то, что все члены, кроме конструктора, являются закрытыми. Вообще говоря, этот класс и является программой. Все, что происходит, происходит с помощью обратных вызовов, поэтому никакого кода, кроме этого класса, не требуется. Мы упорядочили объявления, чтобы определение класса стало более удобочитаемым. Конструктор передает аргументы всем своим подобъектам и связывает их с окном.

```
Lines_window::Lines_window(Point xy, int w, int h,
                           const string& title)
    :Window{xy,w,h,title},
    next_button{Point{x_max()-150,0}, 70, 20,
                "Next point", cb_next},
    quit_button{Point{x_max()-70,0}, 70, 20,
                "Quit", cb_quit},
    next_x{Point{x_max()-310,0}, 50, 20, "next x:"},
    next_y{Point{x_max()-210,0}, 50, 20, "next y:"},
    xy_out{Point{100,0}, 100, 20, "current (x,y):"},
```

```
    color_menu{Point{x_max()-70,30},70,20,
               Menu::vertical,"color"},
    menu_button{Point{x_max()-80,30}, 80, 20,
                "color menu", cb_menu}
{
    attach(next_button);
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    xy_out.put("no point");
    color_menu.attach(new Button{Point{0,0},0,0, "red",cb_red});
    color_menu.attach(new Button{Point{0,0},0,0, "blue",cb_blue});
    color_menu.attach(new Button{Point{0,0},0,0, "black",cb_black});
    attach(color_menu);
    color_menu.hide();
    attach(menu_button);
    attach(lines);
}
```

Обратите внимание на то, что инициализация выполняется в порядке определения данных-членов. Это правильный порядок инициализации.

Фактически инициализация членов всегда выполняется в порядке их объявления. Некоторые компиляторы (любезно) выдают предупреждения, если конструктор базового класса или члена указан вне этого порядка.

16.8. Отладка GUI-программы

После того как программа с графическим пользовательским интерфейсом начнет работать, ее отладка будет довольно простой: что видите, то и получаете. Однако часто имеется трудный период перед тем, как в окне начнут появляться первые фигуры и виджеты или даже само окно появится на экране. Протестируем следующую функцию `main()`.

```
int main()
{
    Lines_window{Point{100,100},600,400,"lines"};
    return gui_main();
}
```

Вы видите ошибку? Независимо от того, видите вы ее или нет, эту программу следует испытать; она компилируется и выполняется, но вместо линий

на экране в лучшем случае появляется какое-то мерцание. Как найти ошибку в такой программе? Для этого можно сделать следующее.

- Тщательно исследовать части программы (классы, функции, библиотеки).

- Упростить весь новый код, понемногу увеличивая объем программы, начиная с простейшей версии и тщательно отслеживая строку за строкой.
- Проверить все установки редактора связей.
- Сравнить ее исходный текст с кодом уже работающей программы.
- Объяснить код другу.

Среди всех этих предложений самым трудным является отслеживание выполнения кода. Если вы умеете работать с отладчиком программ, у вас есть шанс, но простая вставка операторов вывода в данном случае бесполезна — проблема заключается в том, что на экране не появляется никакой вывод. Даже отладчики иногда испытывают проблемы, поскольку в компьютере несколько действий выполняется одновременно (многопоточность), так как ваша программа — не единственная программа, пытающаяся взаимодействовать с экраном. Ключом является упрощение кода и систематическое его исследование.

И все же, в чем проблема? Вот правильная версия (см. раздел 16.5).

```
int main()
{
    Lines_window win{Point{100,100},600,400,"lines"};
    return gui_main();
}
```

Мы "забыли" указать имя `win` объекта класса `Lines_window`. Поскольку на самом деле мы не используем это имя, это кажется разумным, но компилятор решит, что, поскольку вы не используете окно, его можно сразу удалить. В результате окно существует всего несколько миллисекунд. Не удивительно, что мы его не заметили.

Другая распространенная проблема заключается в том, что окно располагается *точно* поверх другого окна. Результат выглядит так, будто на экране открыто только одно окно. А куда делось другое? Мы можем долго искать несуществующие ошибки в своей программе. Та же проблема может возникнуть при размещении одной фигуры поверх другой.

И в заключение (чтобы еще больше огорчить читателей) отметим, что при работе с библиотеками графического пользовательского интерфейса исключения не всегда работают так, как мы ожидаем. Поскольку наша программа управляется библиотекой графического пользовательского интерфейса, сгенерированное исключение может никогда не попасть к своему обработчику — библиотека или операционная система может "съесть" его (используя механизмы обработки ошибок, отличающиеся от исключений языка C++).

К типичным проблемам, выявляемым при отладке, относится и отсутствие изображений объектов **Shape** и **Widget** из-за отсутствия подключения к окну или некорректного поведения объектов, вышедших из области видимости. Рассмотрим, как программист может создать и связать кнопку с меню, породив проблемы.

```
// Вспомогательная функция для загрузки кнопки в меню
void load_disaster_menu(Menu& m)
{
    Point orig{0,0};
    Button b1{orig,0,0,"flood",cb_flood};
    Button b2{orig,0,0,"fire",cb_fire};
    // ...
    m.attach(b1);
    m.attach(b2);
    // ...
}

int main()
{
    // ...
    Menu disasters{Point{100,100},60,20,Menu::horizontal,
                   "disasters"};
    load_disaster_menu(disasters);
    win.attach(disasters);
    // ...
}
```

Этот код не работает. Все кнопки являются локальными для функции **load_disaster_menu**, и их связывание с меню — большая ошибка. Объяснение этого факта приведено в разделе 18.6.4 (*не возвращайте указатель на локальную переменную!*), а схема размещения локальных переменных в памяти была проиллюстрирована в разделе 8.5.8. Дело в том, что после возврата управления из функции **load_disaster_menu()** эти локальные объекты уничтожаются, и меню **disasters** ссылается на несуществующие (уничтоженные) объекты. Результат, скорее всего, будет неожиданным и точно неприятным. Решение заключается в использовании вместо именованных локальных объектов неименованных объектов, созданных с помощью оператора **new**.

```
// Вспомогательная функция для загрузки кнопки в меню
void load_disaster_menu(Menu& m)
{
    Point orig{0,0};
    m.attach(new Button{orig,0,0,"flood",cb_flood});
    m.attach(new Button{orig,0,0,"fire",cb_fire});
    // ...
}
```

Правильное решение даже проще, чем ошибочный (и, увы, широко распространенный) код.

Задание

1. Создайте совершенно новый проект с настройками редактора связей для работы с библиотекой FLTK (описаны в приложении Г.)
2. Используя возможности `Graph_lib`, введите текст программы из раздела 16.5 и заставьте ее работать.
3. Модифицируйте программу так, чтобы она использовала всплывающее меню, описанное в разделе 16.7, и добейтесь ее работы.
4. Измените программу так, чтобы в ней было второе меню для выбора стиля линий, и добейтесь ее корректной работы.

Контрольные вопросы

1. Зачем нужен графический пользовательский интерфейс?
2. Когда нужен текстовый интерфейс?
3. Что такое уровень программного обеспечения?
4. Зачем нужны уровни программного обеспечения?
5. В чем заключается фундаментальная проблема взаимодействия с операционной системой с помощью языка C++?
6. Что такое обратный вызов?
7. Что такое виджет?
8. Как еще называют виджет?
9. Что означает аббревиатура "FLTK"?
10. О каком еще инструментарии GUI вы знаете?
11. Какие системы используют термин *виджет*, а какие предпочитают термин *управляющий элемент*?
12. Приведите примеры виджетов.
13. Когда используются окна для ввода текста?
14. Какие типы данных могут храниться в окнах ввода?
15. Когда используется кнопка?
16. Когда используется меню?
17. Что такое инверсия управления?
18. Опишите основную стратегию отладки программ с графическим пользовательским интерфейсом.

19. Почему отладка программ с графическим пользовательским интерфейсом труднее, чем отладка обычной программы с потоками ввода-вывода?

Термины

виджет	кнопка	пользовательский интерфейс
видимый/невидимый	консольный ввод-вывод	управляющий элемент
графический пользовательский интерфейс	меню	уровень программного обеспечения
диалоговое окно	обратный вызов	функция обратного вызова
инверсия управления	ожидание ввода	цикл ожидания

Упражнения

1. Создайте класс **My_window**, аналогичный классу **Simple_window**, за исключением того, что он имеет две кнопки, **next** и **quit**.
2. Создайте окно (на основе класса **My_window**) с шахматной доской 4×4 из квадратных кнопок. После щелчка на кнопке должно выполняться простое действие, например вывод ее координат в окно вывода или изменение ее цвета (пока не будет выполнен щелчок на другой кнопке).
3. Разместите объект класса **Image** поверх объекта класса **Button**; после щелчка на кнопке переместите оба объекта. Для выбора нового местоположения "кнопки с изображением" используйте следующий генератор случайных чисел из **std_lib_facilities.h**:

```
#include<random>
inline int rand_int(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>{min,max}(ran);
}
```

   Эта функция возвращает случайное целое число в диапазоне [**min**,**max**).
4. Создайте меню с пунктами "окружность", "квадрат", "равносторонний треугольник" и "шестиугольник". Создайте окно ввода (или два таких окна) для ввода пар координат и разместите фигуру, созданную после щелчка на соответствующей кнопке, в заданной точке.
5. Напишите программу, рисующую фигуру по вашему выбору и перемещающую ее в новую точку после щелчка на кнопке Next. Новая точка должна выбираться на основе пары координат, считанной из потока ввода.
6. Создайте "аналоговые часы", т.е. часы с двигающимися стрелками. Определите время, используя средства операционной системы. Основная часть этого упражнения — найти функции, определяющие

время суток и прекращающие выполнение программы на короткий период времени (например, на секунду), а также научиться использовать их по имеющейся документации. *Указание*: `clock()`, `sleep()`.

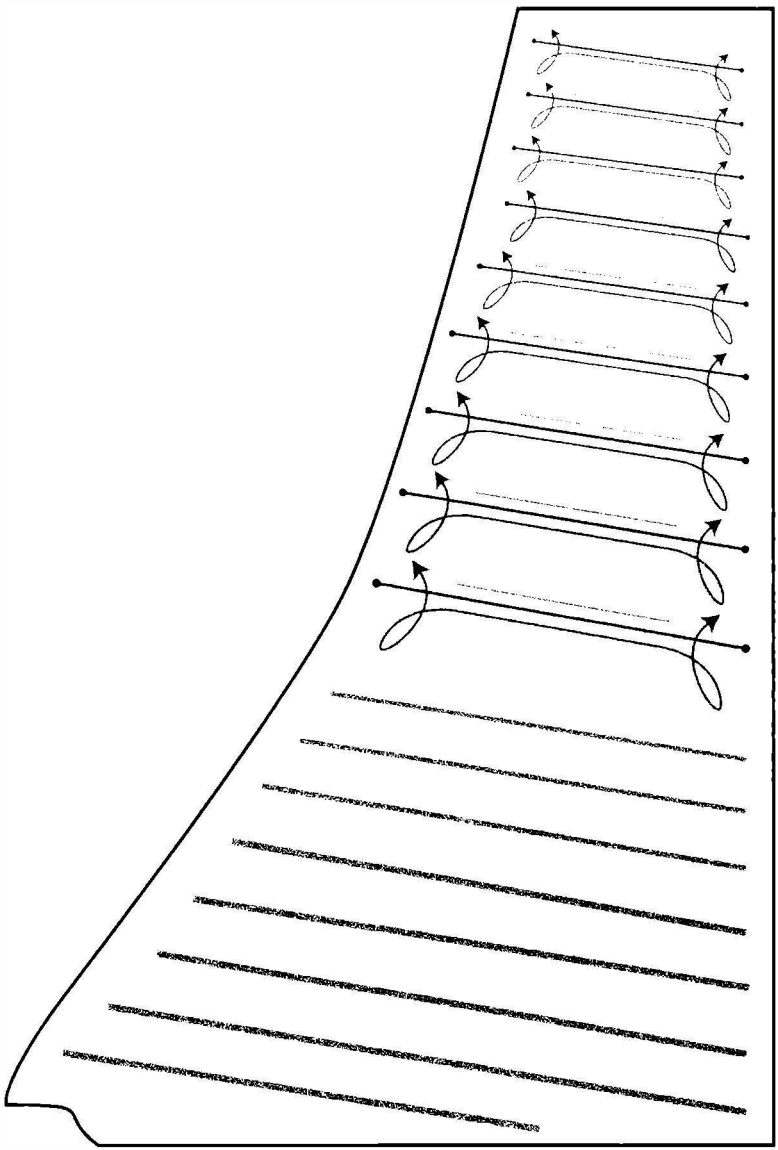
7. Используя приемы из предыдущих упражнений, создайте изображение самолета, летящего по экрану. Добавьте кнопки Start и Stop.
8. Создайте конвертер валют. Считывайте курсы валют из файла в момент запуска программы. Вводите сумму в окно ввода и предусмотрите возможность выбора валют (например, пару меню).
9. Модифицируйте калькулятор из главы 7 так, чтобы выражение вводилось в окне ввода, а результат возвращался в окне вывода.
10. Разработайте программу, в которой можно выбрать функции из большого набора функций (например, `sin()` и `log()`), ввести для них параметры и построить их графики.

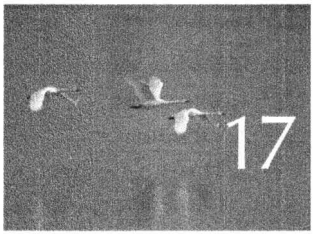
Послесловие

Графический пользовательский интерфейс — неисчерпаемая тема. Большая часть этой темы связана со стилем и совместимостью с существующими системами. Кроме того, много сложностей возникает при работе с чрезвычайно разнообразными виджетами (так, библиотека GUI может предлагать многие десятки альтернативных стилей кнопок). Однако лишь немногие вопросы из этой области относятся к фундаментальным методам программирования, поэтому мы не будем углубляться в этом направлении. Другие темы, такие как масштабирование, вращение, трансформация, трехмерные объекты и так далее, требуют изложения сложных графических и/или математических вопросов, затрагивать которые здесь мы не намерены.

Вы должны знать о том, что большинство систем GUI предоставляют программу-редактор, позволяющую визуально создавать окна, назначать обратные вызовы и задавать действия кнопок, меню и т.д. Для многих приложений такие редакторы позволяют существенно сократить процесс программирования, например, обратных вызовов. Однако всегда следует понимать, как будет работать получающаяся в результате программа. Иногда сгенерированный код эквивалентен тому, что вы видели в главе; в других случаях в коде используются более сложные и/или дорогостоящие механизмы.

Часть III
Данные и алгоритмы





17

Векторы и динамически выделяемая память

По умолчанию используйте `vector`*!*
— Алекс Степанов (Alex Stepanov)

В этой и четырех следующих главах описываются контейнеры и алгоритмы из стандартной библиотеки языка C++, которую обычно называют STL. Мы рассматриваем основные возможности библиотеки STL и описываем их применение. Кроме того, здесь представлены ключевые методы проектирования и программирования, использованные при реализации библиотеки STL, а также некоторые примененные для этого низкоуровневые возможности языка. Среди последних — указатели, массивы и динамически распределяемая память. В центре внимания этой и следующих двух глав находятся проектирование и реализация наиболее популярного и полезного контейнера библиотеки STL: **`vector`**.

17.1. Введение

Наиболее полезным контейнером в стандартной библиотеке языка C++ является класс **vector**. Вектор обеспечивает хранение последовательности элементов заданного типа. Мы можем обращаться к элементу вектора по индексу, расширять вектор с помощью функции **push_back()**, запрашивать у вектора количество его элементов, используя функцию **size()**, а также контролируемо обращаться к элементам вектора, предотвращая выход за пределы допустимого диапазона. Стандартный вектор — удобный, гибкий, эффективный (как в смысле времени работы, так и в смысле используемой памяти) и безопасный с точки зрения статических типов контейнер. Стандартный класс **string** обладает подобными свойствами, как и другие типы стандартных контейнеров, такие как **list** и **map**, которые будут рассматриваться в главе 20. Однако память компьютера не обеспечивает непосредственной поддержки таких полезных типов. Все, что способно поддерживать аппаратное обеспечение *непосредственно*, — это последовательности байтов. Например, в классе **vector<double>** операция **v.push_back(2.3)** добавляет значение **2.3** в последовательность чисел типа **double** и увеличивает на единицу количество элементов вектора **v** (**v.size()**). На самом нижнем уровне компьютер ничего не знает о таких сложных функциях, как **push_back()**; все, что он знает, — как прочитать и записать несколько байтов за раз.

В этой и следующих двух главах мы покажем, как построить класс **vector**, используя основные языковые возможности, доступные любому

программисту. Это сделано для того, чтобы проиллюстрировать полезные концепции и методы программирования и показать, как их можно выразить с помощью средств языка C++. Языковые возможности и методы программирования, использованные при реализации класса `vector`, весьма полезны и очень широко используются.

Разобравшись в вопросах проектирования, реализации и использования класса `vector`, мы сможем понять устройство других стандартных контейнеров, таких как `map`, и испытать элегантные и эффективные методы их использования, обеспечиваемые стандартной библиотекой языка C++ (подробнее об этом речь пойдет в главах 20 и 21). Эти методы, называемые алгоритмами, избавляют нас от необходимости самостоятельно программировать типичные задачи обработки данных. Вместо самостоятельной разработки мы можем воспользоваться тем, что доступно нам как часть каждой реализации языка программирования C++, упрощая написание и тестирование наших библиотек. Мы уже видели и использовали один из наиболее полезных алгоритмов стандартной библиотеки — `sort()`.

Мы будем приближаться к стандартному библиотечному классу `vector` через ряд постепенно усложняющихся вариантов реализации. Сначала мы создадим очень простой класс `vector`. Затем выявим его недостатки и исправим их. Поступив так несколько раз, мы придем к реализации класса `vector`, который является грубым эквивалентом стандартного библиотечного класса `vector`, поставляемого вместе с компиляторами языка C++. Этот процесс постепенного уточнения точно отражает обычный подход к решению программистской задачи. Попутно мы выявим и исследуем многие классические задачи, связанные с использованием памяти и структур данных. Наш основной план приведен ниже.

- *Глава 17.* Как работать с разными объемами памяти? В частности, как создавать разные векторы с разным количеством элементов и как отдельный вектор может иметь разное количество элементов в разные моменты времени? Это приведет нас к проверке объема динамически распределяемой памяти (кучи), указателям, приведению типов (операторам явного приведения типов) и ссылкам.
- *Глава 18.* Как копировать вектор? Как реализовать оператор доступа к элементам по индексу? Кроме того, мы рассмотрим массивы и исследуем их связь с указателями.
- *Глава 19.* Как создавать векторы с разными типами хранящихся в них элементов? Как обрабатывать ошибку выхода за пределы допустимого диапазона? Для ответа на этот вопрос мы изучим шаблоны языка C++ и исключения.

Кроме новых свойств языка и методов программирования, которые мы вводим для создания гибкого, эффективного и безопасного с точки зрения типов вектора, мы будем использовать (в том числе повторно) многое из

описанного ранее. В некоторых случаях мы получим возможность изложить уже рассмотренные вопросы более формально и технично.

Итак, все упирается в прямой доступ к памяти. Зачем он нам нужен? Наши классы **vector** и **string** чрезвычайно полезны и удобны; их легко использовать. В конце концов, контейнеры, такие как **vector** и **string**, разработаны именно для того, чтобы освободить нас от неприятных аспектов работы с реальной памятью. Однако, если только мы не верим в волшебство, мы должны освоить самый низкий уровень управления памятью. Но почему бы нам не поверить в волшебство, т.е. почему бы не поверить, что разработчики класса **vector** знали, что делают? В конце концов, мы же не разбираем физические устройства, обеспечивающие работу памяти компьютера.

Дело в том, что все мы — программисты (специалисты по компьютерным наукам, разработчики программного обеспечения и т.д.), а не физики. Если бы мы изучали физику, то были бы обязаны разбираться в деталях устройства и функционирования памяти компьютера. Но поскольку мы изучаем программирование, то должны вникать в детали устройства программ. С теоретической точки зрения мы могли бы рассматривать низкоуровневый доступ к памяти и средства управления памятью как "детали реализации", так же, как мы рассматриваем физические устройства. Однако в этом случае мы не только вынуждены были бы "верить в волшебство", но и не могли бы разрабатывать новые контейнеры (которые нужны только нам и которых нет в стандартной библиотеке). Кроме того, мы не могли бы разобраться в огромном количестве программного кода, написанного на языках C и C++, который непосредственно использует память. Как будет показано в следующих главах, указатели (низкоуровневый и непосредственный способ обращения к объектам) полезны не только для управления памятью. Невозможно полностью овладеть языком C++, не зная, как работают указатели.

Говоря более философски, я отношусь к большой группе профессионалов в области компьютерных наук, считающих, что отсутствие теоретических знаний и практического понимания работы с памятью порождает проблемы при рассмотрении высокоуровневых тем, таких как структуры данных, алгоритмы и операционные системы.

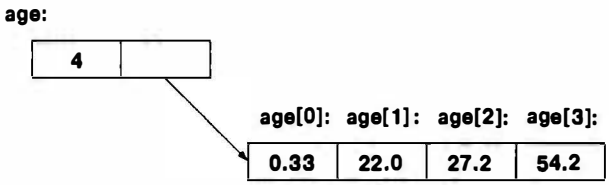
17.2. Основы

Начнем нашу постепенную разработку класса **vector** с очень простого примера его использования.

```
vector<double> age(4); // Вектор с четырьмя элементами
age[0]=0.33;           // типа double
age[1]=22.0;
```

```
age[2]=27.2;
age[3]=54.2;
```

Очевидно, что этот код создает объект класса **vector** с четырьмя элементами типа **double** и присваивает им значения **0.33**, **22.0**, **27.2** и **54.2**. Эти четыре элемента имеют номера 0, 1, 2 и 3. Нумерация элементов в стандартных контейнерах языка C++ всегда начинается с нуля. Нумерация с нуля очень распространена и является универсальным соглашением, которого придерживаются все программисты, пишущие программы на языке C++. Количество элементов в объекте класса **vector** называется его размером. Итак, размер вектора **age** равен четырем. Элементы вектора нумеруются (индексируются) от **0** до **size-1**. Например, элементы вектора **age** нумеруются от **0** до **age.size()-1**. Графически вектор **age** можно изобразить следующим образом.

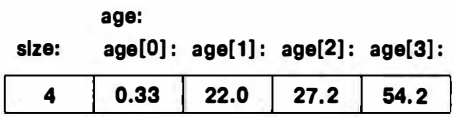


Как реализовать этот "графический проект" в компьютерной памяти? Как хранить значения и обеспечивать к ним доступ? Очевидно, что мы должны определить класс и назвать его **vector**. Далее, нужен один член класса для хранения размера вектора и еще один — для хранения его элементов. Но как представить множество элементов, количество которых может изменяться? Для этого можно было бы использовать стандартный класс **vector**, но в данном контексте это было бы мошенничеством: мы же как раз этот класс и разрабатываем.

Итак, как представить стрелку, изображенную на рисунке? Представим себе, что ее нет. Мы можем определить следующую структуру данных фиксированного размера.

```
class vector {
    int size, age0, age1, age2, age3;
    // ...
};
```

Игнорируя некоторые детали, связанные с обозначениями, мы получим нечто наподобие изображенного на следующем рисунке.



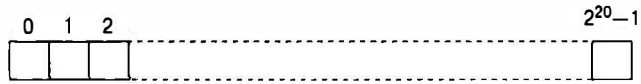
Это просто и красиво, но как только мы попробуем добавить в вектор элемент с помощью функции **push_back()**, мы окажемся в затруднительном положении: мы не можем добавить элемент, так как количество элементов зафиксировано и равно четырем. Нам нужно нечто большее, чем структура данных, хранящая фиксированное количество элементов. Операции, изменяющие количество элементов в объекте класса **vector**, такие как **push_back()**, невозможно реализовать, если количество элементов в классе **vector** фиксировано. Нам нужен некоторый член класса, ссылающийся на множество элементов так, чтобы при расширении памяти он мог ссылаться на другое множество элементов. Нам нужно что-то вроде адреса первого элемента. В языке C++ тип данных, способный хранить адрес, называют *указателем* (pointer). Синтаксически он выделяется суффиксом *, так что **double*** означает указатель на объект типа **double**. Теперь можем определить первый вариант класса **vector**.

```
// ОЧЕНЬ упрощенный вектор элементов типа double
class vector {
    int sz;         // Размер
    double* elem;   // Указатель на первый элемент (типа double)
public:
    vector(int s); // Конструктор: размещает в памяти s чисел
                   // типа double, направляет на них указатель
                   // elem и сохраняет s в члене sz
    int size() const { return sz; } // Текущий размер
};
```

Прежде чем продолжить проектирование класса **vector**, давайте изучим понятие "указатель" более подробно. Понятие "указатель" (вместе с тесно связанным с ним понятием "массив") — это ключ к понятию "память" в языке C++.

17.3. Память, адреса и указатели

Память компьютера представляет собой последовательность байтов. Эти байты нумеруются от нуля до последнего байта. *Адресом* (address) называют некоторое число, идентифицирующее ячейку в памяти. Адрес можно считать разновидностью целых чисел. Первый байт памяти имеет адрес 0, второй — 1 и т.д. Мегабайт памяти можно визуализировать следующим образом.



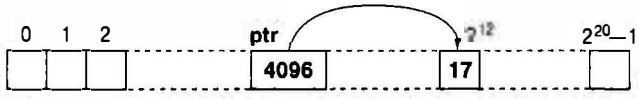
Все, что расположено в памяти, имеет адрес, например:

```
int var = 17;
```

Эта инструкция резервирует участок памяти, размер которого определяется размером типа `int`, для хранения переменной `var` и записывает туда число `17`. Кроме того, адреса можно хранить и выполнять над ними операции. Объект, хранящий адрес, называют *указателем*. Например, тип, необходимый для хранения адреса объекта типа `int`, называется указателем на `int` и обозначается как `int*`.

```
int* ptr = &var; // Указатель ptr хранит адрес переменной var
```

Для определения адреса объекта используется оператор получения адреса, унарный `&`. Так, если переменная `var` хранится в участке памяти, первая ячейка которого имеет адрес 4096 (или 2^{12}), то указатель `ptr` будет хранить число 4096.



В принципе, память компьютера можно рассматривать как последовательность байтов, пронумерованную от `0` до размера памяти в байтах минус единица. Для некоторых машин такое утверждение носит слишком упрощенный характер, но для нашей модели этого пока достаточно.

Каждый тип имеет соответствующий тип указателя, например:

```
int   x = 17;
int* pi = &x;       // Указатель на int

double   e = 2.71828;
double* pd = &e;    // Указатель на double
```

Если мы хотим увидеть значение объекта, на который указываем, то можем применить к указателю оператор разыменования, унарный `*`, например:

```
cout << "pi==" << pi << "; содержимое pi==" << *pi << "\n";
cout << "pd==" << pd << "; содержимое pd==" << *pd << "\n";
```

Значением `*pi` является целое число `17`, а значением `*pd` — значение типа `double`, равное `2.71828`. Значения переменных `pi` и `pd` зависят от того, как компилятор размещает переменные `x` и `e` в памяти. Запись, используемая для значения указателя (адреса), также может изменяться в зависимости от того, какие соглашения приняты в системе; для обозначения значений указателей часто используются шестнадцатеричные числа (раздел A.2.1.1).

Оператор *разыменования* может также находиться в левой части оператора присваивания:

```
*pi = 27;         // ОК: присваиваем 27 переменной типа int,
                  // на которую указывает pi
*pd = 3.14159;    // ОК: присваиваем 3.14159 переменной типа
                  // double, на которую указывает pd
```

```
*pd = *pi;      // ОК: присваиваем значение типа int (*pi)
                // переменной типа double (*pd)
```

Обратите внимание, что несмотря на то, что значение указателя выводится как целочисленное значение, сам указатель целым числом не является. "На что указывает **int**?" — некорректный вопрос. Указывают не целые числа, а указатели. Тип указателя позволяет выполнять операции над адресами, в то время как тип **int** позволяет выполнять (арифметические и логические) операции над целыми числами. Указатели и целые числа нельзя неявно смешивать.

```
int i = pi; // Ошибка: нельзя присвоить объект типа int* объекту типа int
pi = 7;     // Ошибка: нельзя присвоить объект типа int объекту типа int*
```

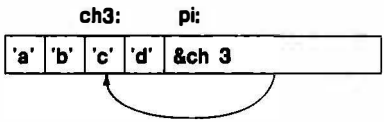
Аналогично указатель на **char** (т.е. **char***) не является указателем на **int** (т.е. **int***), например:

```
pc = pi;    // Ошибка: нельзя присвоить объект типа int* объекту типа char*
pi = pc;    // Ошибка: нельзя присвоить объект типа char* объекту типа int*
```

Почему нельзя присвоить переменную **pc** переменной **pi**? Рассмотрим один из ответов. Тип **char** обычно меньше типа **int**; рассмотрите с учетом этого следующий код.

```
char ch1 = 'a';
char ch2 = 'b';
char ch3 = 'c';
char ch4 = 'd';
int* pi = &ch3; // Указывает на ch3, область памяти, размер которой равен
                // размеру типа char.Ошибка: нельзя присвоить объект char*
                // объекту типа int*
                // Если представить, что это допустимо, то
*pi = 12345;    // получается запись в участок памяти,
*pi = 67890;    // имеющий размер типа int
```

Как именно компилятор размещает переменные в памяти, зависит от его реализации, но, скорее всего, это выглядит следующим образом.



Если бы компилятор разрешал такой код, то мы могли бы записать число **12345** в ячейку памяти, начинающуюся с адреса **&ch3**. Это изменило бы содержание памяти рядом, т.е. значения переменных **ch2** или **ch4**. В худшем (и вполне реальном) случае мы бы перезаписали часть самой переменной **pi**! В этом случае очередное присваивание ***pi=67890** привело бы к размещению числа **67890** в совершенно другой области памяти. Очень хорошо, что такое присваивание запрещено, но таких механизмов защиты на низком уровне программирования очень мало.

В редких ситуациях, когда нам требуется преобразовать переменную типа `int` в указатель или конвертировать один тип показателя в другой, можно использовать оператор `reinterpret_cast` (подробнее об этом — в разделе 17.8).

Мы действительно очень близки к аппаратному обеспечению. Для программиста это не слишком удобно. В нашем распоряжении лишь несколько примитивных операций и почти нет языковой или библиотечной поддержки. Однако нам необходимо знать, как реализованы высокоуровневые средства, такие как класс `vector`. Мы должны знать, как написать код на низком уровне, поскольку не всякий код может быть высокоуровневым (см. главу 25). Кроме того, для того чтобы оценить удобство и относительную надежность высокоуровневого программирования, необходимо почувствовать сложность низкоуровневого программирования. Наша цель — всегда работать на наивысшем уровне абстракции, который допускает поставленная задача и сформулированные ограничения. В этой главе, а также в главах 18 и 19 мы покажем, как вернуться на более комфортабельный уровень абстракции, реализовав класс `vector`.

17.3.1. Оператор `sizeof`

Сколько памяти требуется для хранения типа `int`? А указателя? Ответы на эти вопросы дает оператор `sizeof`.

```
void sizes(char ch, int I, int* pi)
{
    cout << "размер типа char равен " << sizeof(char) << ' '
         << sizeof(ch) << '\n';
    cout << "размер типа int  равен " << sizeof(int)  << ' '
         << sizeof(i) << '\n';
    cout << "размер типа int* равен " << sizeof(int*) << ' '
         << sizeof(p) << '\n';
}
```

Как видите, можно применить оператор `sizeof` как к имени типа, так и к выражению; для типа оператор `sizeof` возвращает размер объекта данного типа, а для выражения — размер типа его результата. Результатом оператора `sizeof` является положительное целое число, а единицей измерения объема памяти является значение `sizeof(char)`, которое по определению равно 1. Как правило, тип `char` занимает один байт, поэтому оператор `sizeof` возвращает количество байтов.

ПОПРОБУЙТЕ

Выполните код, приведенный выше, и посмотрите на его результаты. Затем расширьте этот пример для определения размера типов `bool`, `double` и некоторых других.

Не гарантируется совпадение размеров одного и того же типа в разных реализациях языка C++. В настоящее время выражение **sizeof(int)** в настольных компьютерах и ноутбуках обычно равно четырем. При 8-битовом байте это значит, что тип **int** занимает 32 бита. Однако в процессорах встроенных систем тип **int** занимает 16 битов, а в высокопроизводительных архитектурах размер типа **int** часто равен 64 битам.

Сколько памяти занимает объект класса **vector**? Попробуем выяснить.

```
vector<int> v(1000);  // Вектор с 1000 элементов
cout << "Размер vector<int>(1000) = "
     << sizeof (v) << '\n';
```

Результат может выглядеть так:

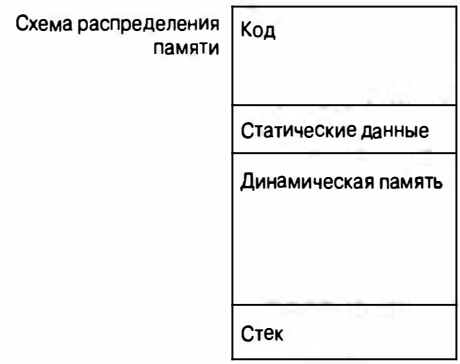
```
Размер vector<int>(1000) = 20
```

Причины этого факта станут очевидными по мере чтения этой и следующей глав (см. также раздел 19.2.1), но уже сейчас ясно, что оператор **sizeof** не пересчитывает элементы.

17.4. Динамически распределяемая память и указатели

Рассмотрим реализацию класса **vector**, приведенную в конце раздела 17.2. Где класс **vector** находит место для хранения своих элементов? Как установить указатель **elem** так, чтобы он указывал на них? Когда начинается выполнение программы, написанной на языке C++, компилятор

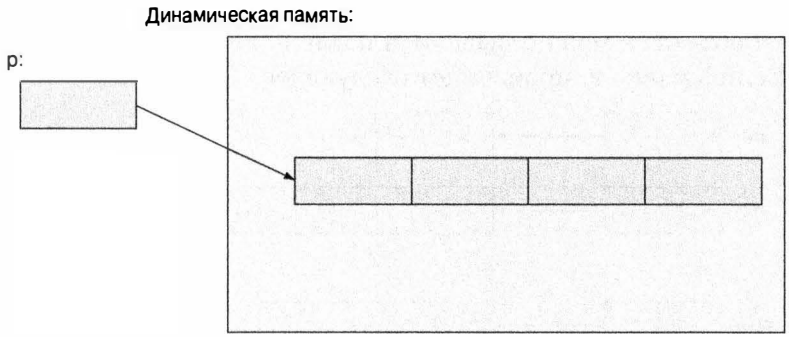
ирует память под код (иногда эту память называют *сегментом кода*) и глобальные переменные (эту память называют *сегментом данных*). Кроме того, выделяется память, которая будет использоваться при вызове функций для хранения их аргументов и локальных переменных (эта память называется *стеком*). Остальная память компьютера может использоваться для других целей; она называется *динамически распределяемой* или просто *динамической*. Это распределение памяти можно проиллюстрировать следующим образом.



Язык C++ делает эту динамическую память (которую также называют *кучей* (heap)) доступной с помощью оператора **new**:

```
double* p = new double[4]; // Размещаем 4 числа double
                           // в динамической памяти
```

Указанная выше инструкция просит систему поддержки выполнения программ разместить четыре числа типа **double** в динамической памяти и вернуть указатель на первое из них. Этот указатель используется для инициализации переменной **p**. Схематически это выглядит следующим образом.



Оператор **new** возвращает указатель на объект, который он создал. Если оператор **new** создал несколько объектов (массив), то он возвращает указатель на первый из этих массивов. Если этот объект имеет тип **X**, то указатель, возвращаемый оператором **new**, имеет тип **X***:

```
char* q = new double[4]; // Ошибка: указатель double*
                         // присваивается переменной char*
```

Данный оператор **new** возвращает указатель на переменную типа **double**, но тип **double** отличается от типа **char**, поэтому мы не должны (и не можем) присвоить указатель на переменную типа **double** указателю на переменную типа **char**.

17.4.1. Размещение в динамической памяти

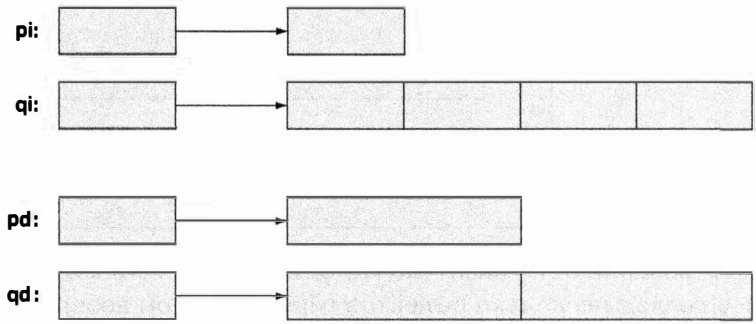
Оператор **new** выполняет *выделение* (allocation) *динамической памяти* (free store).

- Оператор **new** возвращает указатель на выделенную память.
- Значением указателя является адрес первого байта выделенной памяти.
- Указатель указывает на объект определенного типа.
- Указатель *не знает*, на какое количество элементов он указывает.

Оператор **new** может выделять память как для отдельных элементов, так и для последовательности элементов:

```
int* pi = new int;     // Выделяем память для одной переменной int
int* qi = new int[4]; // Выделяем память для четырех переменных
                      //  int (массива)
double* pd = new double;     // Выделяем память для одной переменной
                             // типа double
double* qd = new double[n]; // Выделяем память для n переменных
                             // типа double
```

Обратите внимание на то, что количество объектов может задаваться переменной. Это важно, поскольку позволяет нам выбирать, для какого количества объектов можно выделить память, во время выполнения программы. Если **n** равно 2, произойдет следующее.



Указатели на объекты разных типов имеют разные типы. Рассмотрим пример.

```
pi = pd; // Ошибка: нельзя присвоить double* указателю int*
pd = pi; // Ошибка: нельзя присвоить int* указателю double*
```

Почему нельзя? В конце концов, мы же можем присвоить переменную типа **int** переменной типа **double**, и наоборот. Причина заключается в операторе **[]**. Для того чтобы найти нужный элемент, он использует информацию о размере его типа. Например, элемент **qi[2]** находится на расстоянии, равном двум размерам типа **int** от элемента **qi[0]**, а элемент **qd[2]** находится на расстоянии, равном двум размерам типа **double** от элемента **qd[0]**. Если размер типа **int** отличается от размера типа **double**, как во многих компьютерах, то, разрешив указателю **qi** указывать на память, выделенную для адресации указателем **qd**, можем получить довольно странные результаты.

Это объяснение с практической точки зрения. С теоретической точки зрения ответ таков: присваивание друг другу указателей на разные типы сделало бы возможными *ошибки типа* (type errors).

17.4.2. Доступ с помощью указателей

Кроме оператора разыменования `*`, к указателю можно применять оператор индексирования `[]`:

```
double* p = new double[4]; // Выделяем память для четырех
                           // переменных типа double
double x = *p;             // Читаем первый объект
                           // с помощью указателя p
double y = p[2];           // Читаем третий объект
                           // с помощью указателя p
```

Так же как и в классе **vector**, оператор индексирования начинает отсчет от нуля. Это значит, что выражение `p[2]` указывает на третий элемент; `p[0]` — это первый элемент, поэтому `p[0]` означает то же самое, что и `*p`. Операторы `[]` и `*` можно также использовать для записи.

```
*p = 7.7;    // Записываем число в первый объект
             // с помощью указателя p
p[2] = 9.9; // Записываем число в третий объект
             // с помощью указателя p
```

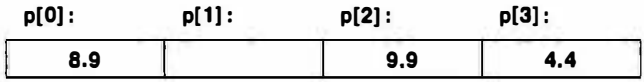
Указатель указывает на объект, расположенный в памяти. Оператор *разыменования* позволяет читать и записывать объект, на который указывает указатель `p`.

```
double x = *p; // Читаем объект, на который указывает p
*p = 8.9;      // Записываем объект, на который указывает p
```

Когда оператор `[]` применяется к указателю `p`, он интерпретирует память как последовательность объектов (имеющих тип, указанный в объявлении указателя), на первый из которых указывает указатель `p`.

```
double x = p[3]; // Читаем четвертый объект с помощью
                 // указателя p
p[3] = 4.4;      // Записываем четвертый объект с помощью
                 // указателя p
double y = p[0]; // p[0] - то же самое, что и *p
```

Вот и все. Здесь нет никаких проверок, никакой тонкой реализации — простой доступ к памяти.



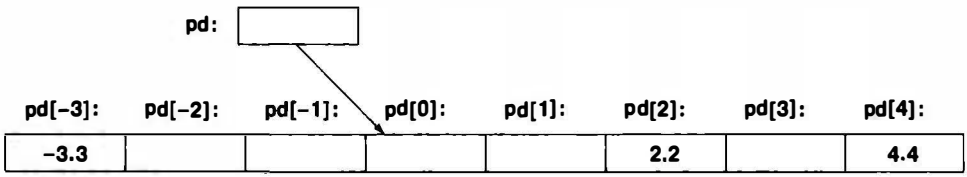
Именно такой простой и оптимально эффективный механизм доступа к памяти нужен нам для реализации класса **vector**.

17.4.3. Диапазоны

Основная проблема, связанная с указателями, заключается в том, что указатель не знает, на какое количество элементов он указывает. Рассмотрим пример.

```
double* pd = new double[3];
pd[2]  =  2.2;
pd[4]  =  4.4;
pd[-3] = -3.3;
```

Имеется ли третий элемент там, куда указывает указатель **pd**? Можно ли обращаться к пятому элементу **pd[4]**? Если мы посмотрим на определение указателя **pd**, то ответим "да" и "нет" соответственно. Однако компилятор об этом не знает; он не отслеживает значения указателей. Наш код просто обращается к памяти так, будто она распределена правильно. Компилятор даже не возразит против выражения **pd[-3]**, как будто можно разместить три числа типа **double** перед элементом, на который указывает указатель **pd**.



Нам не известно, что собой представляют ячейки памяти, на которые указывают выражения **pd[-3]** и **pd[4]**. Однако мы знаем, что они не могут использоваться как часть нашего массива, в котором хранятся три числа типа **double**, на которые указывает указатель **pd**. Вероятнее всего, они являются частью других объектов, и мы просто заблудились. Это плохо. Это катастрофически плохо. Здесь слово "катастрофически" означает либо "моя программа почему-то завершилась аварийно", либо "моя программа выдает неправильные ответы". Попытайтесь произнести это вслух; звучит ужасно. Нужно очень многое сделать, чтобы избежать подобных фраз. Выход за пределы допустимого диапазона представляет собой особенно ужасную ошибку, поскольку очевидно, что при этом опасности подвергаются данные, не имеющие отношения к нашей программе. Считывая содержимое ячейки памяти, находящейся за пределами допустимого диапазона, мы получаем случайное число, которое может быть результатом совершенно других вычислений. Выполняя запись в ячейку памяти, находящуюся за пределами допустимого диапазона, можем перевести какой-то объект в "невозможное" состояние или просто получить совершенно неожиданное и неправильное значение. Такие действия, как правило, остаются незамеченными достаточно долго, поэтому их особенно трудно выявить. Что еще хуже: дважды выполняя программу, в которой происходит выход за

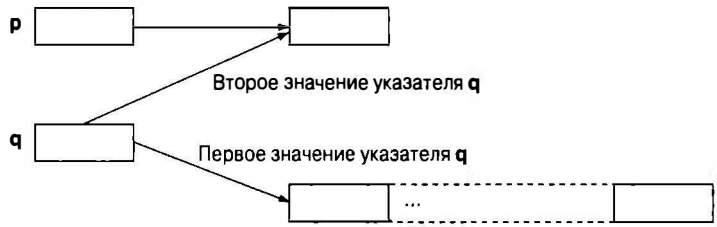
пределы допустимого диапазона, с немного разными входными данными, мы можем прийти к совершенно разным результатам. Ошибки такого рода ("неустойчивые ошибки") выявить труднее всего.

Мы должны гарантировать, что выхода за пределы допустимого диапазона не будет. Одна из причин, по которым мы используем класс **vector**, а не выделяем память непосредственно с помощью оператора **new**, заключается именно в том, что класс **vector** знает свой размер и поэтому выход за пределы допустимого диапазона можно предотвратить.

Предотвратить выход за пределы допустимого диапазона сложно по многим причинам. Одна из них заключается в том, что мы можем присваивать один указатель **double*** другому указателю **double*** независимо от количества элементов, на которые они указывают. Указатель действительно не знает, на сколько элементов он указывает. Рассмотрим пример.

```
double* p = new double;        // Разместить один double
double* q = new double[1000]; // Разместить тысячу double

q[700] = 7.7;                  // Никаких препятствий
q = p;                         // q указывает на то же, что и p
double d = q[700];             // Выход за пределы диапазона!
```

Здесь всего три строки кода, в которых выражение **q[700]** указывает на две разные ячейки памяти, причем во втором случае происходит опасный выход за пределы допустимого диапазона.



Теперь, мы надеемся, вы спросите "А почему указатель не может помнить размер памяти?" Очевидно, что можно было бы разработать указатель, который помнил бы, на какое количество элементов он указывает, — в классе **vector** это сделано практически так. А если вы прочитаете книги, посвященные языку C++, и просмотрите его библиотеки, то обнаружите множество "интеллектуальных указателей", компенсирующих этот недостаток встроенных низкоуровневых указателей. Однако в некоторых ситуациях нам нужны низкоуровневый доступ и понимание механизма адресации объектов, а машина не знает, что именно она адресует. Кроме того, знание механизма работы указателей важно для понимания огромного количества уже написанных программ.

17.4.4. Инициализация

Как всегда, мы хотели бы, чтобы объект уже имел какое-то значение, прежде чем мы приступим к его использованию; иначе говоря, мы хотели бы, чтобы указатели и объекты, на которые они указывают, были инициализированы. Рассмотрим пример.

```
double* p0;                    // Объявление без инициализации:
                               // возможны проблемы
double* p1 = new double;       // Выделение памяти для переменной
                               // типа double без инициализации
double* p2 = new double(5.5); // Инициализируем переменную типа
                               // double числом 5.5
double* p3 = new double[5];    // Выделение памяти для массива из
                               // пяти double без инициализации
```

Очевидно, что объявление указателя **p0** без инициализации может вызвать проблемы. Рассмотрим пример.

```
*p0 = 7.0;
```

Эта инструкция записывает число **7.0** в некую ячейку памяти. Мы не знаем, в какой части памяти расположена эта ячейка. Это может быть безопасно, но рассчитывать на это нельзя. Рано или поздно мы получим тот же результат, что и при выходе за пределы допустимого диапазона: программа аварийно завершит работу или выдаст неправильные результаты.

Огромное количество серьезных проблем в программах, написанных в старом стиле языка C, вызвано использованием неинициализированных указателей и выходом за пределы допустимого диапазона. Мы должны делать все, чтобы избежать таких проблем, частично потому, что наша цель — профессионализм, а частично потому, что мы не

хотим терять время в поисках ошибок такого рода. Выявление и устранение таких ошибок — ужасно нудное и неприятное дело. Намного приятнее и продуктивнее предотвратить такие ошибки, чем вылавливать их.

Память, выделенная оператором **new** для встроенных типов, не инициализируется. Если вы выделяете память для одного объекта и вас не устраивает ее неинициализированность, укажите конкретное значение, как это было сделано при объявлении указателя **p2**: после этого значение ***p2** равно **5.5**. Обратите внимание на круглые скобки **()**, используемые при инициализации. Не перепутайте их с квадратными скобками, **[]**, которые используются для указания массивов!

Мы можем указать список инициализации для массивов объектов, память для которых выделяется оператором **new**, например:

```
double* p4 = new double[5] {0,1,2,3,4};
double* p5 = new double[]  {0,1,2,3,4};
```

Теперь указатель **p4** указывает на объекты типа **double**, содержащие числа **0.0**, **1.0**, **2.0**, **3.0** и **4.0**. То же самое можно сказать и об указателе **p5**; если указан набор элементов выделяемого массива, количество элементов можно не указывать.

Как обычно, мы должны избегать неинициализированных объектов и следить за тем, чтобы они получили значения до того, как будут прочитаны. Компиляторы часто имеют режим отладки, в котором они по умолчанию инициализируют все переменные предсказуемой величиной (обычно нулем). В результате, когда вы отключаете режим отладки, чтобы отправить программу заказчику, запускаете оптимизатор или просто компилируете программу на другой машине, программа, содержащая неинициализированные переменные, может внезапно перестать работать правильно. Не используйте неинициализированные переменные.

Когда мы используем собственные типы, мы получаем больший контроль над инициализацией. Если класс **X** имеет конструктор по умолчанию, то мы получим следующее:

```
X* px1 = new X;      // Один объект класса X, инициализированный
                     // по умолчанию
X* px2 = new X[17]; // 17 объектов класса X, инициализированных
                     // по умолчанию
```

Если тип **Y** имеет конструктор, но не конструктор по умолчанию, мы должны выполнить явную инициализацию:

```
Y* py1 = new Y;      // Ошибка: нет конструктора по умолчанию
Y* py2 = new Y{13}; // OK: инициализирован как Y{13}
Y* py3 = new Y[17]; // Ошибка: нет конструктора по умолчанию
Y* py4 = new Y[17] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

Длинный список инициализации для **new** может оказаться непрактичным, но когда создается только несколько элементов (а это достаточно распространенный случай), это может быть удобным.

17.4.5. Нулевой указатель

Если в вашем распоряжении нет другого указателя, которым можно было бы инициализировать ваш указатель, используйте нулевой указатель **nullptr**.

```
double* p0 = nullptr; // Нулевой указатель
```

При присваивании указателю нулевого значения последнее носит название *нулевого указателя* (null pointer), и зачастую корректность указателя (т.е. то, что он указывает на что-то) проверяется с помощью сравнения его значения с **nullptr**:

```
if (p0 != nullptr) // Проверка корректности указателя p0
```

Этот тест неидеален, поскольку указатель **p0** может содержать случайное ненулевое значение (например, если мы забыли его инициализировать) или адрес объекта, который был удален с помощью оператора **delete** (см. раздел 17.4.6). Однако такая проверка часто оказывается лучшим, что можно сделать. В действительности мы не обязаны явно указывать **nullptr**, поскольку инструкция **if** проверяет, является ли условие ненулевым.

```
if (p0) // Проверка корректности указателя p0;
        // эквивалентно p0!=nullptr
```

Мы предпочитаем более короткую форму проверки, полагая, что она точнее отражает смысл выражения "**p0** корректен", но это дело вкуса.

Нулевой указатель следует использовать тогда, когда некоторый указатель временами указывает на какой-то объект, а временами нет. Эта ситуация встречается реже, чем можно себе представить; подумайте сами: если у вас нет объекта, на который будет указывать указатель, то зачем вам создавать сам указатель? Почему бы не подождать, пока не будет создан объект?

Имя **nullptr** для нулевого указателя появилось в стандарте C++11, так что в старом коде программисты часто использовали **0** (нуль) или **NULL** вместо **nullptr**. Оба старых варианта могут приводить к неразберихе и/или ошибкам, так что предпочтительнее использовать более точное значение **nullptr**.

17.4.6. Освобождение памяти

Оператор **new** выделяет участок динамической памяти. Поскольку память компьютера ограничена, неплохо было бы возвращать память обратно, когда она станет больше ненужной. В этом случае освобожденную память можно было бы использовать для хранения других объектов. Для больших и долго работающих программ такое освобождение памяти играет важную роль. Рассмотрим пример.

```
double* calc(int res_size, int max) // Утечка памяти
{
    double* p = new double[max];
    double* res = new double[res_size];
    // Используем указатель p для вычисления
    // результатов и записи их в массив res
    return res;
}

double* r = calc(100,1000);
```

При работе такой программы каждый вызов функции **calc()** будет "терять" память, выделяемую под массив элементов **double**, адрес которого

присваивается указателю **p**. Например, вызов **calc(100,1000)** сделает недоступным для остальной части программы участок памяти, на котором может разместиться тысяча переменных типа **double**.

Оператор, освобождающий память, называется **delete**. Для того чтобы освободить память для дальнейшего использования, оператор **delete** следует применить к указателю, который был возвращен оператором **new**:

```
double* calc(int res_size, int max)
    // За память, выделяемую для массива res,
    // несет ответственность вызывающий код
{
    double* p = new double[max];
    double* res = new double[res_size];
    // Используем указатель p для вычисления
    // результатов и их записи в res
    delete[] p; // Эта память больше не нужна, освобождаем ее
    return res;
}

double* r = calc(100,1000);
// Используем указатель r
delete[] r; // Эта память больше не нужна, освобождаем ее
```

Между прочим, этот пример демонстрирует одну из основных причин использования динамической памяти: мы можем создавать объекты в функциях и передавать их обратно вызывающему коду.

Оператор **delete** имеет две разновидности:

- **delete p** освобождает память, выделенную с помощью оператора **new** для отдельного объекта;
- **delete[] p** освобождает память, выделенную с помощью оператора **new** для массива объектов.

Выбор правильного варианта должен сделать программист.

Двойное удаление объекта — очень грубая ошибка. Рассмотрим пример.

```
int* p = new int(5);
delete p; // Отлично: p указывает на объект,
          // созданный оператором new
// ... Указатель здесь больше не используется ...
delete p; // Ошибка: p указывает на память, принадлежащую
          // диспетчеру динамической памяти
```

Вторая инструкция **delete p** порождает две проблемы.

- Вы больше не владеете объектом, поэтому диспетчер динамической памяти может изменить внутреннюю структуру данных так, что выполнить инструкцию **delete p** правильно во второй раз будет невозможно.

- Диспетчер динамической памяти может повторно использовать память, на которую указывал указатель `p`, так что теперь указатель `p` указывает на другой объект; удаление этого объекта (принадлежащего другой части программы) приведет к ошибке.

Обе проблемы встречаются в реальных программах, так что это не просто теоретические рассуждения.

Удаление нулевого указателя не приводит ни к каким последствиям (так как нулевой указатель не указывает ни на один объект), поэтому эта операция безвредна:

```
int* p = 0;
delete p; // Отлично: никаких действий не нужно
delete p; // То же самое — по-прежнему никаких действий
```

Зачем возиться с освобождением памяти? Разве компилятор сам не может понять, когда память нам больше не нужна, и освободить ее без вмешательства человека? Может. Такой механизм называется *автоматической сборкой мусора* (automatic garbage collection) или просто *сборкой мусора* (garbage collection). К сожалению, автоматическая сборка мусора — удовольствие недешевое и не подходит идеально для всех приложений. Если вам действительно нужна автоматическая сборка мусора, можно встроить этот механизм в свою программу. Имеются хорошие сборщики мусора (см. `www.stroustrup.com/C++.html`). Однако в этой книге мы предполагаем, что читатели сами разберутся со своим "мусором", а мы покажем, как это сделать удобно и эффективно.

Почему следует избегать утечки памяти? Программа, которая должна работать "бесконечно", не должна допускать никаких утечек памяти. Примерами таких программ являются операционная система, а также большинство встроенных систем (см. главу 25). Библиотеки также не должны допускать утечек памяти, поскольку кто-нибудь может использовать эти библиотеки как часть системы, работающей бесконечно. Словом, в общем случае утечек памяти следует избегать, и все тут. Многие программисты рассматривают утечки памяти как проявление небрежности. Однако эта точка зрения кажется нам слишком категоричной. Если программа выполняется под управлением операционной системы (Unix, Windows или какой-нибудь еще), то после завершения работы программы вся память автоматически возвращается системе. Отсюда следует, что если вам известно, что ваша программа не будет использовать больше памяти, чем ей доступно, то вполне можно допустить утечку, пока операционная система сама не восстановит порядок. Тем не менее, если вы решитесь на это, то надо быть уверенным в том, что ваша оценка объема используемой памяти является правильной, иначе вас сочтут небрежным.

17.5. Деструкторы

Теперь мы знаем, как хранить элементы в векторе. Мы просто выделим достаточное количество динамической памяти и будем обращаться к ней с помощью указателя.

```
// Очень упрощенный вектор чисел типа double
class vector {
    int sz;        // Размер
    double* elem; // Указатель на элементы
public:
    vector(int s)              // Конструктор
        :sz(s),                // Инициализация члена sz
        elem(new double[s])  // Инициализация члена elem
        {
            for(int i=0; i<s; ++i)  // Инициализация
                elem[i]=0;          // элементов
        }
    int size() const { return sz; } // Текущий размер
    // ...
};
```

Таким образом, количество элементов хранится в члене **sz**. Мы инициализируем его в конструкторе, а пользователь класса **vector** может выяснить количество элементов, вызвав функцию **size()**. Память для элементов выделяется в конструкторе с помощью оператора **new**, а указатель, возвращенный оператором **new**, хранится в члене **elem**.

Обратите внимание на то, что мы инициализируем элементы их значением по умолчанию (**0.0**). Класс **vector** из стандартной библиотеки делает именно так, поэтому мы решили поступать так же с самого начала.

К сожалению, наш примитивный класс **vector** допускает утечку памяти. В конструкторе он выделяет память для элементов с помощью оператора **new**. Следуя правилу, сформулированному в разделе 17.4, мы должны освободить эту память с помощью оператора **delete**. Рассмотрим пример.

```
void f(int n)
{
    vector v(n); // Выделяем память для n чисел типа double
    // ...
}
```

После выхода из функции **f()** элементы вектора **v**, созданные в динамической памяти, не удаляются. Мы могли бы определить функцию **clean_up()** — член класса **vector** и вызвать ее следующим образом:

```
void f2(int n)
{
    vector v(n);  // Вектор, выделяющий память для
                  // n переменных типа int
    // ... используем вектор v ...
```

```
    v.clean_up(); // Функция clean_up() удаляет член elem
}
```

Этот подход будет работать. Однако одна из наиболее распространенных проблем, связанных с динамической памятью, заключается в том, что люди забывают об операторе **delete**. Та же проблема может возникнуть и с функцией **clean_up()**: программисты просто будут забывать ее вызвать. Мы можем предложить более удачное решение. Основная идея состоит в том, чтобы компилятор знал не только о конструкторе, но и о функции, играющей роль, противоположную роли конструктора. Такую функцию логично назвать *деструктором* (destructor). Точно так же как конструктор неявно вызывается при создании объекта класса, деструктор неявно вызывается, когда объект выходит за пределы области видимости. Конструктор гарантирует, что объект будет правильно создан и проинициализирован. Деструктор, наоборот, гарантирует, что объект будет правильно очищен перед тем, как будет уничтожен. Рассмотрим пример.

```
class vector {
    int sz;        // Размер
    double* elem; // Указатель на элементы
public:
    vector(int s)              // Конструктор
        :sz{s},                // Инициализация члена sz
        elem{new double[s]}    // Инициализация члена elem
        {
            for(int i=0; i<s; ++i)  // Инициализация
                elem[i]=0;          // элементов
        }
    ~vector()                  // Деструктор
        { delete[] elem; }     // Освобождаем память
};
```

Теперь можно написать следующий код:

```
void f3(int n)
{
    double* p = new double[n]; // Выделяем память для n
                               // чисел типа double
    vector v(n);               // Определяем вектор (выделяем
                               // память для n других double)
    // ... используем p и v ...
    delete[] p;                // Освобождаем память, занятую
                               // массивом чисел типа double
} // vector автоматически освободит память, занятую объектом v
```

Неожиданно оператор **delete[]** начинает выглядеть таким утомительным и подверженным ошибкам! Имея класс **vector**, нет необходимости ни выделять память с помощью оператора **new**, ни освобождать ее с помощью оператора **delete[]** при выходе из функции. Все это намного лучше

сделает класс **vector**. В частности, класс **vector** никогда не забудет вызвать деструктор, чтобы освободить память, занятую его элементами.

Здесь мы не собираемся глубоко вдаваться в детали использования деструкторов. Отметим лишь, что они играют очень важную роль при работе с ресурсами, которые сначала захватываются, а затем возвращаются обратно — с файлами, потоками, блокировками и т.д. Помните, как выполняется закрытие потоков **iostream**? Они сбрасывают буфера, закрывают файлы, освобождают память и т.д. Все это делают их деструкторы. Каждый класс, "владеющий" какими-то ресурсами, должен иметь деструктор.

17.5.1. Генерируемые деструкторы

Если член класса имеет деструктор, то этот деструктор будет вызываться при уничтожении объекта, содержащего этот член:

```
struct Customer {
    string name;
    vector<string> addresses;
    // ...
};

void some_fct()
{
    Customer fred;
    // Инициализация объекта fred
    // Использование объекта fred
}
```

Когда мы выйдем из функции **some_fct()** и объект **fred** покинет свою область видимости, он будет уничтожен; иначе говоря, будут вызваны деструкторы для строки **name** и вектора **addresses**. Это совершенно необходимо, поскольку иначе могут возникнуть проблемы. Происходящее иногда выражается таким образом: "компилятор сгенерировал деструктор для класса **Customer**, который вызывает деструкторы членов". Такая генерация часто выполняется компилятором и позволяет гарантированно вызывать деструкторы членов класса.

Деструкторы членов (и базовых классов) неявно вызываются из деструктора производного класса (либо определенного пользователем, либо сгенерированного). По существу, все правила сводятся к одному: "деструктор вызывается тогда, когда уничтожается объект" (при выходе из области видимости, при выполнении оператора **delete** и т.д.).

17.5.2. Деструкторы и динамическая память

Деструкторы концептуально просты, но в то же время они образуют основу для большинства наиболее эффективных методов программирования на языке C++. Основная идея заключается в следующем.

- Какой бы ресурс ни требовался объекту класса для функционирования, он захватывает его в конструкторе.
- На протяжении своего срока существования объект может освобождать ресурсы и запрашивать новые.
- В конце существования объекта деструктор освобождает все ресурсы, которыми владел объект.

Типичным примером является пара "конструктор–деструктор" в классе **vector**, которая управляет динамической памятью. Мы еще вернемся к этой идее в разделе 19.5. А пока рассмотрим важное сочетание механизма управления динамической памятью и иерархии классов. Взгляните на приведенный далее код.

```
Shape* fct()
{
    Text tt{Point{200,200},"Annemarie"};
    // ...
    Shape* p = new Text{Point{100,100},"Nicholas"};
    return p;
}

void f()
{
    Shape* q = fct();
    // ...
    delete q;
}
```

Этот код выглядит логичным — и он действительно логичен. Все работает, но посмотрите, как именно работает, ведь этот код является примером элегантного, важного и простого метода. При выходе из функции **fct()** объект **tt** класса **Text** (см. раздел 3.11), существующий в ней, уничтожается вполне корректно. Класс **Text** имеет член типа **string**, для которого обязательно нужно вызвать деструктор, — класс **string** занимает и освобождает память примерно так же, как и класс **vector**. Для объекта **tt** это просто; компилятор вызывает сгенерированный деструктор класса **Text**, как описано в разделе 17.5.1. А что можно сказать об объекте класса **Text**, возвращаемом функцией **fct()**? Вызывающая функция **f()** понятия не имеет о том, что указатель **q** указывает на объект класса **Text**; ей известно лишь, что он указывает на объект класса **Shape**. Как же инструкция **delete q** сможет вызвать деструктор класса **Text**?

В разделе 14.2.1 мы вскользь упомянули тот факт, что класс **Shape** имеет деструктор. На самом деле в классе **Shape** есть не просто деструктор, а виртуальный деструктор, и в этом ключ ко всему. Когда мы выполняем инструкцию **delete q**, оператор **delete** анализирует тип указателя **q**, чтобы увидеть, нужно ли вызывать деструктор, и если нужно, то он его вызывает. Инструкция **delete q** вызывает деструктор **~Shape()** класса **Shape**. Однако деструктор **~Shape()** является виртуальным, поэтому с помощью механизма вызова виртуальной функции (см. раздел 14.3.1) он вызывает деструктор класса, производного от класса **Shape**, в данном случае — деструктор **~Text()**. Если бы деструктор **Shape::~Shape()** не был виртуальным, то деструктор **Text::~Text()** не был бы вызван и член класса **Text**, имеющий тип **string**, не был бы корректно уничтожен.

Запомните правило: если класс содержит виртуальную функцию, в нем должен быть виртуальный деструктор. Причины этого заключаются в следующем.

1. Если класс имеет виртуальную функцию, то, скорее всего, он будет использован в качестве базового.
2. Если класс является базовым, то его производный класс, скорее всего, будет использовать оператор **new**.
3. Если объект производного класса размещается в памяти с помощью оператора **new**, а работа с ним осуществляется с помощью указателя на базовый класс, то, скорее всего, он будет удален с помощью обращения к указателю на объект базового класса.

Обратите внимание на то, что деструкторы вызываются неявно или косвенно с помощью оператора **delete**. Они никогда не вызываются непосредственно. Это позволяет избежать массы трудоемкой работы.

ПОПРОБУЙТЕ

Напишите небольшую программу, используя базовые классы и члены, в которых определены конструкторы и деструкторы, выводящие информацию о том, что они были вызваны. Затем создайте несколько объектов и посмотрите, как вызываются конструкторы и деструкторы.

17.6. Доступ к элементам

Для того чтобы **vector** был применим в наших программах, нужно уметь читать и записывать его элементы. Для начала рассмотрим простые функции-члены **get()** и **set()**.

```
// Очень упрощенный вектор double
class vector {
```

```
    int sz;        // Размер
    double* elem; // Указатель на элементы
public:
    vector(int s):sz{s},elem{new double[s]} // Конструктор
        { /* ... */ }
    ~vector() { delete[] elem; }             // Деструктор
    int size() const { return sz; }          // Текущий размер

    double get(int n) const { return elem[n]; } // Чтение
    void set(int n, double v) { elem[n]=v; }    // Запись
};
```

Функции **get()** и **set()** обеспечивают доступ к элементам, применяя оператор **[]** к указателю **elem**.

Теперь мы можем создать вектор, состоящий из чисел типа **double**, и использовать его.

```
vector v(5);
for (int i=0; i<v.size(); ++i) {
    v.set(i,1.1*i);
    cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

Результат выглядит так:

```
v[0]==0
v[1]==1.1
v[2]==2.2
v[3]==3.3
v[4]==4.4
```

Данный вариант класса **vector** чрезмерно прост, а код, использующий функции **get()** и **set()**, очень некрасив по сравнению с обычным доступом с помощью индексации (квадратных скобок). Однако наша цель заключается в том, чтобы, начав с небольшого и простого варианта, постепенно развивать его, тестируя на каждом этапе. Эта стратегия расширения и постоянного тестирования минимизирует количество ошибок и время отладки.

17.7. Указатели на объекты класса

Понятие указателя носит универсальный характер, поэтому мы можем устанавливать его указывающим на любое место в памяти. Например, можно использовать указатели на объект класса **vector** точно так же, как и указатели на переменные типа **char**:

```
vector* f(int s)
{
    vector* p = new vector(s); // Вектор в динамической памяти
    // Заполняем *p
```

```
    return p;
}

void ff()
{
    vector* q = f(4);
    // Используем *q
    delete q;                    // Освобождаем память
}
```

Обратите внимание на то, что, когда мы удаляем объект класса **vector** с помощью оператора **delete**, вызывается деструктор класса **vector**.

```
vector* p = new vector(s); // Размещаем вектор в динамической памяти
delete p;                  // Удаляем вектор из памяти
```

При создании объекта класса **vector** в динамической памяти оператор **new** выполняет следующие действия:

- сначала выделяет память для объекта класса **vector**;
- затем вызывает конструктор класса **vector**, чтобы инициализировать объект; этот конструктор выделяет память для элементов объекта класса **vector** и инициализирует их.

Удаляя объект класса **vector**, оператор **delete** выполняет следующие действия:

- сначала вызывает деструктор класса **vector**; этот деструктор, в свою очередь, вызывает деструкторы элементов (если таковые имеются), а затем освобождает память, занимаемую элементами вектора;
- затем освобождает память, занятую объектом класса **vector**.

Обратите внимание на то, как хорошо эти операторы работают рекурсивно (см. раздел. 8.5.8). Используя реальный (стандартный) класс **vector**, мы можем выполнить следующий код:

```
vector<vector<double>>* p = new vector<vector<double>>(10);
delete p;
```

Здесь инструкция **delete p** вызывает деструктор класса **vector<vector <double>>**, а он, в свою очередь, вызывает деструктор элементов класса **vector<double>**, и весь вектор аккуратно освобождается, ни один объект не остается не уничтоженным и утечка памяти не возникает.

Поскольку оператор **delete** вызывает деструкторы (для типов, в которых они предусмотрены, например таких, как **vector**), часто говорят, что он *уничтожает* (destroy) объекты, а не просто *удаляет* их из памяти (deallocate).

Как всегда, следует помнить, что "голый" оператор **new** за пределами конструктора таит в себе опасность забыть об операторе **delete**. Если у вас нет хорошей (т.е. действительно простой, такой

как `Vector_ref` из разделов 13.10 и Д.4) стратегии удаления объектов, попробуйте включить операторы `new` в конструкторы, а операторы `delete` — в деструкторы.

Пока что все идет неплохо, но как же нам получить доступ к членам вектора, имея только указатель? Обратите внимание на то, что все классы поддерживают доступ к своим членам с помощью оператора . (точка), примененного к имени объекта.

```
vector v(4);
int x = v.size();
double d = v.get(3);
```

Аналогично все классы поддерживают оператор -> (стрелка) для доступа к своим членам с помощью указателя на объект:

```
vector* p = new vector(4);
int x = p->size();
double d = p->get(3);
```

Как и оператор . (точка) , оператор -> (стрелка) можно использовать для доступа к данным-членам и функциям-членам. Поскольку встроенные типы, такие как `int` и `double`, не имеют членов, оператор -> к ним не применяется. Операторы "точка" и "стрелка" часто называют *операторами доступа к членам класса* (member access operators).

17.8. Путаница с типами: `void*` и операторы приведения типов

Используя указатели и массивы, расположенные в динамической памяти, мы вступаем в более тесный контакт с аппаратным обеспечением. По существу, наши операции с указателями (инициализация, присваивание, `*` и `[]`) непосредственно отображаются в машинные инструкции. На этом уровне язык предоставляет программисту мало удобств и возможностей, предусматриваемых системой типов. Однако иногда от них приходится отказываться, даже несмотря на меньшую степень защиты от ошибок.

Естественно, мы не хотели бы совсем отказываться от защиты, представляемой системой типов, но иногда у нас нет логичной альтернативы (например, когда мы должны обеспечить работу с программой, написанной на другом языке программирования, в котором ничего не известно о системе типов языка C++). Кроме того, существует множество ситуаций, когда необходимо использовать старые программы, разработанные без учета системы безопасности статических типов. В таких случаях нам нужны две вещи.

- Тип указателя, который указывает на память без учета информации о том, какие объекты в ней размещены.

- Операция, сообщающая компилятору при использовании такого указателя, объект какого типа располагается в указанном месте памяти (принимается без доказательства).

Тип **void*** означает "указатель на ячейку памяти, тип которой компилятору неизвестен". Он используется тогда, когда необходимо передать адрес из одной части программы в другую, причем каждая из них ничего не знает о типе объекта, с которым работает другая часть. Примерами являются адреса, служащие аргументами функций обратного вызова (раздел 16.3.1), а также распределители памяти самого нижнего уровня (такие, как реализация оператора **new**).

Объектов типа **void** не существует, но, как мы видели, ключевое слово **void** означает "функция ничего не возвращает".

```
void v;    // Ошибка: объектов типа void не существует
void f(); // Функция f() ничего не возвращает;
          // f() не возвращает объект типа void
```

Указателю типа **void*** можно присвоить указатель на любой объект, например:

```
void* pv1 = new int;         // OK: int* превращается в void*
void* pv2 = new double[10]; // OK: double* превращается в void*
```

Поскольку компилятор ничего не знает о том, на что ссылается указатель типа **void***, мы должны сообщить ему об этом.

```
void f(void* pv)
{
    void* pv2 = pv;  // Правильно (тип void* для этого
                     // и предназначен)
    double* pd = pv; // Ошибка: невозможно привести тип
                     // void* к double*
    *pv = 7;         // Ошибка: невозможно разыменовать void*
                     // (тип объекта, на который указывает pv,
                     // неизвестен)
    pv[2] = 9;       // Ошибка: void* нельзя индексировать
    int* pi = static_cast<int*>(pv); // OK: явное преобразование
    // ...
}
```

Оператор **static_cast** позволяет явно преобразовать указатели связанных типов один в другой, например такие, как **void*** или **double*** (раздел А.5.7). Имя **static_cast** — это сознательно выбранное отвратительное имя для отвратительного (и опасного) оператора, который следует использовать только в случае крайней необходимости. Его редко можно встретить в программах (если он вообще используется). Операции, такие как **static_cast**, называют *явным преобразованием типа* (explicit type conversion) или просто *приведением* (cast).

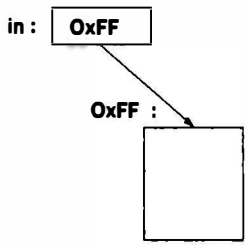
В языке C++ предусмотрены два оператора приведения типов, которые потенциально еще хуже оператора `static_cast`.

- Оператор `reinterpret_cast` может преобразовать тип в совершенно другой, никак не связанный с ним, например `int` в `double*`.
- Оператор `const_cast` позволяет отбросить квалификатор `const`.

Рассмотрим пример.

```
Register* in = reinterpret_cast<Register*>(0xff);

void f(const Buffer* p)
{
    Buffer* b = const_cast<Buffer*>(p);
    // ...
}
```

Первый пример — классическая ситуация, в которой необходимо применить оператор `reinterpret_cast`. Мы сообщаем компилятору, что определенная часть памяти (участок, начинающийся с ячейки `0xFF`) рассматривается как объект класса `Register` (возможно, со специальной семантикой). Такой код необходим, например, при разработке драйверов устройств.



Во втором примере оператор `const_cast` аннулирует квалификатор `const` в объявлении `const Buffer*` указателя `p`. Разумеется, мы должны понимать, что делаем.

Как минимум оператор `static_cast` не позволяет спутать указатели и целые числа или аннулировать квалификатор `const`, поэтому при необходимости привести один тип к другому следует предпочитать оператор `static_cast`. Если вы пришли к выводу, что вам необходимо приведение типов, подумайте еще раз: нельзя ли написать программу иначе, без приведения типов? Можно ли переписать программу так, чтобы приведение типов стало ненужным? Если вам не приходится использовать код, написанный не вами, или управлять аппаратным обеспечением, то, безусловно, можно и нужно обойтись без оператора `static_cast`. В противном случае могут возникнуть трудноуловимые и опасные ошибки. Если вы используете оператор `reinterpret_cast`, то не следует ожидать от него переносимости.

17.9. Указатели и ссылки

Ссылку (reference) можно интерпретировать как автоматически разыменовываемый постоянный указатель или альтернативное имя объекта. Указатели и ссылки отличаются следующими особенностями.

- Присвоение чего-либо указателю изменяет значение указателя, а не объекта, на который он указывает.
- Для того чтобы получить указатель, как правило, необходимо использовать оператор **new** или **&**.
- Для доступа к объекту, на который указывает указатель, используются операторы * и [].
- Присвоение ссылке нового значения изменяет значение объекта, на который она ссылается, а не саму ссылку.
- После инициализации ссылку невозможно перенаправить на другой объект.
- Присваивание ссылок выполняет глубокое копирование (новое значение присваивается объекту, на который указывает ссылка); присваивание указателей не использует глубокое копирование (новое значение присваивается указателю, а не объекту).
- Нулевые указатели представляют опасность.

Рассмотрим пример.

```
int x = 10;
int* p = &x;    // Для получения указателя нужен оператор &
*p = 7;         // Для присвоения значения переменной x
                // через указатель p используется *
int x2 = *p;    // Считываем значение x с помощью указателя p
int* p2 = &x2; // Получаем указатель на другую переменную
                // типа int
p2 = p;         // Указатели p2 и p указывают на переменную x
p = &x2;        // Указатель p указывает на другой объект
```

Соответствующий пример для ссылок приведен ниже.

```
int y = 10;
int& r = y;     // Символ & означает тип, а не инициализатор
r = 7;          // Присвоение значения переменной y
                // с помощью ссылки r (оператор * не нужен)
int y2 = r;     // Считываем переменную y с помощью ссылки r
                // (оператор * не нужен)
int& r2 = y2;   // Ссылка на другую переменную типа int
r2 = r;         // значение переменной y присваивается
                // переменной y2
r = &y2;        // Ошибка: нельзя изменить значение ссылки
                // (нельзя присвоить int* ссылке int&)
```

Обратите внимание на последний пример; это значит не только то, что эта конструкция не работает, — после инициализации невозможно связать ссылку с другим объектом. Если вам нужно указать на другой объект, используйте указатель. Использование указателей описано в разделе 17.9.3.

Как ссылка, так и указатель основаны на адресации памяти. Они просто используют адреса по-разному, чтобы предоставить программисту несколько разные возможности.

17.9.1. Указатели и ссылки как параметры функций

Если вы хотите изменить значение переменной значением, вычисленным функцией, можно использовать три варианта:

```
int incr_v(int x) { return x+1; } // Вычисляет и возвращает новое
                                  // значение
void incr_p(int* p) { ++*p; }     // Передает указатель
                                  // (разыменовывает его
                                  // и увеличивает результат)
void incr_r(int& r) { ++r; }      // Передает ссылку
```

Какой выбор сделать? Нам кажется, что возврат значения чаще приводит к более очевидному (а значит, менее подверженному ошибкам) коду:

```
int x = 2;
x = incr_v(x); // Копируем x в incr_v(); затем копируем
               // результат и присваиваем его
```

Этот стиль предпочтительнее для небольших объектов, таких как тип `int`. Если "большой объект" имеет перемещающий конструктор (см. раздел 18.3.4), то и его можно эффективно передавать туда и обратно.

Как сделать выбор между передачей аргумента по ссылке и с помощью указателя? К сожалению, каждый из этих вариантов имеет свои преимущества и недостатки, поэтому ответ на этот вопрос не очевиден. Каждый программист должен принимать решение в зависимости от ситуации.

Использование передачи аргумента с помощью указателя предостерегает программиста о том, что значение может измениться. Рассмотрим пример.

```
int x = 7;
incr_p(&x);   // Здесь необходим оператор &
incr_r(x);
```

Необходимость использования оператора `&` в вызове функции `incr_p(&x)` предупреждает пользователя о том, что переменная `x` может измениться. Вызов же функции `incr_r(x)`, напротив, выглядит совершенно "невинно". Это приводит к небольшому предпочтению версии с передачей указателя.

С другой стороны, если в качестве аргумента функции вы используете указатель, то следует опасаться, что функции может быть передан нулевой указатель, т.е. `nullptr`. Рассмотрим пример.

```
incr_p(0); // Крах: функция incr_p() пытается разыменовать нуль
int* p = 0;
incr_p(p); // Крах: функция incr_p() пытается разыменовать нуль
```

Совершенно очевидно, что это ужасно. При написании функции **incr_p()** от этого можно защититься:

```
void incr_p(int* p)
{
    if (p==nullptr)
        error("функции incr_p() передан нулевой указатель");
    ++*p;  // Разыменовываем указатель и увеличиваем
           // объект, на который он указывает
}
```

Увы, теперь функция **incr_p()** не выглядит столь же простой и привлекательной, как ранее. В главе 5 обсуждалось, как устранить проблему, связанную с некорректными аргументами. В противоположность этому пользователи, применяющие ссылки (например, в функции **incr_r()**), должны предполагать, что ссылка связана с объектом.

Если "передача ничего" (когда объект на самом деле не передается) с точки зрения семантики функции допустима, аргумент следует передавать с помощью указателя. Примечание: так как это не относится к операции инкремента, при условии **p==nullptr** следует генерировать исключение.

Итак, правильный ответ формулируется так: выбор зависит от природы функции.

- Для маленьких объектов предпочтительнее передача по значению.
- Для функций, допускающих в качестве своего аргумента "отсутствующий объект" (представленный значением **nullptr**), следует использовать указатели (и не забывать о проверке **nullptr**!).
- В противном случае в качестве параметра следует использовать ссылку.

(См. также раздел 8.5.6.)

17.9.2. Указатели, ссылки и наследование

В разделе 14.3 мы видели, как можно использовать производный класс, такой как **Circle**, вместо объекта его открытого базового класса **Shape**. Эту идею можно выразить в терминах указателей или ссылок: указатель **Circle*** можно неявно преобразовать в указатель **Shape***, поскольку класс **Shape** является открытым базовым классом по отношению к классу **Circle**. Рассмотрим пример.

```
void rotate(Shape*s, int n); // Поворачиваем фигуру *s на угол n
```

```
Shape* p = new Circle{Point{100,100},40};
Circle c{Point{200,200},50};
rotate(p,35);
rotate(&c,45);
```

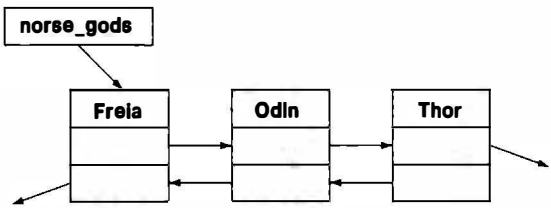
Это можно сделать и с помощью ссылок.

```
void rotate(Shape&s, int n); // Поворачиваем фигуру s на угол n

Shape& r = c;
rotate(r,55);
rotate(*p,65);
rotate(c,75);
```

Этот факт является чрезвычайно важным для большинства объектно-ориентированных технологий программирования (см. разделы 14.3, 14.4).

17.9.3. Пример: списки

Наиболее распространенными и полезными структурами данных являются списки. Как правило, список создается с помощью узлов, каждый из которых содержит определенную информацию и указатель на другие узлы. Это — классический пример использования указателей. Например, короткий список норвежских богов можно представить в следующем виде.



Такой список называют *двусвязным* (doubly-linked list), поскольку для данного узла мы можем определить предшествующий и последующий узлы. Список, в котором существуют только последующие узлы, называют *односвязным* (singly-linked list). Мы используем двусвязные узлы, когда хотим упростить удаление элемента. Узлы списка можно определить следующим образом:

```
struct Link {
    string value;
    Link* prev;
    Link* succ;
    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        : value{v}, prev{p}, succ{s} {}
};
```

Иначе говоря, имея объект типа `Link`, мы можем получить доступ к последующему элементу, используя указатель `succ`, а к предыдущему элементу — используя указатель `prev`. Нулевой указатель позволяет указать, что

узел не имеет предшествующего или последующего узла. Список норвежских богов можно закодировать так:

```
Link* norse_gods = new Link{"Thor",0,0};
norse_gods = new Link{"Odin",0,norse_gods};
norse_gods->succ->prev = norse_gods;
norse_gods = new Link{"Freia",0,norse_gods};
norse_gods->succ->prev = norse_gods;
```

Мы создали этот список, создавая структуры **Link** и соединяя их так, как показано на рисунке: сначала — Тор, за ним следует Один, являющийся предшественником Тора, а завершает список Фрея — предшественница Одина. Следуя указателям, можно убедиться, что мы правы и каждый указатель **succ** и **prev** ссылается на правильного бога. Однако этот код мало понятен, так как мы не определили явно и не именовали операцию вставки.

```
Link* insert(Link* p, Link* n) // Вставка n перед p (фрагмент)
{
    n->succ = p;          // p следует после n
    p->prev->succ = n;    // n следует после предшественника p
    n->prev = p->prev;    // Предшественник p становится
                          // предшественником n
    p->prev = n;          // n становится предшественником p
    return n;
}
```

Этот фрагмент программы работает, если указатель **p** действительно указывает на объект типа **Link** и этот объект действительно имеет предшественника. Убедитесь, что это именно так. Размышляя об указателях и связанных структурах, таких как список, состоящий из объектов типа **Link**, мы практически всегда рисуем на бумаге диаграммы, состоящие из прямоугольников и стрелок, чтобы проверить программу на небольших примерах. Пожалуйста, не пренебрегайте этим эффективным средством.

Приведенная версия функции **insert()** неполна, поскольку в ней не предусмотрен случай, когда указатели **n**, **p** или **p->prev** равны **nullptr**. Добавив соответствующие проверки, мы получим немного более сложный, но зато правильный вариант функции **insert**.

```
Link* insert(Link* p, Link* n) // Вставка n перед p; возврат n
{
    if (n==nullptr) return p;
    if (p==nullptr) return n;
    n- >succ = p;         // p следует за n
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;    // Предшественник p становится
                          // предшественником n
    p->prev = n;          // n становится предшественником p
```

```
    return n;
}
```

Теперь мы можем написать такой код:

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Freia"});
```

Теперь все возможные неприятности, чреватые ошибками и связанные с указателями **prev** и **succ**, спрятаны. Проверка корректности указателей очень утомительна и подвержена ошибкам, поэтому ее *обязательно* следует скрывать в хорошо спроектированных и тщательно проверенных функциях. В частности, многие ошибки в программах возникают оттого, что программисты забывают проверять, не равен ли указатель **nullptr**, — как это было (преднамеренно) продемонстрировано в первой версии функции **insert()**.

Обратите внимание на то, что мы использовали аргументы по умолчанию (см. разделы 15.3.1, A.9.2), чтобы освободить пользователей от необходимости указывать предшествующие и последующие элементы в каждом вызове конструктора.

17.9.4. Операции над списками

Стандартная библиотека содержит класс **list**, который будет описан в разделе 20.4. Он скрывает все операции со связями, но в данном разделе мы самостоятельно разработаем список, основанный на классе **Link**, чтобы узнать, что скрывается "под покровом" стандартного списка, и продемонстрировать еще несколько примеров использования указателей.

Какие операции необходимы нашему классу **Link**, чтобы позволить пользователю избежать ошибок, связанных с указателями? В некотором смысле это дело вкуса, но мы все же приведем некоторый полезный набор.

- Конструктор.
- **insert**: вставка перед элементом.
- **add**: вставка после элемента.
- **erase**: удаление элемента.
- **find**: поиск узла с заданным значением.
- **advance**: переход к *n*-му последующему узлу.

Эти операции можно написать следующим образом:

```
Link* add(Link* p, Link* n) // Вставка n после p; возврат n
{
    // Почти такая же, как insert (см. упр. 11)
}
```

```
Link* erase(Link* p)          // Удаление *p из списка;
{                             // возврат преемника p
    if (p==nullptr) return nullptr;
    if (p->succ) p->succ->prev = p->prev;
    if (p->prev) p->prev->succ = p->succ;
    return p->succ;
}

Link* find(Link* p, const string& s) // Поиск s в списке;
                      // возврат nullptr, если не найден
{
    while (p) {
        if (p->value == s) return p;
        p = p->succ;
    }
    return nullptr;
}

Link* advance(Link* p, int n) // Переход на n позиций в списке
// Возвращает nullptr, если это невозможно
// Положительное n – движение вперед; отрицательное – назад
{
    if (p==nullptr) return nullptr;
    if (0<n) {
        while (n--) {
            if (p->succ == nullptr) return nullptr;
            p = p->succ;
        }
    }
    else if (n<0) {
        while (n++) {
            if (p->prev == nullptr) return nullptr;
            p = p->prev;
        }
    }
    return p;
}
```

Обратите внимание на использование постфиксного инкремента `n++`. Он подразумевает, что сначала используется текущее значение переменной, а затем оно увеличивается на единицу.

17.9.5. Использование списков

В качестве небольшого примера создадим два списка.

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Zeus"});
norse_gods = insert(norse_gods,new Link{"Freia"});
```

```
Link* greek_gods = new Link{"Hera"};
greek_gods = insert(greek_gods,new Link{"Athena"});
greek_gods = insert(greek_gods,new Link{"Mars"});
greek_gods = insert(greek_gods,new Link{"Poseidon"});
```

"К сожалению", мы наделали много ошибок: Зевс — греческий бог, а не норвежский, греческий бог войны — Арес, а не Марс (Марс — это его римское имя). Эти ошибки можно исправить следующим образом:

```
Link* p = find(greek_gods, "Mars");
if (p) p->value = "Ares";
```

Обратите внимание на проверку, возвращает ли функция `find()` значение `nullptr`. Мы, конечно, уверены, что этого не может быть (в конце концов, мы только что вставили имя Марса в список `greek_gods`), но в реальности что-то могло произойти не так, как ожидалось.

Аналогично можно перенести Зевса в его Пантеон.

```
Link* p = find(norse_gods,"Zeus");
if (p) {
    erase(p);
    insert(greek_gods,p);
}
```

Вы заметили ошибку? Она довольно тонкая (конечно, если вы не работаете со списками непосредственно). Что если на удаленный с помощью функции `erase()` узел указывал указатель `norse_gods`? Разумеется, на самом деле этого не было, но в жизни бывает всякое, и хорошая программа должна это учитывать.

```
Link* p = find(norse_gods, "Zeus");
if (p) {
    if (p==norse_gods) norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods,p);
}
```

Заодно мы исправили и вторую ошибку: вставляя Зевса *перед* первым греческим богом, мы должны установить на него указатель списка. Указатели — чрезвычайно полезный и гибкий, но очень тонкий инструмент.

В заключение распечатаем наш список.

```
void print_all(Link* p)
{
    cout << "{ ";
    while (p) {
        cout << p->value;
        if (p=p->succ)cout << ", ";
    }
    cout << " }";
}
```

```
print_all(norse_gods);
cout<<"\n";

print_all(greek_gods);
cout<<"\n";
```

Результат должен быть следующим:

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

17.10. Указатель this

Обратите внимание на то, что каждая из функций, работающих со списком, получает в качестве первого аргумента указатель **Link*** для доступа к данным, хранящимся в этом объекте. Такие функции мы часто делаем членами класса. Можно ли упростить класс **Link** (или использование списка), сделав соответствующие операции членами класса? Может быть, сделать указатели закрытыми, чтобы только функции-члены класса могли обращаться к ним? Попробуем.

```
class Link {
public:
    string value;
    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        :value{v}, prev{p}, succ{s} {}

    Link* insert(Link* n); // Вставляет n перед данным объектом
    Link* add(Link* n);    // Вставляет n после данного объекта
    Link* erase();         // Удаляет данный объект из списка
    Link* find(const string& s);              // Поиск в списке
    const Link* find(const string& s) const; // Поиск в
                    // константном списке (см. раздел 18.5.1)
    Link* advance(int n) const;               // Перемещение
                    // на n позиций по списку
    Link* next() const { return succ; }
    Link* previous() const { return prev; }
private:
    Link* prev;
    Link* succ;
};
```

Этот фрагмент выглядит многообещающе. Мы определили операции, не изменяющие состояние объекта класса **Link**, как константные функции-члены. Мы также добавили (не модифицирующие) функции **next()** и **previous()**, чтобы пользователи могли перемещаться по списку, — поскольку непосредственный доступ к указателям **succ** и **prev** теперь запрещен. Мы оставили значение узла открытым, потому что (пока) у нас нет причин его скрывать; ведь это "просто данные".

Попробуем теперь реализовать функцию **Link::insert()**, скопировав ранее разработанную глобальную функцию и модифицировав ее.

```
Link* Link::insert(Link* n)    // Вставка n перед p; возврат n
{
    Link* p = this;            // Указатель на данный объект
    if (n==nullptr) return p; // Ничего не вставляем
    if (p==nullptr) return n; // Ничего не вставляем
    n->succ = p;               // p следует за n
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;         // Предшественник p становится
                               // предшественником n
    p->prev = n;               // n предшествует p
    return n;
}
```

Но как получить указатель на объект, для которого была вызвана функция **Link::insert()**? Без помощи языка это сделать невозможно. Однако в каждой функции-члене идентификатор **this** представляет собой указатель на объект, для которого она вызывается. В качестве альтернативы можно просто использовать **this** вместо **p**.

```
Link* Link::insert(Link* n) // Вставка n перед this; возврат n
{
    if (n==nullptr) return this;
    if (this==nullptr) return n;
    n->succ = this;          // this следует за n
    if (this->prev) this->prev->succ = n;
    n->prev = this->prev;    // Предшественник this становится
                             // предшественником n
    this->prev = n;          // n предшествует this
    return n;
}
```

Код становится немного более многословным, но при обращении к члену текущего объекта **this** можно не упоминать, поэтому код можно сократить.

```
Link* Link::insert(Link* n) // Вставка n перед this; возврат n
{
    if (n==nullptr) return this;
    if (this==nullptr) return n;
    n->succ = this;          // this следует за n
    if (prev) prev->succ = n;
    n->prev = prev;          // Предшественник this становится
                             // предшественником n
    prev = n;                // n предшествует this
    return n;
}
```

Другими словами, при каждом обращении к члену класса происходит неявное использование указателя **this**. Единственная ситуация, в которой

его необходимо упомянуть явно, возникает, когда нужно обратиться ко всему объекту.

Обратите внимание на то, что указатель **this** имеет специфический смысл: он указывает на объект, для которого вызывается функция-член. Он не указывает на какой-то из ранее использованных объектов. Компилятор гарантирует, что мы не сможем изменить значение указателя **this** в функции-члене. Рассмотрим пример.

```
struct S {
    // ...
    void mutate(S* p)
    {
        this = p; // Ошибка: this не допускает изменений
        // ...
    }
};
```

17.10.1. Еще об использовании списков

Поработав над реализацией, посмотрим, как теперь выглядит использование списка.

```
Link* norse_gods = new Link{"Thor"};
norse_gods = norse_gods->insert(new Link{"Odin"});
norse_gods = norse_gods->insert(new Link{"Zeus"});
norse_gods = norse_gods->insert(new Link{"Freia"});

Link* greek_gods = new Link{"Hera"};
greek_gods = greek_gods->insert(new Link{"Athena"});
greek_gods = greek_gods->insert(new Link{"Mars"});
greek_gods = greek_gods->insert(new Link{"Poseidon"});
```

Это очень похоже на предыдущий код. Как и раньше, исправим наши "ошибки". Например, укажем правильное имя бога войны.

```
Link* p = greek_gods->find("Mars");
if (p) p->value = "Ares";
```

Перенесем Зевса в его Пантеон.

```
Link* p2 = norse_gods->find("Zeus");
if (p2) {
    if (p2==norse_gods) norse_gods = p2->next();
    p2->erase();
    greek_gods = greek_gods->insert(p2);
}
```

И наконец выведем список на печать.

```
void print_all(Link* p)
{
```

```
    cout << "{ ";
    while (p) {
        cout << p->value;
        if (p=p->next()) cout << ", ";
    }
    cout << " }";
}

print_all(norse_gods);
cout<<"\n";

print_all(greek_gods);
cout<<"\n";
```

В итоге получим тот же результат, что и ранее:

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

Какая из этих версий лучше: та, в которой функция `insert()` и другие являются функциями-членами, или та, в которой они не принадлежат классу? В данном случае это не имеет особого значения, но вспомните, что было написано в разделе 9.7.5.

Следует отметить, что мы создали не класс списка, а только класс узла. В результате мы вынуждены следить за тем, какой из указателей является указателем на первый элемент. Можно улучшить наш код, определив класс `List`, но продемонстрированный выше дизайн класса является очень распространенным. Стандартный класс `list` рассматривается в разделе 20.4.

Задание

Это задание состоит из двух частей. Первая часть упражнений должна дать вам представление о динамических массивах и их отличии от класса `vector`.

1. Разместите в динамической памяти массив, состоящий из десяти чисел типа `int`, используя оператор `new`.
2. Выведите в поток `cout` значения этих десяти чисел типа `int`.
3. Освободите память, занятую массивом (используя оператор `delete[]`).
4. Напишите функцию `print_array10(ostream& os, int* a)`, выводящую в поток `os` значения из массива `a` (содержащего десять элементов).
5. Разместите в динамической памяти массив, состоящий из десяти чисел типа `int`; инициализируйте его значениями 100, 101, 102 и т.д.; выведите эти значения на печать.

6. Разместите в динамической памяти массив, состоящий из одиннадцати чисел типа `int`; инициализируйте его значениями 100, 101, 102 и т.д.; выведите эти значения на печать.
7. Напишите функцию `print_array(ostream& os, int* a, int n)`, выводящую в поток `os` значения массива `a` (содержащего `n` элементов).
8. Разместите в динамической памяти массив, состоящий из двадцати чисел типа `int`; инициализируйте его значениями 100, 101, 102 и т.д.; выведите эти значения на экран.
9. Вы не забыли удалить массивы? (Если забыли, сделайте это сейчас.)
10. Выполните задания 5, 6 и 8, используя класс `vector`, а не массив, и функцию `print_vector()` вместо функции `print_array()`.

Вторая часть задания посвящена указателям и их связи с массивами. Используйте функцию `print_array()` из последнего задания.

1. Разместите в динамической памяти переменную типа `int`, инициализируйте ее значением 7 и присвойте ее адрес переменной `p1`.
2. Выведите на печать значения указателя `p1` и переменной типа `int`, на которую он указывает.
3. Разместите в динамической памяти массив, состоящий из семи чисел типа `int`; инициализируйте его значениями 1, 2, 4, 8 и т.д.; присвойте адрес массива переменной `p2`.
4. Выведите на печать значение указателя `p2` и массив, на который он ссылается.
5. Объявите указатель типа `int*` с именем `p3` и инициализируйте его значением указателя `p2`.
6. Присвойте указатель `p1` указателю `p2`.
7. Присвойте указатель `p3` указателю `p2`.
8. Выведите на печать значения указателей `p1` и `p2`, а также то, на что они указывают.
9. Освободите всю память, которую вы ранее использовали.
10. Разместите в динамической памяти массив, состоящий из десяти чисел типа `int`; инициализируйте их числами 1, 2, 4, 8 и т.д.; присвойте его адрес указателю `p1`.
11. Разместите в динамической памяти массив, состоящий из десяти чисел типа `int`, присвойте его адрес указателю `p2`.
12. Скопируйте значения из массива, на который указывает указатель `p1`, в массив, на который указывает указатель `p2`.
13. Повторите задания 10–12, используя класс `vector`, а не массив.

Контрольные вопросы

1. Зачем нужны структуры данных с переменным количеством элементов?
2. Назовите четыре вида памяти, используемой при работе типичной программы.
3. Что такое динамическая память? Как еще ее называют? Какие операторы поддерживают работу с ней?
4. Что такое оператор разыменования и зачем он нужен?
5. Что такое адрес? Как язык C++ работает с адресами?
6. Какую информацию об объекте несет указатель, который на него указывает? Какую полезную информацию он не содержит?
7. На что может указывать указатель?
8. Что такое утечка памяти?
9. Что такое ресурс?
10. Как можно инициализировать указатель?
11. Что такое нулевой указатель? Зачем он нужен?
12. Когда нужен указатель (а не ссылка или именованный объект)?
13. Что такое деструктор? Когда он нужен?
14. Зачем нужен виртуальный деструктор?
15. Как вызываются деструкторы членов класса?
16. Что такое приведение типов? Когда оно необходимо?
17. Как получить доступ к члену класса с помощью указателя?
18. Что такое двусвязный список?
19. Что собой представляет ключевое слово `this` и когда оно используется?

Термины

`delete`
`delete[]`
`new`
`nullptr`
`this`
`void*`
адрес
взятие адреса: **`&`**
виртуальный деструктор
выделение памяти
деструктор
деструктор члена
диапазон
динамическая память
доступ к члену: **`->`**
индекс
индексирование: **`[]`**
контейнер
нулевой указатель
освобождение
память
преобразование типа
приведение
разыменование
связь
содержимое памяти: **`*`**
список
указатель
утечка памяти
утечка ресурсов

Упражнения

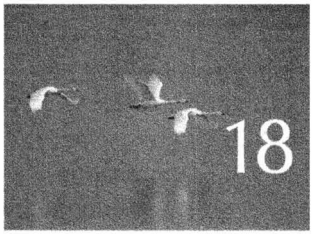
1. Какой формат вывода значений указателей в вашей реализации языка? *Указание*: не читайте документацию.
2. Сколько байтов занимают типы `int`, `double` и `bool`? Ответьте на вопрос, не используя оператор `sizeof`, разве что для проверки вашего ответа.
3. Напишите функцию `void to_lower(char* s)`, заменяющую все прописные символы в С-строке `s` их строчными эквивалентами. Например, строка `"Hello, World!"` должна принять вид `"hello, world!"`. Не используйте стандартные библиотечные функции. С-строка представляет собой массив символов, который завершается нулем, поэтому если вы обнаружите символ `0`, то это значит, что вы находитесь в конце массива.
4. Напишите функцию `char* strdup(const char*)`, копирующую С-строку в выделенную в динамической памяти строку. Не используйте стандартные библиотечные функции.
5. Напишите функцию `char* findx(const char* s, const char* x)`, находящую первое вхождение С-строки `x` в строку `s`.
6. В этой главе ничего не говорилось о том, что произойдет, если, используя оператор `new`, вы столкнетесь с нехваткой памяти. Это называется *исчерпанием памяти* (memory exhaustion). Выясните, что случится. У вас есть две очевидные альтернативы: обратиться к документации или написать программу с бесконечным циклом, в котором происходит многократное выделение памяти и не выполняется ее освобождение. Попробуйте оба варианта. Сколько примерно памяти вы сможете выделить, пока она не исчерпается?
7. Напишите программу, считывающую символы из потока `cin` в массив, выделенный в динамической памяти. Читайте отдельные символы, пока не встретится восклицательный знак (`!`). Не используйте класс `std::string` и не беспокойтесь об исчерпании памяти.
8. Выполните упр. 7 еще раз, но теперь считывайте символы в строку `std::string`, а не в динамическую память (класс `string` знает, как использовать динамическую память).
9. Как увеличивается стек: вверх (в сторону старших адресов) или вниз (в сторону младших адресов)? В каком направлении изначально (пока не было выполнено ни одного освобождения) растет выделяемая память? Напишите программу, позволяющую выяснить это.
10. Посмотрите на решение упр. 7. Может ли ввод вызвать переполнение массива; иначе говоря, можете ли вы ввести больше символов, чем

выделено памяти (это серьезная ошибка)? Что произойдет, если вы попытаетесь ввести больше символов, чем выделено памяти?

11. Завершите программу из раздела 17.10.1, создающую "список богов".
12. Зачем нужны две версии функции `find()`?
13. Модифицируйте класс `Link` из раздела 17.10.1, чтобы он хранил значение `struct God`. Этот тип должен иметь члены типа `string`: имя, мифология, транспортное средство и оружие. Например, `God{"Зевс", "Греция", "", "Молния"}` или `God{"Один", "Норвегия", "Восьминогий летающий конь Слейпнер", "Копье Гунгнир"}`. Напишите функцию `print_all()`, построчно выводящую имена богов и их атрибуты. Добавьте функцию-член `add_ordered()`, размещающую новый элемент, создаваемый с помощью оператора `new`, в правильной лексикографической позиции. Используя объекты `Link` со значениями типа `God`, составьте список богов из трех мифологий; затем переместите элементы (богов) из этого списка в три лексикографически упорядоченных списка — по одному для каждой мифологии.
14. Можно ли написать список богов из раздела 17.10.1 в виде односвязного списка; другими словами, можно ли удалить член `prev` из класса `Link`? По каким причинам мы могли бы это сделать? В каких ситуациях разумно использовать односвязные списки? Переделайте такой пример с использованием только односвязного списка.

Послесловие

Зачем возиться с такими низкоуровневыми механизмами, как указатель и динамическая память, вместо того чтобы просто воспользоваться классом `vector`? Ну, один из ответов состоит в том, что кто-то же написал класс `vector` и аналогичные абстракции, поэтому хотелось бы знать, как это можно сделать. Существуют языки программирования, не содержащие указателей и не имеющие проблем, связанных с низкоуровневым программированием. В основном программисты, работающие с такими языками, перепоручают решение задач, связанных с непосредственным доступом к аппаратному обеспечению, программистам, работающим на языке C++ (или на других языках, пригодных для низкоуровневого программирования). Однако нам кажется, что главная причина заключается в том, что невозможно понять компьютер и программирование, не зная, как программа взаимодействует с физическими устройствами. Люди, ничего не знающие об указателях, адресах памяти и так далее, часто имеют неверные представления о возможностях языка программирования, с которым они работают; такие заблуждения приводят к созданию программ, которые "почему-то не работают".



18

Векторы и массивы

Да будет покупатель бдителен!
Добрый совет

В этой главе показано, как копировать векторы и обращаться к их элементам с помощью индексов. Для этого мы обсуждаем копирование в целом и рассматриваем связь вектора с низкоуровневым понятием массива. Мы демонстрируем также связь массивов с указателями и анализируем проблемы, возникающие вследствие этой связи. В главе также рассматриваются пять важнейших операций, которые должны быть предусмотрены для любых типов: конструктор, конструктор по умолчанию, копирующий конструктор, копирующее присваивание и деструктор. Контейнеры, кроме того, должны иметь перемещающий конструктор и перемещающее присваивание.

18.1. Введение

Для того чтобы подняться в воздух, самолет должен разогнаться до скорости взлета. Пока самолет грохочет по взлетной полосе, он представляет собой не более чем тяжелый и неуклюжий грузовик. Однако, поднявшись в воздух, самолет становится совершенно иным — элегантным и эффективным транспортным средством. Это объясняется тем, что в воздухе самолет находится в своей стихии.

В этой главе мы находимся на середине взлетной полосы, цель которой — с помощью возможностей и технологий языка программирования избавиться от ограничений и сложностей, связанных с использованием памяти компьютера. Мы стремимся достичь той стадии программирования, на которой типы обладают именно теми свойствами, которые основаны на логических потребностях. Для этого мы должны преодолеть фундаментальные ограничения, связанные с аппаратным обеспечением, такие как

- объект в памяти имеет фиксированный размер;
- объект в памяти находится в конкретном месте;
- компьютер предоставляет только самые необходимые операции над объектами (например, копирование слова, сложение двух слов и т.п.).

По существу, эти ограничения относятся к встроенным типам и операциям языка C++ (и унаследованы от языка C; см. раздел 22.2.5 и главу 27). В главе 17 мы уже познакомились с типом `vector`, управляющим доступом ко всем своим элементам и обеспечивающим операции, которые выглядят

"естественными" с точки зрения пользователя, но не с точки зрения аппаратного обеспечения.

В этой главе мы сосредоточим свое внимание на копировании. Это важное, но скорее техническое понятие. Что мы имеем в виду под копированием нетривиального объекта? До какой степени копии являются независимыми после выполнения операции копирования? Какие операции копирования существуют? Как их специфицировать? Как они связаны с другими фундаментальными операциями, например с инициализацией и очисткой?

Мы обязательно обсудим проблему работы с памятью без помощи высокоуровневых типов, таких как **vector** и **string**, изучим массивы и указатели, их взаимосвязь и способы применения, а также ловушки, связанные с их использованием. Это важная информация для любого программиста, вынужденного работать с низкоуровневыми кодами, написанными на языке C++ или C.

Обратим ваше внимание на то, что детали класса **vector** характерны для векторов и способов построения новых высокоуровневых типов из низкоуровневых в языке программирования C++. Однако любой высокоуровневый тип (**string**, **vector**, **list**, **map** и др.) в любом языке создается из одинаковых машинных примитивов и отражает разнообразие решений фундаментальных проблем, описанных в этой главе.

18.2. Инициализация

Рассмотрим наш **vector**, каким он был в конце главы 17:

```
class vector {
    int sz;              // Размер
    double* elem;        // Указатель на элементы
public:
    vector(int s)        // Конструктор
    :sz{s},
    elem{new double[s]} // Выделение памяти
    { /* ... */ }
    ~vector()            // Деструктор
    { delete[] elem; }   // Освобождение памяти
    // ...
};
```

Выглядит неплохо, но что если мы захотим инициализировать вектор множеством значений, отличающихся от значения по умолчанию? Например, так:

```
vector v1 = {1.2, 7.89, 12.34 };
```

Это можно сделать, и это гораздо лучше, чем сначала выполнять инициализацию значениями по умолчанию, а затем присваивать значения, которые нам нужны:

```
vector v2(2); // Утомительно и подвержено ошибкам
v2[0] = 1.2;
v2[1] = 7.89;
v2[2] = 12.34;
```

По сравнению с **v1**, "инициализация" **v2** представляется утомительной и подверженной ошибкам (мы сознательно указали неверное количество элементов в этом фрагменте кода). Применение функции **push_back()** спасает нас от необходимости указывать размер.

```
vector v3;    // Утомительно и многословно
v2.push_back(1.2);
v2.push_back(7.89);
v2.push_back(12.34);
```

Но этот код все равно остается многословным. Так как же нам написать конструктор, который принимает в качестве аргумента список инициализации? Список элементов типа **T** в фигурных скобках **{}** для программиста представляет собой объект типа **initializer_list<T>** из стандартной библиотеки, который является ни чем иным, как списком элементов типа **T**, так что мы можем написать следующий исходный текст.

```
class vector {
    int sz;       // Размер
    double* elem; // Указатель на элементы
public:
    vector(int s) // Конструктор (s — количество элементов)
    :sz{s}, elem{new double[sz]} // Неинициализированная память
    {
        for (int i = 0; i<sz; ++i)
            elem[i] = 0.0;       // Инициализация
    }
                 // Конструктор со списком инициализации
    vector(initializer_list<double> lst)
    :sz{lst.size()},
    elem{new double[sz]}         // Неинициализированная память
    {
        // Инициализация с помощью std::copy(); (раздел Б.5.2)
        copy( lst.begin(),lst.end(),elem);
    }
    // ...
};
```

Мы использовали алгоритм **copy** из стандартной библиотеки (§Б.5.2). Он копирует последовательность элементов, определяемую первыми двумя аргументами (в данном случае — начало и конец **initializer_list**) в последовательность элементов, начало которой указывает его третий аргумент (в данном случае — последовательность элементов вектора, начинающаяся с **elem**).

Теперь мы можем записать

```
vector v1 = {1,2,3}; // Три элемента 1.0, 2.0, 3.0
vector v2(3);        // Три элемента со значениями по умолчанию (0.0)
```

Обратите внимание, что мы использовали круглые скобки для количества элементов, а фигурные — для списков элементов. Нам нужна запись, которая позволяла бы различать их, например:

```
vector v1{3}; // Один элемент со значением 3.0
vector v2(3); // Три элемента со значениями по умолчанию 0.0
```

Это не слишком элегантно, но эффективно. Если у компилятора есть выбор, то значение в фигурных скобках {} он трактует как значение элемента и передает его конструктору как элемент `initializer_list`.

В большинстве случаев — включая все случаи в данной книге — знак равенства = перед списком инициализации {} не является обязательным, так что мы можем писать

```
vector v11 = {1,2,3}; // Три элемента 1.0, 2.0, 3.0
vector v12 {1,2,3};   // Три элемента 1.0, 2.0, 3.0
```

Разница в данном случае чисто стилевая, и является делом вкуса.

Обратите внимание, что мы передаем `initializer_list<double>` по значению. Это сделано сознательно и требуется правилами языка: тип `initializer_list` по сути является обработчиком для элементов, находящихся "где-то" (см. раздел §Б.6.4).

18.3. Копирование

Вернемся к нашему незавершенному вектору.

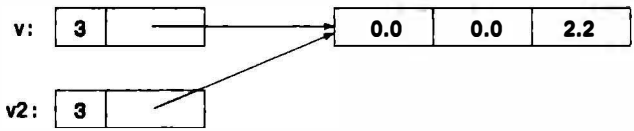
```
class vector {
    int sz;              // Размер
    double* elem;        // Указатель на элементы
public:
    vector(int s)        // Конструктор
    :sz{s},
    elem{new double[s]} // Выделение памяти
    { /* ... */ }
    ~vector()            // Деструктор
    { delete[] elem; }   // Освобождение памяти
    // ...
};
```

Попробуем скопировать один из таких векторов.

```
void f(int n)
{
    vector v(3);   // Определяем вектор из трех элементов
    v.set(2,2.2);  // Устанавливаем v[2] равным 2.2
    vector v2 = v; // Что здесь происходит?
    // ...
}
```

В идеале объект **v2** должен стать копией объекта **v** (т.е. оператор = создает копии); иначе говоря, должны выполняться условия **v2.size()==v.size()** и **v2[i]==v[i]** для всех **i** в диапазоне [**0:v.size()**). Кроме того, при выходе из функции **f()** вся выделенная память освобождается. Именно это (разумеется) и делает класс **vector** из стандартной библиотеки, но не наш все еще слишком простой класс **vector**. Наша цель — улучшить наш класс **vector** так, чтобы правильно действовать в таких ситуациях, но сначала попытаемся понять, как на самом деле работает наша текущая версия. Что именно она делает неправильно, как и почему? Поняв это, мы сможем устранить проблему. Еще более важно то, что мы сможем распознать аналогичные проблемы, которые могут возникнуть в других ситуациях.

По умолчанию копирование класса означает "скопировать все данные-члены". Это часто говорится в некотором идеальном смысле. Например, мы копируем объект класса **Point**, копируя его координаты. Однако при копировании членов класса, являющихся указателями, возникают проблемы. В частности, для векторов в нашем примере выполняются условия **v.sz==v2.sz** и **v.elem==v2.elem**, так что наши векторы выглядят следующим образом:



Иначе говоря, объект **v2** не содержит копии элементов объекта **v**; он владеет ими совместно с объектом **v**. Мы могли бы написать следующий код.

```
v.set(1,99);  // Устанавливаем v[1] равным 99
v2.set(0,88); // Устанавливаем v2[0] равным 88
cout << v.get(0) << ' ' << v2.get(1);
```

В результате мы получили бы на экране **88 99**. Но это не то, к чему мы стремились. Если бы не существовало скрытой связи между объектами **v** и **v2**, то вывод имел бы вид **0 0**, поскольку мы не записывали никаких значений в ячейку **v[0]** или **v2[1]**. Вы могли бы возразить, что такое поведение является "интересным", "изящным" или "иногда полезным", но мы ждали не этого, и это не то, что реализовано в стандартном классе **vector**. Кроме того, когда мы вернем результат из функции **f()**, произойдет явная катастрофа. При этом неявно будут вызваны деструкторы объектов **v** и **v2**; деструктор объекта **v** освободит использованную память с помощью инструкции

```
delete[] elem;
```

Но то же самое сделает и деструктор объекта **v2**! Поскольку в обоих объектах, **v** и **v2**, указатель **elem** указывает на одну ту же ячейку памяти, эта память будет освобождена дважды, что может привести к катастрофическим результатам (см. раздел 17.4.6).

18.3.1. Копирующие конструкторы

Итак, что нам надо делать? Это очевидно: необходимо предусмотреть операцию копирования, которая копировала бы элементы и вызывалась при инициализации одного вектора другим.

Инициализация объектов класса выполняется в конструкторе. Таким образом, нам нужен конструктор, создающий копии. Такой конструктор, что не удивительно, называется *копирующим* (copy constructor) или конструктором копирования. В качестве аргумента он принимает ссылку на объект, который подлежит копированию (т.е. копирующий конструктор для класса **vector** должен выглядеть следующим образом: **vector(const vector&);**). Этот конструктор будет вызываться, когда мы попытаемся инициализировать один объект класса **vector** другим. Мы передаем объект по ссылке, поскольку не хотим (и это очевидно) копировать аргумент конструктора, который определяет копирование... Мы передаем эту ссылку со спецификатором **const**, потому что не хотим модифицировать аргумент (см. раздел 8.5.6). Итак, уточним определение класса **vector**.

```
class vector {
    int sz;
    double* elem;
public:
    vector(const vector&) ;  // Копирующий конструктор
    // ...
};
```

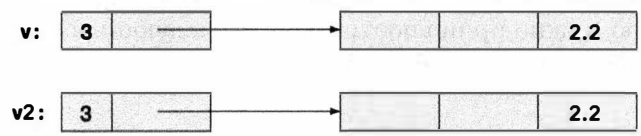
Копирующий конструктор устанавливает количество элементов (**sz**) и выделяет память для элементов (инициализируя указатель **elem**) перед копированием значений элементов из аргумента.

```
vector::vector(const vector& arg)
// Размещает элементы, инициализирует их копированием
    :sz{arg.sz}, elem{new double[arg.sz]}
{
    copy(arg,arg+sz,elem); // std::copy(); см. раздел §Б.5.2
}
```

При наличии такого конструктора мы можем вернуться к рассмотренному выше примеру.

```
vector v2 = v;
```

Это определение инициализирует объект **v2**, вызывая копирующий конструктор класса **vector** с аргументом **v**. Теперь в рассмотренном ранее примера с вектором с тремя аргументами мы получим следующий результат.



В итоге деструкторы смогут корректно удалить каждое из множества элементов. Очевидно, что два объекта класса **vector** теперь не зависят друг от друга, и мы можем изменять значения элементов в объекте **v**, не влияя на содержание объекта **v2**, и наоборот.

```
v.set(1,99);  // Устанавливаем v[1] равным 99
v2.set(0,88); // Устанавливаем v2[0] равным 88
cout << v.get(0) << ' ' << v2.get(1);
```

На экран выводится `0 0`.

Вместо инструкции

```
vector v2 = v;
```

можно написать

```
vector v2{v};
```

Когда объекты **v** (инициализатор) и **v2** (инициализируемая переменная) имеют один и тот же тип и в этом типе определена операция копирования, приведенные выше инструкции эквивалентны, и то, какую из них вы используете, зависит от ваших личных предпочтений.

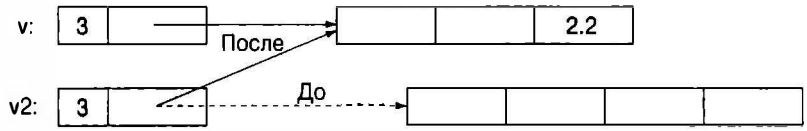
18.3.2. Копирующее присваивание

Копирование векторов может осуществляться не только при инициализации, но и при присваивании. Как и инициализация, копирование по умолчанию производится поэлементно, что вновь может привести к двойному удалению (см. раздел 18.3.1) и утечке памяти. Рассмотрим пример.

```
void f2(int n)
{
    vector v(3);    // Определяем вектор
    v.set(2,2.2);
    vector v2(4);
    v2 = v;         // Присваивание: что здесь происходит?
    // ...
}
```

Мы бы хотели, чтобы вектор **v2** был копией вектора **v** (именно так функционирует стандартный класс **vector**), но поскольку в нашем классе **vector** о присваивании ничего не сказано, будет использоваться присваивание по умолчанию; иначе говоря, выполняется почленное присваивание, и члены **sz** и **elem** объекта **v2** становятся идентичными членам **sz** и **elem** объекта **v** соответственно.

Эту ситуацию можно проиллюстрировать следующим образом:



При выходе из функции **f2()** возникнет такая же катастрофичная ситуация, как и при выходе из функции **f()** в разделе 18.3 до того, как мы определили копирующий конструктор: элементы, на которые указывают члены обоих векторов (**v** и **v2**), будут удалены дважды с помощью оператора **delete[]**. Кроме того, возникнет утечка памяти, первоначально выделенной для элементов вектора **v2**, хранящего четыре элемента. Их просто некому удалить. Решение этой проблемы по сути не отличается от решения проблемы копирующей инициализации (см. раздел 18.3.1). Определим корректно работающее копирующее присваивание.

```
class vector {
    int sz;
    double* elem;
public:
    // Копирующее присваивание:
    vector& operator=(const vector&) ;
    // ...
};

vector& vector::operator=(const vector& a)
// Делаем данный вектор копией a
{
    double* p = new double[a.sz]; // Выделение памяти
    copy(a.elem,a.elem+a.sz,p);   // Копирование элементов
    delete[] elem;                // Освобождение памяти
    elem = p;                     // Переназначение указателя
    sz = a.sz;
    return *this; // Возврат ссылки на себя (см. раздел 17.10)
}
```

Присваивание немного сложнее, чем создание, поскольку мы имеем дело с уже готовым старым вектором со своими элементами. Наша основная стратегия состоит в копировании элементов из источника класса **vector**.

```
double* p = new double[a.sz]; // Выделение памяти
copy(a.elem,a.elem+a.sz,p);   // Копирование элементов
```

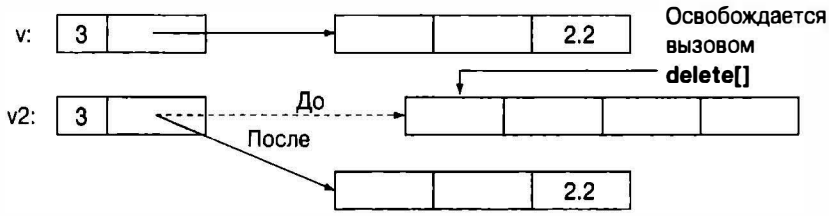
Затем мы освобождаем старые элементы из целевого объекта класса **vector**:

```
delete[] elem;                // Освобождение памяти
```

В заключение мы переназначаем указатель **elem** на новые элементы.

```
elem = p;                     // Переназначение указателя
sz = a.sz;
```

Результат можно проиллюстрировать следующим образом.



Теперь утечка памяти в классе **vector** устранена, и каждая выделенная память освобождается только один раз.

При реализации присваивания код можно упростить, освобождая занятую старыми элементами память до создания копии, но обычно лучше не стирать информацию, пока вам нечем ее заменить. Кроме того, если вы это сделаете, при попытке присвоить объект класса **vector** самому себе могут начаться неприятности.

```
vector v(10);
v=v;          // Самоприсваивание
```

Всегда убеждайтесь, что ваша реализация обрабатывает эту ситуацию если не оптимально, то по крайней мере корректно.

18.3.3. Терминология, связанная с копированием

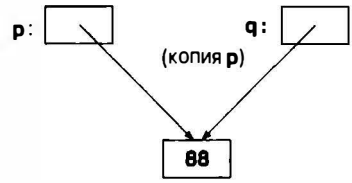
Копирование встречается в большинстве программ и языков программирования. Основная проблема при этом заключается в том, что именно копируется: указатель (или ссылка) или информация, на которую он указывает (или, соответственно, ссылается).

- *Поверхностное копирование* (shallow copy) копирует только указатель, поэтому в результате на один и тот же объект могут указывать два указателя. Этот механизм копирования лежит в основе работы указателей и ссылок.
- *Глубокое копирование* (deep copy) выполняет копирование информации, на которую указывает указатель, так что в результате два указателя указывают на разные объекты. Именно так работает копирование, реализованное в классах **vector**, **string** и т.п. Если мы хотим, чтобы для наших классов выполнялось глубокое копирование объектов, в них необходимо определить корректные копирующий конструктор и оператор копирующего присваивания.

Рассмотрим пример поверхностного копирования.

```
int* p = new int{77};
int* q = p;       // Копируем указатель p
*p = 88;          // Изменяем значение int, на которое
                  // указывают указатели p и q
```

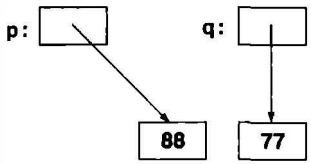
Эту ситуацию можно проиллюстрировать следующим образом.



В противоположность можно реализовать глубокое копирование.

```
int* p = new int{77};
int* q = new int{*p}; // Выделение нового int, затем копирование
                      // значения, на которое указывает p
*p = 88;              // Изменение значения int, на которое
                      // указывает указатель p
```

Эту ситуацию можно проиллюстрировать так.



Используя эту терминологию, можно сказать, что проблема в нашем исходном классе **vector** заключается в том, что мы выполняли поверхностное копирование и не копировали элементы, на которые указывал указатель **elem**. Наш усовершенствованный класс **vector**, как и класс **vector** из стандартной библиотеки, выполняет глубокое копирование, выделяя новую память для элементов и копируя их значения. О типах, предусматривающих мелкое копирование (таких, как указатели и ссылки), говорят, что они имеют *семантику указателей* (pointer semantics) или *ссылок* (reference semantics), поскольку копируют адреса. О типах, осуществляющих глубокое копирование (таких, как **string** и **vector**), говорят, что они имеют *семантику значений* (value semantics), т.е. копируют значения, на которые указывают. С точки зрения пользователя, типы с семантикой значений ведут себя так, как будто никакие указатели не используются, а существуют только значения, которые можно копировать. С точки зрения копирования их можно рассматривать как "работающие, как обычные числа".

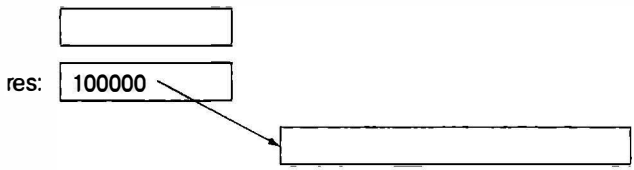
18.3.4. Перемещение

Если вектор содержит большое количество элементов, их копирование может оказаться дорогостоящей операцией. Поэтому мы должны копировать векторы только тогда, когда это необходимо. Рассмотрим пример.

```
vector fill(istream& is)
{
    vector res;
```

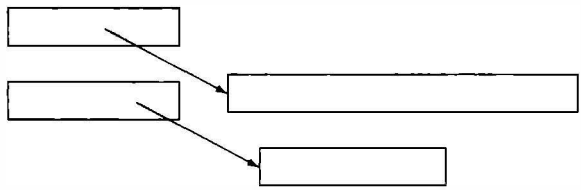
```
    for (double x; is>>x; ) res.push_back(x);
    return res;
}

void use()
{
    vector vec = fill(cin);
    // ... Используем vec ...
}
```

Здесь мы заполняем локальный вектор **res** данными из входного потока и возвращаем его функции **use()**. Копирование **res** при выходе из **fill()** и в вектор **vec** могут быть весьма дорогостоящими операциями. Но почему мы хотим именно копировать? Нам это не нужно! Мы никогда не будем использовать исходный вектор **res** после возврата. Фактически уничтожение **res** представляет собой часть возврата из функции **fill()**. Но как мы можем избежать копирования? Давайте вновь обратимся к размещению вектора в памяти:



Нам бы хотелось "украсть" представление **res** для использования в **vec**. Другими словами, мы бы хотели, чтобы **vec** использовал элементы **res**, не выполняя копирования.

Вектор **res** после перемещения указателя на элементы и их счетчика из **res** в **vec** больше не хранит элементы. Значения из **res** при выходе из **fill()** успешно перемещаются в **vec**. Теперь **res** может быть уничтожен (просто и эффективно) без каких-либо нежелательных побочных эффектов:



Мы успешно переместили 100 тысяч значений типа **double** на выходе из **fill()** и передали их в вызывающую функцию ценой четырех присваиваний одного слова.

Как же выразить это в коде на языке программирования C++? В дополнение к операциям копирования мы определяем операции перемещения:

```
class vector {
    int sz;
    double* elem;
```

```
public:
    vector(vector&& a);           // Перемещающий конструктор
    vector& operator=(vector&&); // Перемещающее присваивание
    // ...
};
```

Забавная запись `&&` означает "ссылка на rvalue". Мы используем ее, чтобы определить перемещающие операции. Обратите внимание, что такие операции не получают константные аргументы, т.е. мы пишем `(vector&&)`, а не `(const vector&&)`. При перемещении необходимо изменить исходный объект, делая его "пустым". Определения перемещающих операций обычно достаточно просты, проще и эффективнее их копирующих эквивалентов. Для вектора мы получаем следующий код.

```
vector::vector(vector&& a)
:sz{a.sz}, elem{a.elem} // Копируем elem и sz из a
{
    a.sz = 0;           // Делаем вектор a пустым
    a.elem = nullptr;
}

vector& vector::operator=(vector&& a) // Перемещаем a в текущий вектор
{
    delete[] elem;      // Освобождение старой памяти
    elem = a.elem;      // Копирование elem и sz из a
    sz = a.sz;
    a.elem = nullptr;   // Делаем вектор a пустым
    a.sz = 0;
    return *this;       // Возврат ссылки на себя (см. раздел 17.10)
}
```

Определяя перемещающий конструктор, мы делаем легким и дешевым перемещение больших объемов информации, таких как вектор со многими элементами. Вернемся к нашему примеру.

```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
```

Для реализации возврата неявно используется перемещающий конструктор. Компилятору известно, что возвращаемая локальная переменная (`res`) выходит из области видимости, так что ее можно перемещать, а не копировать.

Важность перемещающих конструкторов заключается в том, что нам не надо работать с указателями или ссылками для передачи большого количества информации из функции. Рассмотрим следующую альтернативу — не без изъянов, но традиционную.

```
vector* fill2(istream& is)
{
    vector* res = new vector;
    for (double x; is>>x; ) res->push_back(x);
    return res;
}

void use2()
{
    vector* vec = fill(cin);
    // ... Используем vec ...
    delete vec;
}
```

В этом случае мы должны не забыть удалить **vector**. Как описано в разделе 17.4.6, удаление объектов, размещенных в динамической памяти, не так уж просто выполнять согласованно и корректно, как могло бы показаться.

18.4. Основные операции

Настал момент, когда мы можем приступить к обсуждению того, какие конструкторы должен иметь класс, должен ли он содержать деструктор и требуется ли реализовать копирующие и перемещающие операции. Следует рассмотреть семь важных операций.

- Конструкторы с одним или несколькими аргументами.
- Конструктор по умолчанию.
- Копирующий конструктор (копирование объектов одинаковых типов).
- Копирующее присваивание (копирование объектов одинаковых типов).
- Перемещающий конструктор (перемещение объектов одинаковых типов).
- Перемещающее присваивание (перемещение объектов одинаковых типов).
- Деструктор.

Обычно нам нужен один или несколько конструкторов, получающих аргументы, необходимые для инициализации объекта, например:

```
string s{"cat.jpg"};      // Инициализация s символьной строкой "cat.jpg"
Image ii{Point{200,300}, // Инициализация Point координатами {200,300},
         "cat.jpg"};     // с последующим выводом содержимого
                         // файла cat.jpg в указанной точке
```

Смысл и использование инициализатора полностью определяются конструктором. Стандартный конструктор класса **string** использует в качестве начального значения символьную строку, а стандартный конструктор

класса **vector** в качестве параметра получает количество элементов. Обычно конструктор используется для установки инварианта (см. раздел 9.4.3). Если мы не в состоянии определить хороший инвариант для класса, который может установить его конструктор, то, вероятно, мы плохо спроектировали класс или структуру данных.

Конструкторы, имеющие аргументы, сильно зависят от класса, в котором они реализованы. Остальные операции имеют более или менее стандартную структуру.

Как понять, что классу необходим конструктор по умолчанию? Он требуется тогда, когда мы хотим создавать объекты класса без указания инициализатора. Наиболее распространенный пример такой ситуации возникает, когда мы хотим поместить объекты класса в стандартный контейнер, имеющий тип **vector**. Приведенные ниже инструкции работают только потому, что для типов **int**, **string** и **vector<int>** существуют значения, предусмотренные по умолчанию.

```
vector<double> vi(10); // Вектор из 10 элементов типа double,
                       // каждый из них инициализирован 0.0
vector<string> vs(10); // Вектор из 10 элементов типа string,
                       // каждый из них инициализирован ""
vector<vector<int>> vvi(10); // Вектор из 10 векторов,
                       // каждый из них инициализирован
                       // значением vector{}
```

Таким образом, иметь конструктор по умолчанию часто полезно. Возникает следующий вопрос: а когда именно стоит иметь конструктор по умолчанию? Ответ: когда мы можем установить инвариант класса с осмысленным и очевидным значением по умолчанию. Для числовых типов, таких как **int** и **double**, очевидным значением является **0** (для типа **double** оно принимает вид **0.0**). Для типа **string** очевидным выбором является пустая строка **""**. Для класса **vector** можно использовать пустой вектор. Если тип **T** имеет значение по умолчанию, то оно имеет вид **T{}**. Например, **double{}** равно **0.0**, **string{}** равно **""**, а **vector<int>{}** — это пустой **vector**, предназначенный для хранения переменных типа **int**.

Если класс захватывает ресурсы, то он должен иметь деструктор. Ресурс — это то, что вы "где-то взяли" и должны вернуть, когда закончите его использовать. Очевидным примером является память, выделенная с помощью оператора **new**, которую вы должны освободить, используя оператор **delete** или **delete[]**. Для хранения элементов нашему классу **vector** выделяется память; поскольку она должна быть освобождена, класс должен иметь деструктор. Другие ресурсы, которые используются в более сложных программах, — это файлы (если вы открыли файл, то должны его закрыть), блокировки, дескрипторы потоков или сокеты (используемые для обеспечения взаимосвязи между процессами и удаленными компьютерами).

Еще один признак того, что в классе необходим деструктор, — это наличие членов класса, которые являются указателями или ссылками. Если одним из членов класса является указатель или ссылка, скорее всего, в нем требуются деструктор и операции копирования.

Класс, который должен иметь деструктор, практически всегда требует копирующий конструктор и копирующее присваивание. Причина состоит в том, что если объект захватывает ресурс (и имеет член класса — указатель, указывающий на этот ресурс), то копирование по умолчанию (почленное поверхностное копирование) почти наверняка приведет к ошибке. И вновь классическим примером такой ситуации является класс **vector**.

Аналогично класс, который нуждается в деструкторе, практически всегда требует также наличия перемещающего конструктора и перемещающего присваивания. Причина состоит в том, что если объект захватывает ресурс (и имеет член класса — указатель, указывающий на этот ресурс), то копирование по умолчанию (почленное поверхностное копирование) почти наверняка приведет к ошибке, а обычный подход (копирующие операции, дублирующие все состояние объекта) может оказаться дорогостоящим. Здесь также классическим примером является класс **vector**.

Кроме того, если производный класс может иметь деструктор, то базовый класс должен иметь виртуальный деструктор (см. раздел 17.5.2).

18.4.1. Явные конструкторы

Конструктор, имеющий один аргумент, определяет преобразование типа этого аргумента в свой класс. Это может оказаться очень полезным. Рассмотрим пример.

```
class complex {
public:
    complex(double);    // Преобразование double в complex
    complex(double,double);
    // ...
};

complex z1 = 3.18;      // OK: преобразует 3.18 в (3.18,0)
complex z2 = complex{1.2, 3.4};
```

Однако неявные преобразования следует применять редко и осторожно, поскольку они могут вызвать неожиданные и нежелательные эффекты. Например, наш класс **vector**, определенный выше, имеет конструктор, принимающий аргумент типа **int**, откуда следует, что он определяет преобразование типа **int** в класс **vector**:

```
class vector {
    // ...
```

```
    vector(int);
    // ...
};

vector v = 10;          // Создаем вектор из 10 double Что? Присваиваем
v = 20;                 // вектору v новый вектор из 20 double
void f(const vector&);
f(10);                  // Что? Вызываем функцию f с новым
                        // вектором из 10 double
```

Кажется, мы получили куда больше, чем хотели. К счастью, подавить такое неявное преобразование довольно просто. Конструктор с ключевым словом **explicit** допускает только обычную семантику конструирования и не допускает неявные преобразования. Рассмотрим пример.

```
class vector {
    // ...
    explicit vector(int);
    // ...
};

    vector v = 10; // Ошибка: преобразования int в vector нет
    v = 20;        // Ошибка: преобразования int в vector нет
    vector v0(10); // OK

    void f(const vector&);
    f(10);         // Ошибка: преобразования int в vector нет
    f(vector(10)); // OK
```

Для того чтобы избежать неожиданных преобразований, мы — и стандарт языка — определили конструктор класса **vector** с одним аргументом как **explicit**. Очень жаль, что все конструкторы не имеют спецификатора **explicit** по умолчанию; если вы сомневаетесь, объявляйте конструктор, который может быть вызван с одним аргументом, с использованием модификатора **explicit**.

18.4.2. Отладка конструкторов и деструкторов

Конструкторы и деструкторы вызываются в точно определенных и предсказуемых местах программы. Однако мы не всегда пишем явные вызовы, например **vector(2)**; чаще мы объявляем объект класса **vector** и передаем его как аргумент функции по значению или создаем вектор в динамической памяти с помощью оператора **new**. Это может вызвать замешательство у людей, думающих в терминах синтаксиса. Не существует единственной синтаксической конструкции, которая вызывала бы конструктор. О конструкторах и деструкторах проще думать следующим образом.

- Когда создается объект класса **X**, вызывается один из его конструкторов.
- Когда уничтожается объект типа **X**, вызывается его деструктор.

Деструктор вызывается всегда, когда уничтожается объект класса; это происходит, когда объект выходит из области видимости, программа завершает работу или к указателю на объект применяется оператор **delete**. Некоторый подходящий конструктор вызывается каждый раз, когда создается объект класса; это происходит при инициализации переменной, при создании объекта с помощью оператора **new** (за исключением встроенных типов), а также при копировании объекта.

Но когда это все происходит? Для того чтобы прочувствовать это, добавим в конструкторы, операции присваивания и деструкторы инструкции вывода. Рассмотрим пример.

```
struct X { // Простой тестовый класс
    int val;

    void out(const string& s, int nv)
    {
        cerr << this << "->" << s << ": " << val
             << " (" << nv << ")\n";
    }
    X(){ out("X()",0); val=0; } // Конструктор по умолчанию
    X(int v) { val=v; out( "X(int)",v); }
                                // Копирующий конструктор:
    X(const X& x){ val=x.val; out("X(X&) ",x.val); }
    X& operator=(const X& a)    // Копирующее присваивание
    {
        out("X::operator=()",a.val);
        val=a.val; return *this;
    }
    ~X() { out("~X()",0); }     // Деструктор
};
```

Проследим, что происходит при выполнении операций над объектом класса **X**.

```
X glob(2);              // Глобальная переменная

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
```

```
    X loc {4};              // Локальная переменная
    X loc2 {loc};           // Копирующий конструктор
    loc = X{5};             // Копирующее присваивание
    loc2 = copy(loc);       // Передача по значению и возврат
    loc2 = copy2(loc);
    X loc3 {6};
    X& r = ref_to(loc); // Передача по ссылке и возврат
    delete make(7);
    delete make(8);
    vector<X> v(4);         // Значения по умолчанию
    XX loc4;
    X* p = new X{9};        // X в динамической памяти
    delete p;
    X* pp = new X[5];       // Массив X в динамической памяти
    delete[] pp;
}
```

Попробуйте выполнить эту программу.

ПОПРОБУЙТЕ

Мы имеем в виду следующее: выполните эту программу и убедитесь, что понимаете результаты ее работы. Если вы их действительно понимаете, то знаете почти все, что требуется знать о создании и уничтожении объектов.

В зависимости от качества вашего компилятора вы можете заметить "отсутствующие копии", связанные с вызовами функций `copy()` и `copy2()`. Мы (люди) видим, что эти функции ничего не делают; они просто возвращают переданное значение без каких-либо изменений. Если компилятор настолько хорош, что заметит это, то сможет удалить эти вызовы копирующего конструктора. Другими словами, компилятор может считать, что конструктор копирования только копирует объекты и не выполняет никаких побочных действий. Некоторые компиляторы достаточно "умны", чтобы исключить фиктивные копии. Однако не гарантируется, что все компиляторы настолько интеллектуальны, так что, если вы хотите добиться переносимой эффективности, подумайте о применении перемещающих операций (раздел 18.3.4).

Зачем мы возимся с этим "глупым классом `X`"? Это напоминает упражнения для пальцев, которые выполняют музыканты. После этих упражнений многие вещи, обладающие намного большим смыслом, становятся понятнее и легче. Кроме того, если у вас возникнут проблемы с конструкторами и деструкторами, вы сможете вставить в них операторы вывода и посмотреть, как они работают. Для более крупных программ такая отладка становится утомительной, но и для них применимы подобные методы. Например, можно узнать, происходит ли утечка памяти, определив, равна ли

нулю разность между количеством вызовов конструкторов и деструкторов. Часто источником проблем является то, что программист забыл определить копирующие конструкторы и копирующее присваивание для классов, выделяющих память или содержащих указатели на объекты. Впрочем, эти проблемы легко устранить.

Если ваши проблемы слишком большие, чтобы разобраться с ними с помощью таких простых средств, освойте профессиональные средства поиска таких проблем; они называются "детекторами утечек" (leak detectors). В идеале, разумеется, следует не устранять утечки, а программировать так, чтобы они вообще не возникали.

18.5. Доступ к элементам вектора

До сих пор (см. раздел 17.6) для доступа к элементам вектора мы использовали функции-члены **set()** и **get()**. Но этот способ слишком громоздок и некрасив. Мы бы хотели использовать обычную индексацию: **v[i]**. Для этого следует определить функцию-член с именем **operator[]**. Вот первая (наивная) попытка это сделать.

```
class vector {
    int sz;                 // Размер
    double* elem;           // Указатель на элементы
public:
    // ...
    double operator[](int n)
        { return elem[n]; } // Возврат элемента
};
```

Все выглядит хорошо и просто, но, к сожалению, слишком просто. Разрешив оператору индексирования (**operator[]()**) возвращать значение, мы разрешили чтение, но не запись элементов.

```
vector v(10);
int x = v[2]; // Хорошо
v[3] = x;     // Ошибка: v[3] не является lvalue
```

Здесь выражение **v[i]** интерпретируется как вызов оператора **v.operator[](i)**, который возвращает значение **i**-го элемента вектора **v**. Для такого слишком наивного варианта класса **vector** значение **v[3]** является числом с плавающей точкой, а не переменной, содержащей число с плавающей точкой.

ПОПРОБУЙТЕ

Создайте вариант класса **vector**, скомпилируйте его и посмотрите на сообщение об ошибке, которое ваш компилятор выдаст для инструкции **v[3]=x;**.

В следующей версии мы разрешим оператору **operator[]** возвращать указатель на соответствующий элемент:

```
class vector {
    int sz;                  // Размер
    double* elem;            // Указатель на элементы
public:
    // ...
    double* operator[](int n)
        { return &elem[n]; } // Возврат указателя
};
```

При таком определении мы можем записывать элементы:

```
vector v(10);
for (int i=0; i<v.size(); ++i)
{   // Работает, но по-прежнему некрасиво
    *v[i] = i;
    cout << *v[i];
}
```

Здесь выражение **v[i]** интерпретируется как вызов **v.operator[](i)** и возвращает указатель на **i**-й элемент вектора **v**. Проблема в том, что теперь мы должны писать оператор * для разыменования указателя, указывающего на данный элемент. Это столь же некрасиво, как и функции **set()** и **get()**. Проблему можно устранить, если оператор индексирования будет возвращать ссылку.

```
class vector {
    // ...
    double& operator[](int n)
    { return elem[n]; }     // Возврат ссылки
};
```

Теперь можно написать следующий исходный текст.

```
vector v(10);
for (int i=0; i<v.size(); ++i) { // Работает!
    v[i] = i;                    // v[i] возвращает ссылку
                                 // на i-й элемент
    cout << v[i];
}
```

Мы обеспечили работоспособность традиционного обозначения: выражение **v[i]** интерпретируется как вызов оператора **v.operator[](i)** и возвращает ссылку на **i**-й элемент вектора **v**.

18.5.1. Константная перегрузка

Оператор **operator[]()**, определенный выше, имеет один недостаток: его нельзя вызвать для константного вектора:

```
void f(const vector& cv)
{
    double d = cv[1]; // Ошибка, хотя должно работать
    cv[1] = 2.0;      // Ошибка (как и должно быть)
}
```

Причина заключается в том, что наша функция-член **vector::operator[]()** потенциально может изменять объект класса **vector**. На самом деле она этого не делает, но компилятор-то об этом не знает, потому что мы забыли сообщить ему об этом. Для того чтобы решить эту проблему, необходимо предусмотреть версию оператора с модификатором **const** (см. раздел 9.7.4). Это легко сделать.

```
class vector {
    // ...
    double& operator[](int n);       // Для неконстантных векторов
    double  operator[](int n) const; // Для константных векторов
};
```

Очевидно, что мы не можем вернуть ссылку типа **double&** из версии с модификатором **const**, поэтому возвращаем значение типа **double**. С таким же успехом мы могли бы вернуть и константную ссылку **const double&**, но, поскольку объект типа **double** невелик, возвращать ссылку смысла не имеет (см. раздел 8.5.6), так что мы решили возвращать значение. Теперь можно написать следующий код.

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1]; // Отлично (константный вариант [])
    cv[1] = 2.0;      // Ошибка  (константный вариант [])
    double d = v[1];  // Отлично (неконстантный вариант [])
    v[1] = 2.0;       // Отлично (неконстантный вариант [])
}
```

Поскольку объекты класса **vector** часто передаются по константной ссылке, такая версия оператора **operator[]()** с модификатором **const** является существенным дополнением к нашему классу.

18.6. Массивы

До сих пор мы использовали слово *массив* (array) для последовательности объектов, расположенных в динамической памяти. Массивы можно также выделять где угодно как именованные переменные. Фактически массивы часто используются следующим образом.

- Как глобальные переменные (правда, использование глобальных переменных часто является плохой идеей).
- Как локальные переменные (однако в этом случае имеются серьезные ограничения на использование массивов).

- Как аргументы функции (но массив не знает своего размера).
- Как члены класса (хотя массивы, являющиеся членами класса, может быть трудно инициализировать).

Возможно, вы заметили, что мы отдаем существенное предпочтение классу **vector** по сравнению с массивами. Класс **std::vector** следует использовать при любой возможности. Однако массивы существовали задолго до появления векторов и являлись их приблизительным прототипом во многих языках (особенно в языке C), поэтому их следует хорошо знать, чтобы иметь возможность работать со старыми программами или с программами, написанными людьми, не признающими преимущества класса **vector**.

Так что же такое массив? Как его определить и как использовать? *Массив* — это однородная последовательность объектов, расположенных в смежных ячейках памяти; иначе говоря, все элементы массива имеют один и тот же тип и между ними нет промежутков. Элементы массива нумеруются в порядке возрастания начиная с нуля. В объявлении массив указывается с помощью квадратных скобок.

```
const int max = 100;
int gai[max]; // Глобальный массив (из 100 чисел типа int); "живет вечно"
void f(int n)
{
     char lac[20];    // локальный массив; "живет" до
                      // выхода из области видимости
     int lai[60];
     double lad[n];   // Ошибка: размер массива не является константой
     // ...
}
```

Обратите внимание на ограничение: количество элементов именованного массива должно быть известно на этапе компиляции. Если мы хотим, чтобы количество элементов массива было переменным, то должны разместить его в динамической памяти и обращаться к нему через указатель. Именно так поступает класс **vector** со свои массивом элементов.

Как и к элементам массивов, размещенным в динамической памяти, доступ к элементам именованных массивов осуществляется с помощью операторов индексирования и разыменования (**[]** и *******):

```
void f2()
{
    char lac[20];    // Локальный массив; "живет" до
                     // выхода из области видимости
    lac[7] = 'a';
    *lac = 'b';      // Эквивалент инструкции lac[0]='b'
    lac[-2] = 'b';   // ??
    lac[200] = 'c'; // ??
}
```

Эта функция компилируется, но, как мы знаем, это еще не гарантия корректной работы. Использование оператора `[]` очевидно, но проверка выхода за пределы допустимого диапазона отсутствует, поэтому, хотя функция `f2()` и компилируется, результат записи `lac[-2]` и `lac[200]` обычно приводит к катастрофе (как при любом выходе за пределы допустимого диапазона). Не делайте этого. Массивы не выполняют проверку на выход за пределы допустимого диапазона. Здесь нам снова приходится непосредственно работать с физической памятью, так что на "системную поддержку" не рассчитывайте.

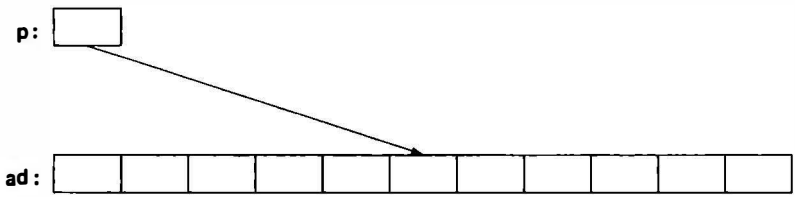
А не мог ли компилятор как-то увидеть, что массив `lac` содержит только двадцать элементов, так что выражение `lac[200]` — это ошибка? В принципе, мог бы, но, насколько нам известно, в настоящее время не существует ни одного такого компилятора. Дело в том, что отследить границы массива на этапе компиляции в общем случае невозможно, а перехват простейших ошибок (таких, как приведены выше) не решает всех проблем.

18.6.1. Указатели на элементы массива

Указатель может ссылаться на элемент массива. Рассмотрим пример.

```
double ad[10];
double* p = &ad[5]; // Указывает на элемент ad[5]
```

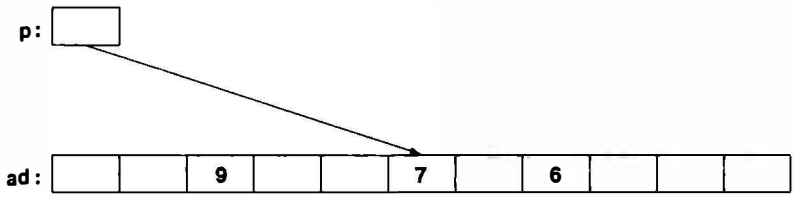
Указатель `p` ссылается на переменную типа `double`, известную как `ad[5]`.



Этот указатель можно индексировать и разыменовывать.

```
*p =7;
p[2] = 6;
p[-3] = 9;
```

Теперь ситуация выглядит следующим образом.

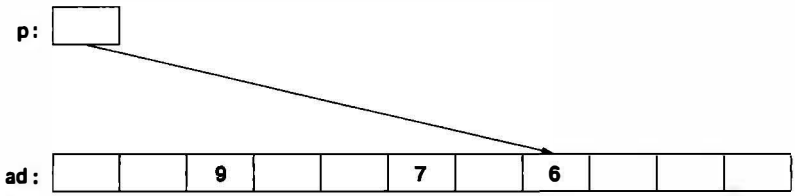


Таким образом, мы можем индексировать указатель как положительными, так и отрицательными значениями. До тех пор, пока результаты не выходят за пределы допустимого диапазона, эти выражения являются корректными. Однако выход за пределы допустимого диапазона является некорректным (как и в случае массивов, размещенных в динамической памяти; см. раздел 17.4.3). Как правило, выход за пределы массива компилятором не распознается и (раньше или позже) приводит к неприятностям.

Если указатель указывает на элемент внутри массива, то для его перенаправления на другой элемент можно использовать операции сложения и вычитания.

```
p += 2; // Переносим указатель p на два элемента вправо
```

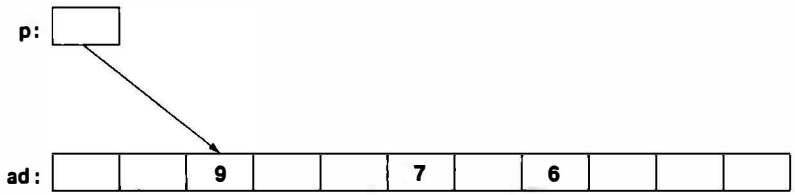
Теперь ситуация выглядит следующим образом.



Аналогично при вычитании

```
p -= 5; // Переносим указатель p на пять элементов влево
```

получаем следующую ситуацию.



Использование операций +, –, += и –= для переноса указателей называется *арифметикой указателей* (pointer arithmetic). Очевидно, поступая так, мы должны проявлять большую осторожность, чтобы не выйти за пределы массива.

```
p += 1000;      // Абсурд: p указывает в массив, содержащий
                // только 10 чисел
double d = *p; // Некорректно: возможно неправильное значение
                // (определенно совершенно непредсказуемое)
*p = 12.34;     // Некорректно: вероятно, будут испорчены
                // некоторые неизвестные данные
```

К сожалению, не все серьезные ошибки, связанные с арифметикой указателей, легко обнаружить. Лучше всего просто избегать использования арифметики указателей.

Наиболее распространенным использованием арифметики указателей является инкремент указателя с помощью оператора ++ для указания на следующий элемент массива и декремент с помощью оператора -- для ссылки на предыдущий элемент. Например, мы могли вы вывести элементы массива **ad** следующим образом:

```
for(double* p = &ad[0]; p<&ad[10]; ++p) cout << *p << '\n';
```

Или в обратном порядке:

```
for(double* p = &ad[9]; p>=&ad[0]; --p) cout << *p << '\n';
```

Такое использование арифметики указателей не слишком широко распространено. По нашему мнению, последний ("обратный") пример небезопасен. Почему **&ad[9]**, а не **&ad[10]**? Почему **>=**, а не **>**? Эти примеры были бы одинаково хороши (и одинаково эффективны), если бы мы использовали индексацию. Кроме того, они были бы совершенно эквивалентны при использовании класса **vector**, в котором проверка выхода за пределы допустимого диапазона осуществляется проще.

Отметим, что в большинстве реальных программ арифметика указателей связана с передачей указателя в качестве аргумента функции. В этом случае компилятор не знает, на какое количество элементов ссылается указатель, и вы должны следить за этим сами. Такой ситуации следует избегать всеми силами.

Почему в языке C++ вообще разрешена арифметика указателей? Ведь это так хлопотно и не дает ничего нового по сравнению с тем, что можно сделать с помощью индексирования:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "Невозможно!\n";
```

В основном это произошло по историческим причинам. Эти правила были разработаны для языка C десятилетия назад, и отменить их, не выбросив в мусорную корзину огромное количество программ, невозможно. Частично это объясняется тем, что арифметика указателей обеспечивает определенное удобство в некоторых низкоуровневых приложениях, например в механизме управления памятью.

18.6.2. Указатели и массивы

Имя массива используется для обращения ко всем его элементам:

```
char ch[100];
```

Размер массива **ch**, т.е. значение **sizeof(ch)**, равен 100. Однако имя массива легко и без замечаний превращается в указатель:

```
char* p = ch;
```

Здесь указатель `p` инициализируется адресом `&ch[0]`, а размер `sizeof(p)` равен чему-то около 4 (в зависимости от компьютера, но в любом случае не 100!).

Это свойство может быть полезным. Например, рассмотрим функцию `strlen()`, подсчитывающую количество символов в массиве, который завершается нулевым символом.

```
int strlen(const char* p) // Аналогична стандартной
                          // функции strlen()
{
    int count = 0;
    while (*p) { ++count; ++p; }
    return count;
}
```

Теперь можем вызвать ее как с аргументом `strlen(ch)`, так и с аргументом `strlen(&ch[0])`. Возможно, вы заметили, что такое обозначение дает очень небольшое преимущество, и мы с вами согласны.

Одна из причин, по которым имена массивов могут превращаться в указатели, состоит в желании избежать передачи большого объема данных по значению. Рассмотрим пример.

```
int strlen(const char a[]) // Аналогична стандартной
                           // функции strlen()
{
    int count = 0;
    while (a[count]) { ++count; }
    return count;
}

char lots [100000];

void f()
{
     int nchar = strlen(lots);
     // ...
}
```

Наивно (но вполне обоснованно) мы могли бы ожидать, что при выполнении этого вызова будут скопированы 100 тыс. символов, переданных в качестве аргумента функции `strlen()`, но этого не происходит. Вместо этого объявление аргумента `char p[]` рассматривается как эквивалент `char*p`, а вызов `strlen(lots)` — как эквивалент вызова `strlen(&lots[0])`. Это предотвращает затратное копирование, но должно вас удивить. Почему вы должны удивиться? Да потому, что в любой другой ситуации, если вы при передаче объекта не потребуете явно, чтобы он передавался по ссылке (см. разделы 8.5.3–8.5.6), этот объект будет скопирован.

Обратите внимание на то, что указатель, полученный из имени массива, указывает на его первый элемент и не является переменной, так что ему ничего нельзя присвоить.

```
char ac[10];
ac = new char [20];      // Ошибка: присваивание чего-либо
                         // имени массива запрещено
&ac[0] = new char [20]; // Ошибка: значению указателя ничего
                         // присвоить нельзя
```

И напоследок — проблема, с которой компилятор может справиться.

Вследствие неявного превращения имени массива в указатель мы не можем даже скопировать массивы с помощью оператора присваивания.

```
int x[100];
int y[100];
// ...
x = y;            // Ошибка
int z[100] = y;   // Ошибка
```

Это логично, но часто неудобно. Чтобы скопировать массив, вы должны написать более сложный код, например:

```
for (int i=0; i<100; ++i)
    x[i]=y[i];                  // Копируем 100 чисел типа int
memcpy(x,y,100*sizeof(int));    // Копируем 100*sizeof(int) байт
copy(y,y+100,x);                // Копируем 100 чисел типа int
```

Поскольку в языке C нет векторов, в нем интенсивно используются массивы. Вследствие этого в огромном количестве программ, написанных на языке C++, используются массивы (подробнее об этом — в разделе 27.1.2). В частности, строки в стиле C (массивы символов, завершаемые нулем; эта тема рассматривается в разделе 27.5) распространены очень широко.

Если хотите выполнить присваивание, то используйте нечто вроде класса **vector** из стандартной библиотеки. Код копирования объектов класса **vector**, эквивалентный приведенному выше, можно записать следующим образом:

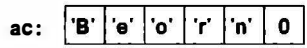
```
vector<int> x(100);
vector<int> y(100);
// ...
x = y; // Копируем 100 чисел типа int
```

18.6.3. Инициализация массива

Массив символов типа **char** может быть инициализирован с помощью строкового литерала, например:

```
char ac[] = "Beorn"; // Массив из шести символов
```

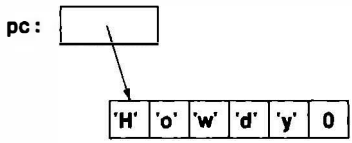
Подсчитайте эти символы. Их пять, но **ac** становится массивом из шести символов, потому что компилятор добавляет завершающий нуль в конце строкового литерала.



Строка, завершающаяся нулевым символом, представляет собой обычное явление в языке C и многих системах. Такие массивы символов, завершающиеся нулем, мы называем *строками в стиле языка C* (C-style string) или просто C-строками. Все строковые литералы являются строками в стиле языка C.

```
char* pc = "Howdy"; // Указатель pc указывает на массив
                    // из шести символов
```

Графически это можно изобразить следующим образом.



Переменная типа **char**, имеющая числовое значение 0, — это не символ **'0'** или какая-то иная буква или цифра. Цель завершающего нулевого символа — помочь функции найти конец строки. Помните: массив не знает своего размера. Полагаясь на соглашение о завершающем нуле, мы можем написать следующий код:

```
int strlen(const char* p) // Аналог стандартной функции strlen()
{
    int n = 0;
    while (p[n]) ++n;
    return n;
}
```

На самом деле мы не обязаны определять функцию **strlen()**, поскольку это стандартная библиотечная функция, определенная в заголовочном файле **<string.h>** (разделы 27.5 и Б.11.3). Обратите внимание, что функция **strlen()** подсчитывает символы, но игнорирует завершающий нулевой символ; иначе говоря, для хранения n символов в строке в стиле языка C необходимо иметь память для хранения $n+1$ переменной типа **char**.

С помощью литеральных строк можно инициализировать только символьные массивы, но любой массив можно инициализировать списком значений, тип которых соответствует типу элементов массива. Рассмотрим пример.

```
int ai[] = {1, 2, 3, 4, 5, 6};          // Массив из шести чисел типа int
int ai2[100] = {0,1,2,3,4,5,6,7,8,9}; // Остальные 90 элементов
                                        // инициализируются нулями
```

```
double ad[100] = {};      // все элементы инициализированы нулями
char chars[] = {'a', 'b', 'c'};        // Завершающего нуля нет!
```

Обратите внимание на то, что количество элементов в массиве **ai** равно шести (а не семи), а количество элементов в массиве **chars** равно трем (а не четырем), — правило "добавить нуль в конце" относится только к литеральным строкам. Если размер массива не задан явно, он определяется по списку инициализации (это весьма полезное правило). Если количество элементов в списке инициализации окажется меньше, чем количество элементов массива (как в определениях массивов **ai2** и **ad**), остальные элементы инициализируются значениями, предусмотренными для данного типа элементов по умолчанию.

18.6.4. Проблемы с указателями

Как и массивами, указателями часто злоупотребляют. Люди часто сами создают себе проблемы, используя указатели и массивы. В частности, все серьезные проблемы, связанные с указателями, включают попытки доступа к области памяти, которая не является объектом ожидаемого типа, причем многие из этих проблем, в свою очередь, вызваны выходом за пределы массива. Здесь мы рассмотрим такие проблемы.

- Обращение по нулевому указателю.
- Обращение по неинициализированному указателю.
- Выход за пределы массива.
- Обращение к удаленному объекту.
- Обращение к объекту, вышедшему из области видимости.

На практике во всех перечисленных ситуациях главная проблема, стоящая перед программистом, заключается в том, что внешне фактический доступ выглядит вполне невинно; просто указатель ссылается на неправильное значение. Что еще хуже (при записи с помощью указателя), проблема может проявиться намного позднее, когда окажется, что некий объект, не связанный с данным кодом, был поврежден. Рассмотрим следующие примеры.

Не обращайтесь к памяти с помощью нулевого указателя.

```
int* p = nullptr;
*p = 7;     // Ой!
```

Очевидно, что когда это обычно происходит в реальной программе, между инициализацией и использованием указателя размещен какой-то код. Чаще всего эта ошибка возникает при передаче указателя **p** функции или при его получении в результате работы функции. Мы рекомендуем никуда не передавать нулевой указатель, но, если вы должны это делать, проверьте указатель перед его использованием:

```
int* p = fct_that_can_return_a_nullptr();
if (p == nullptr) {
    // Что-то делаем
}
else {
    // Используем p
    *p = 7;
}
```

и

```
void fct_that_can_receive_a_nullptr(int* p)
{
    if (p == nullptr) {
        // Что-то делаем
    }
    else {
        // Используем p
        *p = 7;
    }
}
```

Основными средствами, позволяющими избежать ошибок, связанных с нулевыми указателями, являются ссылки (см. раздел 17.9.1), а для уведомления об ошибках — исключения (см. разделы 5.6 и 19.5).

Инициализируйте указатели.

```
int* p;
*p = 9;    // Ой!
```

В частности, не забывайте инициализировать указатели, являющиеся членами класса.

Не обращайтесь к несуществующим элементам массива.

```
int a[10];
int* p = &a[10];
*p = 11;       // Ой!
a[10] = 12;    // Ой!
```

Будьте осторожны с первым и последним элементами в циклах и постарайтесь не передавать массивы с помощью указателей на их первые элементы. Вместо этого используйте класс **vector**. Если вам действительно необходимо использовать массив в нескольких функциях (передавая его как аргумент), будьте особенно осторожны и передавайте вместе с указателем размер массива.

Не обращайтесь к памяти с помощью удаленного указателя.

```
int* p = new int(7);
// ...
delete p;
// ...
*p = 13;    // Ой!
```

Инструкция `delete p` или код, размещенный после нее, может неосторожно обратиться к значению `*p` или использовать его косвенно. Изо всех рассматриваемых ситуаций труднее всего систематически обеспечить отсутствие данной ситуации. Наиболее эффективной защитой от нее является запрет на использование "голых" операторов `new`, требующих выполнения "голых" операторов `delete`: выполняйте операторы `new` и `delete` в конструкторах и деструкторах или используйте контейнеры, такие как `Vector_ref` (раздел Д.4).

Не возвращайте указатель на локальную переменную.

```
int* f()
{
    int x = 7;
    // ...
    return &x;
}

// ...

int* p = f();
// ...
*p = 15;   // Ой!
```

Возврат из функции `f()` или код, размещенный после него, может неосторожно обратиться к значению `*p` или использовать его косвенно. Проблема заключается в том, что локальные переменные, объявленные в функции, размещаются в стеке при входе в функцию и удаляются оттуда при выходе из нее. В частности, если локальной переменной является объект класса, то вызывается его деструктор (см. раздел 17.5.1). Компиляторы способны распознать большинство проблем, связанных с возвратом указателей на локальные переменные, но делают это только некоторые из них.

Рассмотрим логически эквивалентный пример.

```
vector& ff()
{
    vector x(7);  // 7 элементов
    // ...
    return x;
}   // Здесь вектор x уничтожается

// ...

vector& p = ff();
// ...
p[4] = 15; // Ой!
```

Такую разновидность проблемы возврата распознают только очень редкие компиляторы.

Обычно программисты недооценивают эти проблемы. Однако многие опытные программисты терпели неудачи, сталкиваясь с бесчисленными

вариациями и комбинациями проблем, порожденных использованием простых массивов и указателей. Решение очевидно — не засоряйте свою программу указателями, массивами, операторами **new** и **delete**. Если вы все же так поступаете, то "быть осторожным" в реальной жизни просто недостаточно. Пользуйтесь векторами, идиомой RAII ("Resource Acquisition Is Initialization" — "Захват ресурса есть инициализация"; см. раздел 19.5), а также другими систематическими подходами к управлению памятью и другими ресурсами.

18.7. Примеры: палиндром

Довольно технических примеров! Попробуем решить маленькую головоломку. *Палиндром* (palindrome) — это слово, которое одинаково читается как слева направо так и справа налево. Например, слова *anna*, *petep* и *malayalam* являются палиндромами, а слова *ida* и *homesick* — нет. Есть два основных способа определить, является ли слово палиндромом.

- Создать копию букв, расположенных в противоположном порядке, и сравнить ее с оригиналом.
- Проверить, совпадает ли первая буква с последней, вторая — с предпоследней и так далее до средины.

Мы выбираем второй подход. Существует много способов выразить эту идею в коде. Они зависят от представления слова и от способа отслеживания сравниваемых букв в слове. Мы напишем небольшую программу, которая будет по-разному проверять, является ли слово палиндромом — просто чтобы увидеть, как разные возможности языка программирования влияют на внешний вид и работу программы.

18.7.1. Палиндромы с использованием `string`

Прежде всего напишем вариант программы, использующий класс **string** из стандартной библиотеки, в котором сравниваемые буквы отслеживаются с помощью целочисленных индексов.

```
bool is_palindrome(const string& s)
{
    int first = 0;            // Индекс первой буквы
    int last = s.length()-1; // Индекс последней буквы
    while (first < last) {    // Мы еще не достигли середины слова
        if (s[first] != s[last]) return false;
        ++first;              // Переход вперед
        --last;               // Переход назад
    }
    return true;
}
```

Мы возвращаем значение **true**, если достигли средины слова, не обнаружив несовпадения букв. Предлагаем вам просмотреть этот код и самим убедиться, что он работает правильно, когда в строке вообще нет букв, когда строка состоит только из одной буквы, когда в строке содержится четное количество букв и когда в строке содержится нечетное количество букв. Разумеется, мы не должны полагаться только на логику, стараясь убедиться, что программа работает правильно, а обязаны выполнить тесты. Можно набросать следующую программу для тестирования функции **is_palindrome()**.

```
int main()
{
    for (string s; cin>>s; ) {
        cout << s << " is";
        if (!is_palindrome(s)) cout << " not";
        cout << " a palindrome\n";
    }
}
```

По существу, причина, по которой мы используем класс **string**, заключается в том, что объекты класса **string** хорошо работают со словами. Они достаточно просто считывают слова, разделенные пробельными символами, и знают свой размер. Если бы мы хотели применить функцию **is_palindrome()** к строкам, содержащим пробельные символы, то могли бы считывать их с помощью функции **getline()** (см. раздел 11.5). Так можно было бы продемонстрировать, что строки *ah ha* и *as df fd sa* являются палиндромами.

18.7.2. Палиндромы с использованием массивов

А если бы у нас не было класса **string** (или **vector**) и нам пришлось бы хранить символы в массиве?

```
bool is_palindrome(const char s[], int n)
    // s указывает на первый символ массива из n символов
{
    int first = 0;          // Индекс первой буквы
    int last = n-1;         // Индекс последней буквы
    while (first < last) { // Мы еще не достигли средины слова
        if (s[first]!=s[last]) return false;
        ++first;            // Переход вперед
        --last;             // Переход назад
    }
    return true;
}
```

Для изучения работы функции **is_palindrome()** сначала необходимо записать символы в массив. Один из безопасных способов (без риска переполнения массива) выглядит так:

```
istream& read_word(istream& is, char* buffer, int max)
    // Считывает не более max-1 символов в массив buffer
{
    is.width(max); // При выполнении следующего оператора >>
                   // будет считано не более max-1 символов
    is >> buffer;  // Читаем слово, завершающееся пробельным
                   // символом, добавляем в конец нулевой символ
    return is;
}
```

Правильная установка ширины потока **istream** предотвращает переполнение массива при выполнении следующей операции **>>**. К сожалению, это также означает, что нам неизвестно, завершается ли чтение из-за пробельного символа или из-за заполнения буфера (т.е. нам надо считывать больше символов). Кроме того, кто помнит детали поведения функции **width()** при вводе? В реальности в качестве входного буфера лучше использовать **string** или **vector**, поскольку они могут увеличиваться, чтобы разместить всю входную информацию полностью. Завершающий нулевой символ необходим, так как большинство операций над массивами символов (С-строками) предполагают наличие такого завершающего символа. Используя функцию **read_word()**, можно написать следующий код:

```
int main()
{
    constexpr int max = 128;
    for (char s[max]; read_word(cin,s,max); ) {
        cout << s << " is";
        if (!is_palindrome(s,strlen(s))) cout << " not";
        cout << " a palindrome\n";
    }
}
```

Вызов **strlen(s)** возвращает количество символов в массиве после выполнения вызова **read_word()**, а инструкция **cout<<s** выводит символы из массива, завершающегося нулевым символом.

Решение задачи с помощью класса **string** намного аккуратнее, чем с помощью массивов, а кроме того, оно гораздо лучше справляется с возможными длинными строками (см. упр. 10).

18.7.3. Палиндромы с использованием указателей

Вместо индексов для идентификации символов можно было бы применить указатели.

```
bool is_palindrome(const char* first, const char* last)
    // Указатель first указывает на первую букву
    // Указатель last указывает на последнюю букву
{
    while (first < last) { // Мы еще не достигли средины
```

```
        if (*first!=*last) return false;
        ++first;            // Переход вперед
        --last;             // Переход назад
    }
    return true;
}
```

Заметим, что к указателям можно применять операции инкремента и декремента. Инкремент приводит к тому, что указатель указывает на следующий элемент массива, а декремент — на предыдущий. Если в массиве нет следующего или предыдущего элемента, возникнет серьезная ошибка, связанная с выходом за пределы допустимого диапазона. Это еще одна проблема, связанная с указателями.

Функцию `is_palindrome()` можно вызывать, например, следующим образом:

```
int main()
{
    const int max = 128;
    for (char s[max]; read_word(cin,s,max); ) {
        cout << s << " is";
        if (!is_palindrome(&s[0],&s[strlen(s)-1]))
            cout << " not";
        cout << " a palindrome\n";
    }
}
```

Просто ради интереса сделаем функцию `is_palindrome()` рекурсивной:

```
bool is_palindrome(const char* first, const char* last)
    // Указатель first указывает на первую букву
    // Указатель last указывает на последнюю букву
{
    if (first<last) {
        if (*first!=*last) return false;
        return is_palindrome(first+1,last-1);
    }
    return true;
}
```

Этот код становится очевидным, если перефразировать определение *палиндрома*: слово является палиндромом, если его первый и последний символы совпадают и если подстрока, получающаяся после отбрасывания первого и последнего символов, также является палиндромом.

Задание

В этой главе мы ставим два задания: одно необходимо выполнить с помощью массивов, а второе — с помощью векторов. Выполните оба задания и сравните количество усилий, которые вы при этом затратили.

Задание с массивами

1. Определите глобальный массив `ga` типа `int`, состоящий из десяти целых чисел и инициализированный числами 1, 2, 4, 8, 16 и т.д.
2. Определите функцию `f()`, принимающую в качестве аргументов массив типа `int` и значение типа `int`, задающее количество элементов в массиве.
3. В функции `f()` выполните следующее.
   - Определите локальный массив `la` типа `int`, состоящий из десяти элементов.
   - Скопируйте значения из массива `ga` в массив `la`.
   - Выведите на печать элементы массива `la`.
   - Определите указатель `p` на тип `int` и инициализируйте его адресом массива, расположенного в динамической памяти и хранящего такое же количество элементов, как и массив, являющийся аргументом функции.
   - Скопируйте значения из массива, являющегося аргументом функции, в массив в динамической памяти.
   - Выведите на печать элементы массива, расположенного в динамической памяти.
   - Освободите память, выделенную для массива.
4. В функции `main()` сделайте следующее.
   - Вызовите функцию `f()` с аргументом `ga`.
   - Определите массив `aa`, содержащий десять элементов, и инициализируйте его первыми десятью значениями факториала (т.е. 1, 2*1, 3*2*1, 4*3*2*1 и т.д.).
   - Вызовите функцию `f()` с аргументом `aa`.

Задание с вектором из стандартной библиотеки

1. Определите глобальный вектор `vector<int> gv`; инициализируйте его десятью целыми числами 1, 2, 4, 8, 16 и т.д.
2. Определите функцию `f()`, принимающую аргумент типа `vector<int>`.
3. В функции `f()` сделайте следующее.

- Определите локальный вектор `vector<int> lv` с тем же количеством элементов, что и вектор, являющийся аргументом функции.
- Скопируйте значения из вектора `gv` в вектор `lv`.
- Выведите на печать элементы вектора `lv`.
- Определите локальный вектор `vector<int> lv2`; инициализируйте его так, чтобы он стал копией вектора, являющегося аргументом функции.
- Выведите на печать элементы вектора `lv2`.

4. В функции `main()` сделайте следующее.
   - Вызовите функцию `f()` с аргументом `gv`.
   - Определите вектор `vector<int> vv` и инициализируйте его первыми десятью значениями факториала (1, 2*1, 3*2*1, 4*3*2*1 и т.д.).
   - Вызовите функцию `f()` с аргументом `vv`.

Контрольные вопросы

1. Что означает выражение "Да будет покупатель бдителен!"?
2. Что по умолчанию представляет собой копирование объектов класса?
3. Когда копирование объектов класса по умолчанию является приемлемым, а когда нет?
4. Что такое копирующий конструктор?
5. Что такое копирующее присваивание?
6. В чем разница между копирующим присваиванием и копирующей инициализацией?
7. Что такое поверхностное копирование? Что такое глубокое копирование?
8. Как сравнить копию объекта класса `vector` с его прототипом?
9. Перечислите пять основных операций над классом.
10. Что собой представляет конструктор с ключевым словом `explicit`? Когда следует использовать этот модификатор?
11. Какие операции могут применяться к объекту класса неявно?
12. Что такое массив?
13. Как скопировать массив?
14. Как инициализировать массив?
15. Когда передача указателя на аргумент предпочтительнее передачи его по ссылке и почему?
16. Что такое строка в стиле C, или C-строка?
17. Что такое палиндром?

Термины

`explicit` конструктор	копирующее присваивание	основные операции
глубокое копирование	копирующий конструктор	палиндром
инициализация массива	массив	перемещающее присваивание
конструктор по умолчанию	поверхностное копирование	перемещающий конструктор

Упражнения

1. Напишите функцию `char*strdup(const char*)`, копирующую С-строку в динамическую память, которую она выделяет для этой копии строки. Не используйте никаких стандартных функций. Не используйте индексирование, вместо него применяйте оператор разыменования `*`.
2. Напишите функцию `char*findx(const char*s, const char*x)`, находящую первое вхождение С-строки `x` в С-строку `s`. Не используйте никаких стандартных функций. Не используйте индексирование, вместо него применяйте оператор разыменования `*`.
3. Напишите функцию `int strcmp(const char*s1, const char*s2)`, сравнивающую две С-строки. Если в лексикографическом порядке строка `s1` меньше строки `s2`, функция должна возвращать отрицательное число, если строки совпадают — нуль, а если строка `s1` больше строки `s2` — положительное число. Не используйте никаких стандартных функций. Не используйте индексирование, вместо него применяйте оператор разыменования `*`.
4. Что случится, если передать функциям `strdup()`, `findx()` и `strcmp()` аргумент, не являющийся С-строкой? Попробуйте! Сначала необходимо выяснить, как получить указатель `char*`, который не указывает на массив символов, завершающийся нулем, а затем применить его (никогда не делайте этого в реальном — не экспериментальном — коде; это может привести к катастрофе). Поэкспериментируйте с "неправильными С-строками" в динамической памяти или стеке. Если результаты будут выглядеть разумно, отключите режим отладки. Переделайте и заново выполните все три функции так, чтобы они получали еще один аргумент — максимально допустимое количество символов в передаваемой строке. Затем протестируйте функции с правильными и неправильными С-строками.
5. Напишите функцию `string cat_dot(const string&s1, const string&s2)`, выполняющую конкатенацию двух строк с точкой между ними. Например, вызов `cat_dot("Нильс", "Бор")` должен вернуть строку `Нильс.Бор`.
6. Модифицируйте функцию `cat_dot()` из предыдущего упражнения так, чтобы в качестве третьего аргумента она получала строку, используемую как разделитель (вместо точки).

7. Напишите варианты функции `cat_dot()` из предыдущих упражнений, получающие в качестве аргументов С-строки и возвращающие С-строку, размещенную в динамической памяти. Не используйте никаких функций или типов из стандартной библиотеки. Протестируйте эти функции на нескольких строках. Убедитесь, что вся память, выделенная с помощью оператора `new`, освобождается с помощью оператора `delete`. Сравните усилия, затраченные вами на выполнение упр. 5 и 6.
8. Перепишите все функции из раздела 18.7, используя для сравнения обращенную копию строки; например, введите строку `"home"`, из нее сгенерируйте строку `"emoh"` и сравните эти две строки, чтобы убедиться, что слово *home* — не палиндром.
9. Проанализируйте схему распределения памяти, описанную в разделе 17.4. Напишите программу, сообщающую, в каком порядке располагаются статическая память, стек и динамическая память. В каком направлении растет стек: в сторону старших или младших адресов? Допустим, массив расположен в динамической памяти. Какой элемент будет иметь больший адрес — с большим индексом или с меньшим?
10. Проанализируйте решение задачи о палиндроме из раздела 18.7.2 с использованием массива. Исправьте его так, чтобы можно было работать с длинными строками: 1) выдавайте сообщение, если введенная строка оказалась слишком длинной; 2) разрешите использовать произвольно длинные строки. Прокомментируйте сложность обеих версий.
11. Разберитесь, что собой представляет *список с пропусками* (skip list), и реализуйте эту разновидность списка. Это непростое упражнение!
12. Реализуйте версию игры "Hunt the Wumpus" ("Охота на Вампуса" или просто "Вамп"). Это простая (не графическая) компьютерная игра, изобретенная Грегори Джобом (Gregory Yob). Цель этой игры — найти довольно смышленого монстра, прячущегося в темном пещерном лабиринте. Ваша задача — убить вампуса с помощью лука и стрел. Кроме вампуса, пещера таит еще две опасности: бездонные ямы и гигантские летучие мыши. Если вы входите в комнату с бездонной ямой, то игра для вас закончена. Если вы входите в комнату с летучей мышью, то она вас хватает и перебрасывает в другую комнату. Если же вы входите в комнату с вампусом или он входит в комнату, где находитесь вы, он вас съедает. Входя в комнату, вы должны получить предупреждение о грозящей опасности.

    "Я чувствую запах вампуса" значит, что он в соседней комнате.

    "Я чувствую ветерок" значит, что в соседней комнате яма.

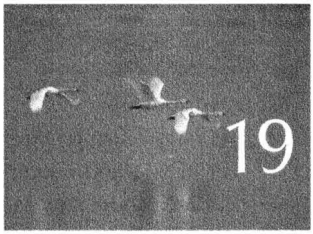
    "Я слышу летучую мышь" значит, что в соседней комнате живет летучая мышь.

Для вашего удобства комнаты пронумерованы. Каждая комната соединена туннелями с тремя другими. Когда вы входите в комнату, то получаете сообщение, например "Вы в комнате номер 12; отсюда идут туннели в комнаты 1, 13 и 4; идти или стрелять?" Возможны ответы наподобие `m13` ("Переход в комнату номер 13") или `s13-4-3` ("Стрелять через комнаты с номерами 13, 4 и 3"). Стрела может пролететь через три комнаты. В начале игры у вас есть пять стрел. Загвоздка со стрельбой заключается в том, что вы можете разбудить вампуса и он войдет в комнату, соседнюю с той, в которой он спал, а она может оказаться вашей комнатой.

Вероятно, самой сложной частью этого упражнения является программирование пещеры и выбор комнат, связанных с другими комнатами. Возможно, вы захотите использовать датчик случайных чисел (например, функцию `randint()` из `std_lib_facilities.h`), чтобы при разных запусках программы использовались разные пещеры и разное расположение летучих мышей и вампуса. *Указание*: обеспечьте способ отладочного вывода состояния лабиринта.

Послесловие

Стандартный класс `vector` основан на средствах низкоуровневого управления памятью, таких как указатели и массивы. Его главное предназначение — помочь программисту избежать сложностей, сопряженных с этими средствами управления памятью. Разрабатывая любой класс, вы должны предусмотреть инициализацию, копирование и уничтожение его объектов.



19

Векторы, шаблоны и исключения

Успех никогда не бывает окончательным.
— Уинстон Черчилль (Winston Churchill)

В этой главе мы завершим проектирование и реализацию наиболее известного и полезного контейнера из библиотеки STL: класса **vector**. Мы покажем, как реализовать контейнеры с переменным количеством элементов и как описать контейнеры, в которых тип является параметром, а также продемонстрируем, как обрабатывать ошибки, связанные с выходом за пределы допустимого диапазона. Как обычно, описанные здесь приемы будут носить универсальный характер, выходя далеко за рамки класса **vector** и даже реализации контейнеров. По существу, мы покажем, как безопасно работать с переменным объемом данных разных типов. Кроме того, в примерах проектирования постараемся учесть конкретные реалии. Наши методы основаны на шаблонах и исключениях, поэтому мы покажем, как определить шаблоны, и продемонстрируем основные способы управления ресурсами, играющими ключевую роль в эффективной работе с исключениями.

19.1. Проблемы

В конце главы 18 наша разработка класса **vector** достигла этапа, на котором мы могли выполнять следующие операции.

- Создавать объекты класса **vector**, элементами которого являются числа с плавающей точкой двойной точности с любым количеством элементов.
- Копировать объекты класса **vector** с помощью присваивания и инициализации.
- Корректно освобождать память, занятую объектом класса **vector**, когда он выходит за пределы области видимости.
- Обращаться к элементам объекта класса **vector**, используя обычные индексные обозначения (как в правой, так и в левой частях оператора присваивания).

Все это хорошо и полезно, но для того, чтобы выйти на ожидаемый уровень сложности (ориентируясь на сложность стандартного библиотечного класса **vector**), мы должны разрешить еще несколько проблем.

- Как изменить размер объекта класса **vector** (изменить количество его элементов)?
- Как перехватить и обработать ошибку, связанную с выходом за границы диапазона при обращении к элементам вектора?

- Как задать тип элементов в объекте класса **vector** в качестве аргумента?

Например, как определить класс **vector** так, чтобы стало возможным написать следующий код:

```
vector<double> vd;     // Элементы типа double
for(double d; cin>>d; )
    vd.push_back(d);   // Увеличение для хранения всех элементов

vector<char> vc(100); // Элементы типа char
int n;
cin >> n;
vc.resize(n);         // Сделать vc имеющим n элементов
```

Очевидно, что такие операции над векторами очень полезны, но почему это так важно с программистской точки зрения? Почему это достойно включения в стандартный набор приемов программирования для использования в будущем? Мы используем две разновидности гибкости. У нас есть одна сущность — класс **vector**, который мы можем изменять двумя способами:

- изменять количество элементов;
- изменять тип элементов.

Эти виды изменчивости весьма полезны и носят фундаментальный характер. Мы постоянно собираем данные. Окидывая взглядом свой письменный стол, я вижу груду банковских счетов, счета за пользование кредитными карточками и за телефонные разговоры. Каждый из этих счетов по существу представляет собой список строк, содержащих информацию разного типа: строки букв и чисел. Передо мной лежит телефон; в нем хранится список имен и телефонных номеров. В книжных шкафах на полках стоят книги. Наши программы такие же, как мой стол: в них имеются контейнеры, состоящие из элементов разных типов. Существуют разные контейнеры (класс **vector** просто используется чаще других), содержащие разную информацию: телефонные номера, имена, суммы банковских операций и документы. По существу, все, что лежит на моем столе, было создано с помощью каких-то компьютерных программ. Очевидным исключением является телефон: он *сам* является компьютером, и когда я просматриваю номера телефонов, то вижу результат работы программы, которая похожа на те, которые мы с вами пишем. Кстати, эти номера можно очень удобно хранить в объекте класса **vector<Number>**.

Очевидно, что не все контейнеры содержат одинаковое количество элементов. Могли бы мы работать с векторами, размер которых фиксируется в момент их инициализации, т.е. могли бы мы написать наш код, не используя функции **push_back()**, **resize()** или другие эквивалентные операции? Конечно, могли бы, но это возложило бы на программиста

совершенно ненужную нагрузку: основной трудностью при работе с контейнерами фиксированного размера является перенос элементов в более крупный контейнер, когда их количество становится слишком большим и превышает первоначальный размер. Например, мы могли выполнить чтение информации в вектор, даже не имея средства для изменения его размера, с помощью следующего кода.

```
// Заполняем вектор, не используя функцию push_back:
vector<double>* p = new vector<double>(10);
int n = 0; // Количество элементов
for(double d; cin >> d; ) {
    if (n==p->size()) {
        vector<double>* q = new vector<double>(p->size()*2);
        copy(p->begin(), p->end(), q->begin());
        delete p;
        p = q;
    }
    (*p)[n] = d;
    ++n;
}
```

Это некрасиво. К тому же вы уверены, что этот код правильно работает? Как можно в этом убедиться? Обратите внимание, что мы внезапно стали использовать указатели и явное управление памятью. Мы были вынуждены это сделать, чтобы имитировать стиль программирования, близкий к машинному уровню при работе с объектами фиксированного размера (массивами; см. раздел 18.6). Одна из причин, обусловивших использование контейнеров, таких как класс **vector**, заключается в желании сделать нечто лучшее; иначе говоря, мы хотим, чтобы класс **vector** сам изменял размер контейнера, освободив пользователей от этой работы и уменьшив вероятность сделать ошибку. Иначе говоря, мы предпочитаем контейнеры, которые могут увеличивать свой размер, чтобы хранить именно столько элементов, сколько нам нужно:

```
vector<double> vd;
for(double d; cin>>d; ) vd.push_back(d);
```

Насколько распространенным является такое изменение размера контейнера? Если эта ситуация встречается редко, то предусматривать для нее специальные средства не имело бы особого смыла; это было бы просто мелким и редко используемым удобством. Однако на самом деле изменение размера встречается очень часто. Наиболее очевидный пример — считывание неизвестного количества значений из потока ввода. Другими примерами являются сбор результатов поиска (нам ведь неизвестно заранее, сколько их будет) и удаление элементов из коллекции один за другим. Таким образом, вопрос заключается не в том, стоит ли реализовывать изменение размера контейнера, а в том, как это сделать.

Почему мы вообще затронули тему, посвященную изменению размера контейнера? Почему бы просто не выделить достаточно памяти и работать с ней? Такая стратегия выглядит наиболее простой и эффективной. Тем не менее она оправдана лишь в том случае, когда мы не запрашиваем слишком много памяти. Программисты, избравшие эту стратегию, вынуждены переписывать свои программы (если они внимательно и систематически отслеживают переполнение памяти) или сталкиваться с аварийными последствиями (если они пренебрегли проверкой переполнения памяти).

Очевидно, что не все векторы должны хранить элементы одного и того же типа. Нам нужны векторы, хранящие числа с двойной точностью, значения температуры, записи (разного вида), строки, операции, кнопки графического пользовательского интерфейса, фигуры, даты, указатели на окна и т.д. Перечисление можно продолжать бесконечно.

Контейнеры также бывают разных видов. Это важное обстоятельство, имеющее значительные последствия, которое обязывает нас хорошенько подумать, прежде чем выбрать конкретный вид контейнера. Почему не все контейнеры представляют собой векторы? Если бы мы имели дело только с одним видом контейнера, то операции над ним можно было бы сделать частью языка программирования. Кроме того, нам не пришлось бы возиться с другими видами контейнеров; мы бы просто всегда использовали `vector`.

Структуры данных играют ключевую роль в большинстве важных приложений. О том, как организовать данные, написано множество толстых и полезных книг. В большинстве из них рассматривается вопрос "Как лучше хранить данные?" Ответ один — нам нужны многочисленные и разнообразные контейнеры, однако это слишком обширная тема, которую в этой книге мы не можем осветить в должной мере. Тем не менее мы уже широко использовали классы `vector` и `string` (`string` представляет собой контейнер для хранения символов). В следующих главах мы опишем классы `list`, `map` (`map` представляет собой дерево пар значений) и матрицы. Поскольку нам нужно множество разнообразных контейнеров, для их поддержки необходимы соответствующие средства языка и методы программирования. Методы, которые мы используем для хранения и доступа к данным, являются одними из наиболее фундаментальных и наиболее полезных методов для реализации нетривиальных видов вычислений.

На уровне машинной памяти все объекты имеют фиксированный размер и не имеют типов. Здесь мы рассматриваем средства языка и методы программирования, позволяющие создавать контейнеры объектов разных типов с переменным количеством элементов. Это обеспечивает значительную степень гибкости и удобства программирования.

19.2. Изменение размера

Какие возможности для изменения размера предлагает класс **vector** из стандартной библиотеки? В нем предусмотрены три простые операции. Допустим, в программе объявлен следующий объект класса **vector**:

```
vector<double> v(n); // v.size()==n
```

Изменить его размер можно тремя способами.

```
v.resize(10);    // Теперь v имеет 10 элементов
v.push_back(7); // Добавляем элемент со значением 7 в конец v;
                // размер v.size() увеличивается на единицу
v = v2;          // Присваиваем другой вектор; теперь v является
                // копией v2 и v.size() == v2.size()
```

Класс **vector** из стандартной библиотеки содержит и другие операции, которые могут изменять размер вектора, например **erase()** и **insert()** (раздел Б.4.7), но здесь мы рассмотрим только, как реализовать три упомянутые выше операции.

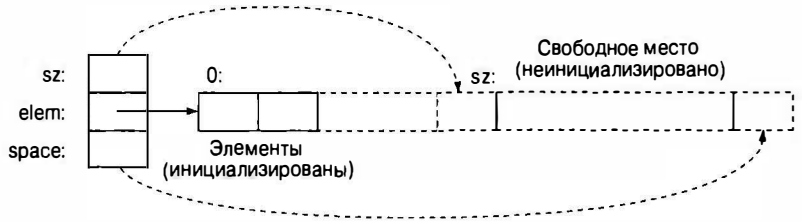
19.2.1. Представление

В разделе 19.1 мы продемонстрировали простейшую стратегию изменения размера: выделить память для нового количества элементов и скопировать в нее старые элементы. Но при частом изменении размера контейнера такая стратегия становится неэффективной. На практике, если мы хотя бы раз меняем размер, то, скорее всего, будем делать это много раз. В частности, в программах редко встречается одиночный вызов функции **push_back()**.

Итак, мы можем оптимизировать наши программы, предусмотрев изменение размера контейнера. На самом деле все реализации класса **vector** отслеживают как количество элементов, так и объем "свободного места", зарезервированного для "будущего расширения". Рассмотрим пример.

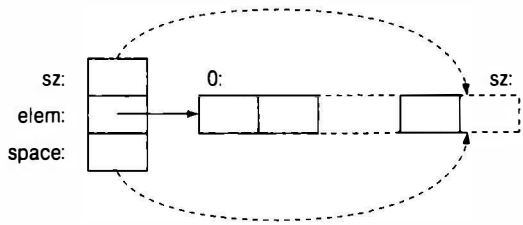
```
class vector {
    int sz;        // Количество элементов
    double* elem; // Адрес первого элемента
    int space;     // Количество элементов плюс "свободное
                   // место"/"слоты" для новых элементов
                   // (текущий выделенный размер)
public:
    // ...
};
```

Графически это можно изобразить следующим образом.



Поскольку нумерация элементов начинается с нуля, мы указываем, что переменная **sz** (количество элементов) ссылается на ячейку, находящуюся за последним элементом, а переменная **space** ссылается на ячейку, расположенную за последним слотом. Им соответствуют указатели, указывающие на ячейки **elem+sz** и **elem+space**.

Когда вектор создается впервые, значение **space** равно значению **sz**, т.е. "свободного места" нет.

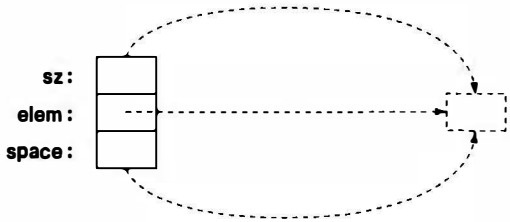


Мы не начинаем выделение дополнительных слотов, пока количество элементов не изменится. Обычно это происходит, когда выполняется условие **space==sz**. Благодаря этому, пока мы не используем функцию **push_back()**, излишних расходов памяти нет.

Конструктор по умолчанию (создающий объект класса **vector** без элементов) устанавливает целочисленные члены класса равными нулю, а указатель — равным **nullptr**:

```
vector::vector():sz{0}, elem{nullptr}, space{0} {}
```

Графически эта ситуация выглядит следующим образом.



Этот выходящий за пределы элемент является умозрительным. Конструктор по умолчанию не выделяет динамическую память и занимает минимальный объем (см. упр. 16).

Наш класс **vector** иллюстрирует методы, которые можно использовать для реализации стандартного вектора (и других структур данных), но при реализации стандартной библиотеки разработчикам предоставлена большая свобода, так что вполне возможно, что в вашей системе класс **std::vector** сделан иначе.

19.2.2. Функции reserve и capacity

Наиболее фундаментальной операцией при изменении размера контейнера (т.е. при изменении количества элементов) является функция **vector::reserve()**. Она добавляет память для новых элементов.

```
void vector::reserve(int newalloc)
{
    if (newalloc<=space) return; // Размер никогда не уменьшается
    double* p =
        new double[newalloc];    // Выделение новой памяти
    for(int i=0; i<sz; ++i)
        p[i] = elem[i];          // Копируем старые элементы
    delete[] elem;               // Освобождение старой памяти
    elem = p;
    space = newalloc;
}
```

Обратите внимание на то, что мы не инициализировали элементы в зарезервированной памяти. Мы просто резервируем память, а как ее использовать — задача функций **push_back()** и **resize()**.

Очевидно, что пользователя может интересовать размер доступной памяти в объекте класса **vector**, поэтому, аналогично стандартному классу, мы предусмотрели функцию-член, возвращающую эту информацию.

```
int vector::capacity() const { return space; }
```

Иначе говоря, для объекта класса **vector** с именем **v** выражение **v.capacity()-v.size()** возвращает количество элементов, которое можно записать в объект **v** с помощью функции **push_back()** без выделения дополнительной памяти.

19.2.3. Функция resize

Имея функцию **reserve()**, реализовать функцию **resize()** для класса **vector** не представляет труда. Необходимо обрабатывать несколько ситуаций.

- Новый размер больше ранее выделенной памяти.
- Новый размер больше прежнего, но меньше или равен ранее выделенной памяти.

- Новый размер равен старому.
- Новый размер меньше прежнего.

Посмотрим, что у нас получилось.

```
void vector::resize(int newsize)
    // Создаем вектор, содержащий newsize элементов
    // Инициализируем каждый элемент значением по умолчанию 0.0
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i)
        elem[i] = 0;       // Инициализируем новые элементы
     sz = newsize;
}
```

Основная работа с памятью поручена функции **reserve()**. Цикл инициализирует новые элементы (если они есть).

Мы не обрабатываем каждую из упомянутых выше ситуаций явно, но, как легко проверить, все они, тем не менее, обрабатываются правильно.

ПОПРОБУЙТЕ

Какие варианты следует предусмотреть (и протестировать), если мы хотим убедиться, что данная функция **resize()** работает правильно? Что вы скажете о случаях **newsize == 0** и **newsize == -77**?

19.2.4. Функция push_back

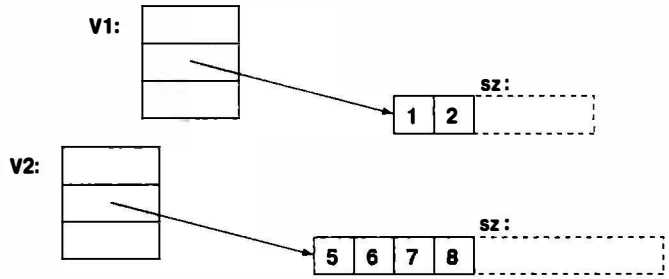
При первом рассмотрении функция **push_back()** может показаться сложной для реализации, но функция **reserve()** все упрощает.

```
void vector::push_back(double d)
    // Увеличивает размер вектора на единицу;
    // инициализирует новый элемент значением d
{
    if (space==0)
        reserve(8);        // Выделяем память для 8 элементов
    else if (sz==space)
        reserve(2*space); // Выделяем дополнительную память
    elem[sz] = d;          // Добавляем d в конец вектора
    ++sz;                  // увеличиваем количество элементов
}
```

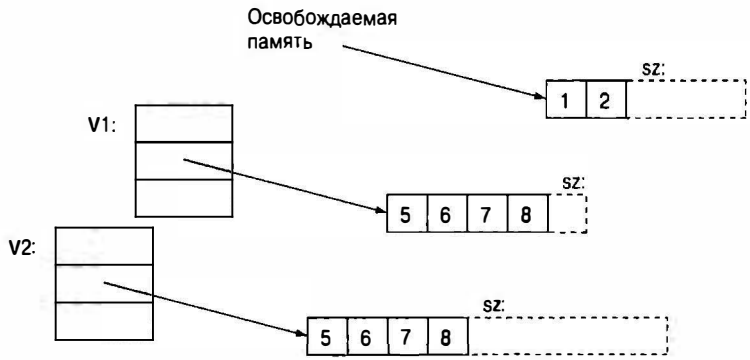
Другими словами, если у нас нет свободной памяти, то мы удваиваем размер выделенной памяти. На практике эта стратегия оказывается очень удачной, поэтому она используется в большинстве реализаций класса **vector** из стандартной библиотеки.

19.2.5. Присваивание

Присваивание векторов можно определить несколькими способами. Например, мы могли бы разрешать присваивание, только если векторы имеют одинаковое количество элементов. Однако в разделе 18.3.2 мы решили, что присваивание векторов должно иметь более общий характер и более очевидный смысл: после присваивания **v1=v2** вектор **v1** является копией вектора **v2**. Рассмотрим следующую диаграмму.



Очевидно, что мы должны скопировать элементы, но нужно ли при этом копировать пустые места за последним элементом? Нет! Новый объект класса **vector** получает копии элементов, но поскольку мы не знаем, как он будет использоваться, то не выделяем свободного места в конце вектора.



Простейшая реализация работает следующим образом:

- выделяем память для копии;
- копируем элементы;
- освобождаем старую память;
- присваиваем членам **sz**, **elem** и **space** новые значения.

Вот как это можно сделать в коде:

```
vector& vector::operator=(const vector& a)
    // Похож на конструктор копирования,
    // но мы должны разобраться со старыми элементами
```

```
{
    double* p =
        new double[a.sz];    // Выделяем новую память
    for (int i = 0; i<a.sz; ++i)
        p[i] = a.elem[i];    // Копируем элементы
    delete[] elem;           // Освобождаем старую память
    space = sz = a.sz;       // Устанавливаем новый размер
    elem = p;                // Устанавливаем указатель на новые элементы
    return *this;            // Возвращаем ссылку на себя
}
```

Согласно общепринятому соглашению оператор присваивания возвращает ссылку на целевой объект. Смысл выражения ***this** объяснялся в разделе 17.10.

Эта реализация вполне корректна, но, немного поразмыслив, легко увидеть, что мы выполняем избыточные операции выделения и освобождения памяти. Что если целевой вектор содержит больше элементов, чем присваиваемый вектор? Что если целевой вектор содержит столько же элементов, сколько и присваиваемый вектор? Во многих приложениях последняя ситуация встречается очень часто. В указанных случаях мы можем просто скопировать элементы в память, уже выделенную ранее целевому вектору.

```
vector& vector::operator=(const vector& a)
{
    if (this==&a) return *this;    // Самоприсваивание, ничего делать не надо

    if (a.sz<=space) {             // Памяти достаточно, новая
                                   // память не нужна
        for (int i = 0; i<a.sz; ++i)
            elem[i] = a.elem[i];   // Копируем элементы
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz]; // Выделяем новую память
    for (int i = 0; i<a.sz; ++i)
        p[i] = a.elem[i];          // Копируем элементы
    delete[] elem;                 // Освобождаем старую память
    space = sz = a.sz;             // Устанавливаем новый размер
    elem = p;                      // Устанавливаем указатель
                                   // на новые элементы
    return *this;                  // Возвращаем ссылку на себя
}
```

В этом фрагменте кода мы сначала проверяем самоприсваивание (например, **v=v**); в этом случае ничего делать не надо. С логической точки зрения эта проверка лишняя, но иногда это оказывается значительной оптимизацией. Данная проверка демонстрирует использование указателя **this**, позволяющего проверить, является ли аргумент **a** тем же объектом, что

и объект, из которого вызывается функция-член (т.е. **operator=()**). Убедитесь, что код будет корректно работать, если эту проверку из него удалить. Проверка **a.sz<=space** также включена для оптимизации. Убедитесь также, что код действительно корректно работает и после удаления этой проверки.

19.2.6. Текущее состояние дел

Итак, мы получили почти реальный класс **vector** для чисел типа **double**.

```
// Почти реальный вектор чисел типа double
class vector {
/*
    Инвариант:
        для 0<=n<sz значение elem[n] является n-м элементом
        sz<=space;
        если sz<space, то после elem[sz-1] есть место
        для (space-sz) чисел типа double
*/
    int sz;        // Размер
    double* elem; // Указатель на элементы (или 0)
    int space;     // Количество элементов плюс количество
                   // свободных слотов
public:
    vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit vector(int s):sz{s},elem{new double[s]},space{s}
    {
        for (int i=0; i<sz; ++i)      // Элементы
            elem[i]=0;                // инициализированы
    }

    // Копирующие конструктор и присваивание:
    vector(const vector&);
    vector& operator=(const vector&);

    // Перемещающие конструктор и присваивание:
    vector(const vector&&);
    vector& operator=(const vector&&);

    ~vector() { delete[] elem; }      // Деструктор

    double& operator[ ](int n){return elem[n];} // Доступ
    const double& operator[](int n) const {return elem[n];}

    int size() const { return sz; }
    int capacity() const { return space; }

    void resize(int newsize);         // Увеличение
    void push_back(double d);
    void reserve(int newalloc);
};
```

Обратите внимание на то, что этот класс содержит все основные операции (раздел 18.4): конструктор, конструктор по умолчанию, копирующие операции, деструктор. Он также имеет операцию для доступа к данным (индексирование `[]`), получения информации об этих данных (`size()` и `capacity()`), а также для управления ростом вектора (`resize()`, `push_back()` и `reserve()`).

19.3. Шаблоны

Однако нам мало иметь вектор, состоящий из чисел типа `double`; мы хотим свободно задавать тип элементов наших векторов, например:

```
vector<double>
vector<int>
vector<Month>
vector<Window*>           // Вектор указателей на Window
vector< vector<Record>>   // Вектор векторов Record
vector<char>
```

Для этого мы должны научиться определять шаблоны. На самом деле мы использовали шаблоны с самого начала, но до сих пор нам не приходилось определять их самостоятельно. Стандартная библиотека содержит все, что нам требовалось до сих пор, но мы не должны полагаться на готовые рецепты. Поэтому следует разобраться, как спроектирована и реализована стандартная библиотека, предоставляющая, например, класс `vector` или функцию `sort()` (разделы 21.1 и Б.5.4). Это не просто теоретический интерес, поскольку, как обычно, средства и методы, использованные при создании стандартной библиотеки, могут помочь при работе над собственными программами. Например, в главах 21 и 22 мы покажем, как с помощью шаблонов реализовать стандартные контейнеры и алгоритмы, а в главе 24 продемонстрируем, как разработать класс матриц для научных вычислений.

По существу, *шаблон* (template) — это механизм, позволяющий программисту использовать типы в качестве параметров класса или функции. Получив эти аргументы, компилятор генерирует конкретный класс или функцию.

19.3.1. Типы как шаблонные параметры

Мы хотим, чтобы тип элементов был параметром класса `vector`. Возьмем класс `vector` и заменим ключевое слово `double` буквой `T`, где `T` — параметр, который может принимать "значения", такие как `double`, `int`, `string`, `vector<Record>` или `Window*`. В языке C++ обозначением для введения параметра `T`, задающего тип, используется префикс `template <typename T>`, означающий "для всех типов `T`", например:

```
// Почти реальный вектор элементов типа T
template<typename T> class vector {
    // Читается как "для всех типов T"
    // (почти так же, как в математике)
    int sz;      // Размер
    T* elem;     // Указатель на элементы
    int space;   // Размер + свободная память
public:
    vector(): sz{0}, elem{nullptr}, space{0} {}

    explicit vector(int s):sz{s},elem{new T[s]},space{s}
    {
        for (int i=0; i<sz; ++i)      // Элементы
            elem[i]=0;                // инициализированы
    }

    // Копирующие конструктор и присваивание:
    vector(const vector&);
    vector& operator=(const vector&);

    // Перемещающие конструктор и присваивание:
    vector(const vector&&);
    vector& operator=(const vector&&);

    ~vector() { delete[] elem; }              // Деструктор

     T& operator[ ](int n){return elem[n];} // Доступ
     const T& operator[](int n) const {return elem[n];}

     int size() const { return sz; }
     int capacity() const { return space; }

     void resize(int newsize);                // Увеличение
     void push_back(const T& d);
     void reserve(int newalloc);
};
```

Это определение класса **vector** совпадает с определением класса **vector**, содержащего элементы типа **double** (см. раздел 19.2.6), за исключением того, что ключевое слово **double** теперь заменено шаблонным параметром **T**. Этот шаблонный класс **vector** можно использовать следующим образом:

```
vector<double> vd;       // T – double
vector<int> vi;          // T – int
vector<double*> vpd;     // T – double*
vector<vector<int>> vvi; // T – vector<T>, в котором T – int
```

Можно просто считать, что компилятор генерирует класс для конкретного типа (соответствующего аргументу шаблона), подставляя его вместо параметра шаблона. Например, когда компилятор видит в программе конструкцию **vector<char>**, он генерирует примерно такой код:

```
class vector_char {
    int sz;      // Размер
    char* elem; // Указатель на элементы
    int space;  // Размер + свободная память
public:
    vector_char(): sz{0}, elem{nullptr}, space{0} {}

    explicit vector_char(int s):sz{s},elem{new char[s]},space{s}
    {
        for (int i=0; i<sz; ++i)    // Элементы
            elem[i]=0;              // инициализированы
    }

    // Копирующие конструктор и присваивание:
    vector_char(const vector_char&);
    vector_char& operator=(const vector_char&);

    // Перемещающие конструктор и присваивание:
    vector_char(const vector_char&&);
    vector_char& operator=(const vector_char&&);

    ~vector_char() { delete[] elem; }          // Деструктор

     char& operator[ ](int n){return elem[n];} // Доступ
     const char& operator[](int n) const {return elem[n];}

     int size() const { return sz; }
     int capacity() const { return space; }

     void resize(int newsize);                 // Увеличение
     void push_back(const char& d);
     void reserve(int newalloc);
};
```

Для класса **vector<double>** компилятор генерирует аналог класса **vector**, содержащего элементы типа **double**, из раздела 19.2.6 (используя подходящее внутреннее имя, означающее **vector<double>**).

Иногда шаблон класса называют *генератором типов* (type generator). Процесс генерации типов (классов) из шаблона класса с заданными аргументами шаблона называется *специализацией* (specialization) или *инстанцированием шаблона* (template instantiation). Например, классы **vector <char>** и **vector<Poly_line*>** являются специализациями класса **vector**. В простых ситуациях, например при работе с классом **vector**, инстанцирование не вызывает затруднений. В более общих и запутанных ситуациях инстанцирование шаблона очень сильно усложняется. К счастью для пользователей шаблонов, вся эта сложность обрушивается только на разработчика компилятора. Инстанцирование шаблона (генерация специализации шаблона) осуществляется на этапе компиляции или редактирования связей, а не во время выполнения программы.

Естественно, мы можем использовать функции-члены такого шаблонного класса, например:

```
void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // ...
}
```

При вызове функции-члена шаблонного класса компилятор генерирует соответствующую функцию. Например, когда компилятор видит вызов `v.push_back("Norah")`, он генерирует функцию

```
void vector<string>::push_back(const string& d) { /* ... */ }
```

используя определение шаблона

```
template<typename T>
void vector<T>::push_back(const T& d) { /* ... */ };
```

Таким образом, для вызова `v.push_back("Norah")` имеется соответствующая функция. Другими словами, если вам нужна функция для данного объекта и аргументов типов, компилятор сам напишет ее на основе вашего шаблона.

Вместо префикса `template<typename T>` можно использовать префикс `template<class T>`. Эти две конструкции означают одно и то же, но некоторые программисты все же предпочитают использовать ключевое слово `typename`, "потому, что оно яснее" и "потому, что никто не подумает, что оно запрещает использовать встроенные типы, например тип `int`, в качестве аргумента шаблона". Мы считаем, что ключевое слово `class` уже означает "тип", поэтому никакой разницы между этими конструкциями нет. Кроме того, слово `class` короче.

19.3.2. Обобщенное программирование

Шаблоны — это основа для обобщенного программирования на языке C++. По существу, простейшее определение "обобщенного программирования" на языке C++ — это "программирование с использованием шаблонов". Хотя, конечно, это определение носит слишком упрощенный характер. Не следует давать определения фундаментальных концепций программирования в терминах возможностей языка программирования. Эти возможности существуют для того, чтобы поддерживать методы программирования, а не наоборот. Как и большинство широко известных понятий, имеется много определений "обобщенного программирования". Мы считаем наиболее полезным следующее простое определение.

> *Обобщенное программирование* — это создание кода, работающего с разными типами, заданными в виде аргументов, причем эти типы должны соответствовать специфическим синтаксическим и семантическим требованиям.

Например, элементы вектора должны иметь тип, который можно копировать (с помощью копирующего конструктора и копирующего присваивания). В главах 20 и 21 будут представлены шаблоны, которые требуют наличия арифметических операций у их аргументов. Когда мы выполняем параметризацию класса, мы получаем *шаблон класса* (class template), который часто называют также *параметризованным типом* (parameterized type) или *параметризованным классом* (parameterized class). Когда мы выполняем параметризацию функции, мы получаем *шаблон функции* (function template), которую часто называют *параметризованной функцией* (parameterized function), а иногда *алгоритмом* (algorithm). По этой причине обобщенное программирование иногда называют "алгоритмически ориентированным программированием" (algorithm-oriented programming); в этом случае основное внимание при проектировании переносится на алгоритмы, а не на используемые ими типы данных.

Поскольку понятие параметризованных типов играет очень важную роль в программировании, мы попытаемся в дальнейшем немного разобраться во всей этой запутанной терминологии. Это даст нам возможность избежать недоразумений, когда мы встретим знакомые понятия в других контекстах.

Данную форму обобщенного программирования, основанную на явных шаблонных параметрах, часто называют *параметрическим полиморфизмом* (parametric polymorphism) — в отличие от обычного полиморфизма на основе иерархий классов и виртуальных функций (ad hoc polymorphism, которому соответствует стиль программирования, именуемый *объектно-ориентированным программированием*; см. разделы 14.3–14.4). Причина, по которой оба стиля программирования называют *полиморфизмом* (polymorphism), заключается в том, что каждый из них дает программисту возможность создавать много версий одной и той же концепции с помощью единого интерфейса. *Полиморфизм* по-гречески означает "много форм", что в нашем случае означает много различных типов, работать с которыми можно с помощью общего интерфейса. В примерах, посвященных классу **Shape**, рассмотренных в главах 16–19, мы буквально работали с разными формами (классами **Text**, **Circle** и **Polygon**) с помощью общего интерфейса, определенного классом **Shape**. Используя класс **vector**, мы фактически работаем со многими векторами (например, **vector<int>**, **vector<double>** и **vector<Shape*>**) с помощью общего интерфейса, определенного шаблоном класса **vector**.

Имеется ряд отличий объектно-ориентированного программирования (с помощью иерархий классов и виртуальных функций) от обобщенного программирования (с помощью шаблонов). Наиболее очевидным является то, что выбор вызываемой функции при обобщенном программировании определяется компилятором во время компиляции, а при объектно-ориентированном программировании он определяется во время выполнения программы[1]. Рассмотрим пример.

```
v.push_back(x); // Внести x в вектор v
s.draw();       // Нарисовать фигуру s
```

В вызове `v.push_back(x)` компилятор определит тип элементов в объекте `v` и применит соответствующую функцию `push_back()`, а для вызова `s.draw()` он косвенно (с помощью таблицы виртуальных функций, связанной с объектом `s`; см. раздел 14.3.1) вызовет некую функцию `draw()`. Это дает объектно-ориентированному программированию ту степень свободы, которой лишено обобщенное программирование, но в то же время делает обычное обобщенное программирование более систематическим, понятным и эффективным.

Подведем итоги.

- *Обобщенное программирование* поддерживается шаблонами, основываясь на разрешении вызовов времени компиляции.

- *Объектно-ориентированное программирование* поддерживается иерархиями классов и виртуальными функциями, основываясь на разрешении вызовов времени выполнения.

Сочетание этих стилей программирования вполне возможно и полезно. Рассмотрим пример.

```
void draw_all(vector<Shape*>& v)
{
    for (int i=0; i<v.size(); ++i) v[i]->draw();
}
```

Здесь мы вызываем виртуальную функцию (`draw()`) с помощью механизма вызова виртуальных функций через указатель на базовый класс `Shape` — это, определенно, объектно-ориентированное программирование. Однако указатели `Shape*` хранятся в объекте класса `vector`, который является параметризованным типом, значит, мы одновременно применяем (простое) обобщенное программирование.

Но довольно философии. Для чего же на самом деле используются шаблоны? Для получения непревзойденно гибких и высокопроизводительных программ.

---

[1] Что позволяет говорить о полиморфизмах времени компиляции и времени выполнения, или о статическом и динамическом полиморфизмах. — *Примеч. ред.*

- Используйте шаблоны, когда важную роль играет производительность программы (например, при интенсивных вычислениях или в программах реального времени; подробнее об этом речь пойдет в главах 24 и 25).
- Используйте шаблоны, когда важна гибкость сочетания информации из разных типов (например, в стандартной библиотеке C++; эта тема будет обсуждаться в главах 20 и 21).

19.3.3. Концепции

Шаблоны обладают многими полезными свойствами, такими как высокая гибкость и почти оптимальная производительность, но, к сожалению, они не идеальны. Как всегда, преимуществам сопутствуют недостатки. Основным недостатком шаблонов является то, что гибкость и высокая производительность достигаются за счет плохого отделения "внутренностей" шаблона (его определения) от его интерфейса (объявления). Это проявляется в плохой диагностике ошибок; особенно жуткими являются сообщения об ошибках. Иногда эти сообщения об ошибках в процессе компиляции выдаются намного позже, чем следовало бы.

При компиляции программы, использующей шаблоны, компилятор "заглядывает" внутрь шаблонов и их аргументов. Он делает это для того, чтобы извлечь информацию, необходимую для генерации оптимального кода. Чтобы эта информация стала доступной, современные компиляторы требуют, чтобы шаблон был полностью определен везде, где он используется. Это относится и к его функциям-членам и ко всем шаблонным функциям, вызываемым из них. В результате авторы шаблонов стараются разместить определения шаблонов в заголовочных файлах. На самом деле стандарт этого не требует, но пока не будут разработаны существенно более эффективные реализации языка, мы рекомендуем вам поступать со своими шаблонами именно так: размещайте в заголовочном файле определения всех шаблонов, используемых в нескольких единицах трансляции.

Мы рекомендуем вам начинать с очень простых шаблонов и постепенно набираться опыта. Один из полезных приемов проектирования мы уже продемонстрировали на примере класса `vector`: сначала разработайте и протестируйте класс, используя конкретные типы. Если программа работает, замените конкретные типы шаблонными параметрами и протестируйте программу с использованием разных аргументов. Для обеспечения общности, безопасности типов и высокой производительности программ используйте библиотеки шаблонов, например стандартную библиотеку языка C++. В главах 20 и 21, посвященных контейнерам и алгоритмам стандартной библиотеки, приведено много примеров использования шаблонов.

Стандарт C++14 предоставляет механизм чрезвычайно усовершенствованных проверок интерфейсов шаблонов. Например, в C++11 мы пишем

```
template<typename T> // Для всех типов T
class vector {
    // ...
};
```

Мы не можем в точности утверждать, каким должен быть ожидаемый тип `T`. Стандарт говорит об этих требованиях, но только обычным, естественным "человеческим" языком, а не языком кода, понятного компилятору. Мы называем множество требований к аргументу шаблона *концепцией* (concept). Аргумент шаблона должен соответствовать этим требованиям — концепциям — шаблона, к которому он применяется. Например, вектор требует, чтобы его элементы могли быть скопированы или перемещены, чтобы можно было получать их адреса, и (при необходимости) они могли быть созданы с помощью конструктора по умолчанию. Другими словами, элемент должен соответствовать набору требований, которые мы могли бы назвать `Element`. В C ++ 14 мы можем сделать это явно:

```
template<typename T>       // Для всех типов T
    requires Element<T>() // таких, что T является Element
class vector {
    // ...
};
```

Этот код показывает, что концепция на самом деле представляет собой предикат типа, т.е. функцию, вычисляемую на этапе компиляции (`constexpr`), которая возвращает `true`, если аргумент типа (в данном случае `T`) имеет свойства, требуемые концепцией (в данном случае именуемой `Element`), и `false`, если это не так. Эта запись длинновата, но можно воспользоваться ее сокращением:

```
template<Element T> // Для всех типов T, таких что Element<T>() равно true
class vector {
    // ...
};
```

Если у нас нет компилятора, соответствующего стандарту C++14 и поддерживающего концепции, мы можем указать наши требования в именах и комментариях:

```
template<typename Elem> // Требует Element<Elem>()
class vector {
    // ...
};
```

Компилятор не понимает наши имена и не читает наши комментарии, но явно выраженная концепция помогает нам лучше понимать собственный код, улучшает дизайн обобщенного кода и помогает другим программистам

читать наш код. В дальнейшем мы будем использовать некоторые распространенные и полезные концепции.

- `Element<E>()`: `E` может быть элементом контейнера.
- `Container<C>()`: `C` может хранить элементы `Element` и быть доступным как последовательность [`begin()`:`end()`).
- `Forward_iterator<For>()`: `For` может применяться для обхода последовательности [`b`:`e`) (как в случае связанного списка, вектора или массива).
- `Input_iterator<In>()`: `In` может использоваться для одноразового чтения последовательности [`b`:`e`) (как в случае входного потока).
- `Output_iterator<Out>()`: последовательность может быть выведена с помощью `Out`.
- `Random_access_iterator<Ran>()`: `Ran` может использоваться для многократного чтения и записи последовательности [`b`,`e`) и поддерживает индексацию с помощью `[]`.
- `Allocator<A>()`: `A` может применяться для выделения и освобождения памяти (как в случае динамической памяти).
- `Equal_comparable<T>()`: два объекта типа `T` могут сравниваться с использованием оператора `==` для получения булева результата.
- `Equal_comparable<T,U>()`: объекты типа `T` и `U` могут сравниваться с использованием оператора `==` для получения булева результата.
- `Predicate<P,T>()`: можно вызвать `P` с аргументом типа `T` для получения булева результата.
- `Binary_predicate<P,T>()`: можно вызвать `P` с двумя аргументами типа `T` для получения булева результата.
- `Binary_predicate<P,T,U>()`: можно вызвать `P` с аргументами типов `T` и `U` для получения булева результата.
- `Less_comparable<L,T>()`: можно использовать `L` для сравнения двух объектов типа `T` для выяснения, меньше ли один другого, с помощью оператора `<` для получения булева результата.
- `Less_comparable<L,T,U>()`: можно использовать `L` для сравнения двух объектов типа `T` и `U` для выяснения, меньше ли один другого, с помощью оператора `<` для получения булева результата.
- `Binary_operation<B,T,U>()`: можно использовать `B` для выполнения операции над двумя объектами типа `T`.
- `Binary_operation<B,T,U>()`: можно использовать `B` для выполнения операции над двумя объектами типа `T` и `U`.
- `Number<N>()`: `N` ведет себя как число, поддерживая операции `+`, `-`, `*` и `/`.

Для контейнеров и алгоритмов стандартной библиотеки эти (и многие другие) концепции определены с мучительной детализацией. В данной книге, в особенности в главах 20 и 21, мы будем использовать их неформально для документирования наших контейнеров и алгоритмов.

Тип контейнера и тип итератора, `T`, имеет тип значения (записываемый как `Value_type<T>`), который представляет собой тип элемента. Зачастую `Value_type<T>` является типом-членом `T::value_type`; см. классы `vector` и `list` (раздел 20.5).

19.3.4. Контейнеры и наследование

Это одна из разновидностей сочетания объектно-ориентированного и обобщенного программирования, которое люди постоянно, но безуспешно пытаются применять: использование контейнера объектов производного класса в качестве контейнера объектов базового класса. Рассмотрим пример.

```
vector<Shape> vs;
vector<Circle> vc;
vs = vc;                    // Ошибка: требуется vector<Shape>
void f(vector<Shape>&);
f(vc);                      // Ошибка: требуется vector<Shape>
```

Но почему? "В конце концов, — говорите вы, — я могу преобразовать класс `Circle` в класс `Shape`!" Нет, не можете. Вы можете преобразовать указатель `Circle*` в `Shape*`, ссылку `Circle&` в ссылку `Shape&`, но мы сознательно запретили присваивать объекты класса `Shape`, поэтому вы не должны удивляться тому, что произошло бы, помести вы объект класса `Circle` с определенным радиусом в переменную типа `Shape`, которая не имеет радиуса (см. раздел 14.2.4). Если бы это произошло, т.е. если бы мы разрешили такое присваивание, то произошла бы так называемая "*срезка*" ("slicing"), похожая на усечение целых чисел (см. раздел 3.9.2).

Попытаемся снова — на этот раз с использованием указателей.

```
vector<Shape*> vps;
vector<Circle*> vpc;
vps = vpc;                  // Ошибка: требуется vector<Shape*>
void f(vector<Shape*>&);
f(vpc);                     // Ошибка: требуется vector<Shape*>
```

И вновь система типов сопротивляется. Почему? Рассмотрим, что может сделать некоторая функция `f()`.

```
void f(vector<Shape*>& v)
{
    v.push_back(new Rectangle{Point{0,0},Point{100,100}});
}
```

Очевидно, что мы можем записать указатель **Rectangle*** в вектор **vector<Shape*>**. Однако, если бы этот объект класса **vector<Shape*>** в каком-то месте программы рассматривался как объект класса **vector<Circle*>**, то мог бы возникнуть неприятный сюрприз. В частности, если бы компилятор без замечаний пропустил пример, приведенный выше, то что бы указатель **Rectangle*** делал в векторе **vpc**? Наследование — мощный и тонкий механизм, и шаблоны не расширяют его возможности неявно. Существуют способы использования шаблонов для выражения наследования, но эта тема выходит за рамки рассмотрения этой книги. Просто запомните, что из "**D** — это **B**" не следует, что "**C<D>** — это **C<B>**" для произвольного шаблона **C**. Мы должны ценить это обстоятельство как защиту от непреднамеренного нарушения типов. (См. также раздел 25.4.4.)

19.3.5. Целые числа как параметры шаблонов

Очевидно, что параметризация классов с помощью типов является полезной. А что можно сказать о параметризации классов с помощью "других вещей", например целых чисел или строк? По сути, оказаться полезным может любой вид аргументов, но мы будем рассматривать только типы и целочисленные параметры. Другие виды параметров оказываются полезными куда реже, а поддержка языком C++ других видов параметров носит более сложный характер и требует обширных и глубоких знаний возможностей языка.

Рассмотрим пример наиболее распространенного использования целочисленного значения в качестве аргумента шаблона: контейнер, количество элементов которого известно на этапе компиляции.

```
template<typename T, int N> struct array {
    T elem[N];    // Хранит элементы в массиве – члене класса,
                  // использует конструкторы по умолчанию,
                  // деструктор и присваивание

    T& operator[] (int n);      // Доступ: возвращает ссылку
    const T& operator[] (int n) const;

    T* data() { return elem; } // Преобразование в тип T*
    const T* data() const { return elem; }

    int size() const { return N; }
};
```

Мы можем использовать тип **array** (см. также раздел 20.7) примерно так:

```
array<int,256> gb;          // 256 целых чисел
array<double,6> ad = {0.0, 1.1, 2.2, 3.3, 4.4, 5.5};
const int max = 1024;
```

```
void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops;    // Ошибка: n компилятору неизвестно
    // ...
    array<char,max> loc2 = loc; // Создаем резервную копию
    // ...
    loc = loc2;                 // Восстанавливаем
    // ...
}
```

Ясно, что класс **array** очень простой — гораздо более простой и менее мощный, чем класс **vector**; так почему иногда следует использовать его, а не класс **vector**? Один из ответов — "эффективность". Размер объекта класса **array** известен на этапе компиляции, поэтому компилятор может выделить статическую память (для глобальных объектов, таких как **gb**) или память в стеке (для локальных объектов, таких как **loc**), а не динамическую память. Проверяя выход за пределы диапазона, мы сравниваем константы (например, параметр размера *N*). Для большинства программ такое повышение эффективности незначительно, но если мы создаем критический компонент системы (например, драйвер сети), то даже небольшая разница оказывается существенной. Что еще более важно, некоторые программы просто не могут использовать динамическую память. Такие программы обычно работают во встроенных системах и/или в программах, критичных с точки зрения безопасности (см. главу 25). В таких программах массив **array** имеет много преимуществ по сравнению с классом **vector** без нарушения критичного ограничения (запрета на использование динамической памяти).

Зададим противоположный вопрос — не "Почему бы просто не использовать класс **vector**?", а "Почему бы не использовать встроенные массивы?" Как было показано в разделе 18.6, массивы могут порождать ошибки: они не знают своего размера, они конвертируют указатели при малейшей возможности и неправильно копируются; у **array**, как и у **vector**, таких проблем нет:

```
double* p = ad;         // Ошибка: нет неявного преобразования в указатель
double* q = ad.data(); // ok: явное преобразование

template<typename C> void printout(const C& c) // Шаблон функции
{
    for (int i = 0; i<c.size(); ++i) cout << c[i] <<'\n';
}
```

Эту функцию **printout()** можно вызвать как для **array**, так и для **vector**.

```
printout(ad);       // Вызов для array
vector<int> vi;
// ...
printout(vi);       // Вызов для vector
```

Это простой пример обобщенного программирования, примененный к доступу к данным. Он работает благодаря тому, что в шаблоне **array**, как и в шаблоне **vector** используется один и тот же интерфейс (функция **size()** и операция индексирования). Более подробно этот стиль будет рассмотрен в главах 20 и 21.

19.3.6. Вывод аргументов шаблона

Создавая объект конкретного класса на основе шаблона, мы указываем аргументы шаблона:

```
array<char,1024> buf; // Для buf параметр T – char, а N == 1024
array<double,10> b2;  // Для b2 параметр T – double, а N == 10
```

Для шаблона функции компилятор обычно выводит аргументы шаблона из аргументов функций. Рассмотрим пример.

```
template<typename T, int N> void fill(array<T,N>& b, const T& val)
{
    for (int i = 0; i<N; ++i) b[i] = val;
}

void f()
{
    fill(buf,'x'); // У fill() параметр T – char, N == 1024,
                   // так как эти же параметры есть у объекта buf
    fill(b2,0.0);  // У fill() параметр T – double, N == 10,
                   // так как эти же параметры есть у объекта b2
}
```

С формальной точки зрения вызов **fill(buf,'x')** является сокращенной формой записи **fill<char,1024>(buf,'x')**, а **fill(b2,0)** — сокращенной записью вызова **fill<double,10>(b2,0)**, но, к счастью, мы не всегда обязаны быть столь точными. Компилятор сам выводит эту информацию вместо нас.

19.3.7. Обобщение класса vector

Когда мы создавали обобщенный класс **vector** на основе класса "**vector** элементов типа **double**" и выводили шаблон "**vector** элементов типа **T**", мы не пересматривали определения функций **push_back()**, **resize()** и **reserve()**. Теперь мы обязаны это сделать, поскольку в разделах 19.2.2 и 19.2.3 эти функции были определены на основе предположений, которые были справедливы для типа **double**, но не выполняются для всех типов, которые мы хотели бы использовать в качестве типов элементов вектора.

- Как работать с классом **vector<X>**, если тип **X** не имеет значения по умолчанию?

- Как гарантировать, что элементы вектора будут уничтожены в конце работы с ним?

Должны ли мы вообще решать эти проблемы? Мы могли бы заявить "Не создавайте векторы для типов, не имеющих значений по умолчанию" или "Не используйте векторы для типов с деструкторами так, что это может привести к проблемам". Но для сущности, предназначенной для обобщенного использования, такие ограничения являются довольно раздражающими и создают впечатление, что разработчик не понял задачи или не думал о пользователях. Довольно часто такие подозрения оказываются верными, но разработчики стандартной библиотеки к этой категории не относятся. Для того чтобы повторить стандартный класс **vector**, мы должны решить две указанные выше проблемы.

Можно решить проблему с типами, не имеющими значений по умолчанию, предоставив пользователю возможность задавать это значение самостоятельно.

```
template<typename T>
void vector<T>::resize(int newsize, T def = T());
```

Иначе говоря, используйте в качестве значения по молчанию объект, созданный конструктором **T()**, если пользователь не указал иначе. Рассмотрим пример.

```
vector<double> v1;
v1.resize(100);        // 100 копий объекта double(), т.е. 0.0
v1.resize(200,0.0);    // 200 копий числа 0.0 — упоминание 0.0
                       // в данном случае излишне
v1.resize(300, 1.0);   // Добавляем 300 копий числа 1.0
struct No_default {
    No_default(int);   // Единственный конструктор No_default
    // ...
};

vector<No_default> v2(10);     // Ошибка: попытка создать 10
                               // объектов No_default()
vector<No_default> v3;
v3.resize(100, No_default(2));// Добавляем 100 копий объекта
                               // No_default(2)
v3.resize(200);                // Ошибка: попытка создать 200
                               // No_default()
```

Проблему, связанную с деструктором, устранить труднее. Мы оказываемся в действительно трудной ситуации: структура данных частично состоит из инициализированных данных, а частично — из неинициализированных. До сих пор мы старались избегать неинициализированных данных и ошибок, которые ими порождаются. Теперь как разработчики класса **vector** мы столкнулись с проблемой, с которой раньше как пользователи класса **vector** не сталкивались.

Во-первых, мы должны найти способ получения неинициализированной памяти и работы с ней. К счастью, стандартная библиотека предоставляет класс распределителя памяти **allocator**, распределяющий неинициализированную память. Несколько упрощенный его вариант приведен ниже.

```
template<typename T> class allocator {
public:
    // ...
    T* allocate(int n);      // Выделение памяти для n объектов
                             // типа T
    void deallocate(T* p,    // Освобождение памяти, занятой n
                    int n); // объектами типа T с адреса p

    void construct(T* p,
                   const T& v); // Создание объекта типа T со
                                // значением v по адресу p
    void destroy(T* p);      // Уничтожение объекта T по адресу p
};
```

Если вам нужна полная информация по этому вопросу, обратитесь к книге Страуструпа *The C++ Programming Language* (**<memory>** в разделе B.1.1) или к стандарту языка C++. Здесь же показаны четыре фундаментальные операции, позволяющие выполнять следующие действия.

- Выделение памяти, достаточной для хранения объекта типа **T** без инициализации.
- Создание объекта типа **T** в неинициализированной памяти.
- Уничтожение объекта типа **T** и возвращение памяти в неинициализированное состояние.
- Освобождение неинициализированной памяти размера, достаточного для хранения объекта типа **T**.

Не удивительно, что класс **allocator** — именно то, что нужно для реализации функции **vector<T>::reserve()**. Начнем с того, что включим в класс **vector** параметр, указывающий используемый распределитель памяти.

```
template<typename T, typename A = allocator<T>> class vector {
    A alloc; // Работает с памятью, выделяемой для элементов
    // ...
};
```

Кроме распределителя памяти, используемого вместо оператора **new**, остальная часть описания класса **vector** не отличается от прежнего. Как пользователи класса **vector** мы можем игнорировать распределители памяти, пока сами не захотим, чтобы класс **vector** управлял памятью для своих элементов нестандартным образом. Как разработчики класса **vector** и как студенты, пытающиеся понять фундаментальные проблемы

и освоить основные методы программирования, мы должны понимать, как вектор работает с неинициализированной памятью, и предоставить пользователям правильно сконструированные объекты. Единственный код, которого должны коснуться изменения, — это код, непосредственно работающий с памятью, такой, как функция **vector<T>::reserve()**.

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;      // Размер не уменьшается
    T* p = alloc.allocate(newalloc); // Выделяем новую память
    // Копирование:
    for(int i=0; i<sz; ++i) alloc.construct(&p[i],elem[i]);
    // Уничтожение:
    for(int i=0; i<sz; ++i) alloc.destroy(&elem[i]);
    alloc.deallocate(elem,space);     // Освобождаем старую память
    elem = p;
    space = newalloc;
}
```

Мы перемещаем элемент в новый участок памяти, создавая копию в неинициализированной памяти, а затем уничтожая оригинал. Здесь нельзя использовать присваивание, потому что для таких типов, как **string**, присваивание подразумевает, что целевая область памяти уже проинициализирована.

Имея функцию **reserve()**, написать функцию **vector<T,A>::push_back()** можно без труда.

```
template<typename T, typename A>
void vector<T,A>::push_back(const T& val)
{
    if (space==0) reserve(8);        // Начинаем с 8 элементов
    else if (sz==space)
        reserve(2*space);            // Выделяем больше памяти
    alloc.construct(&elem[sz],val); // Добавляем в конец
                                     // значение val
    ++sz;                            // Увеличиваем размер
}
```

Аналогично можно написать функцию **vector<T,A>::resize()**.

```
template<typename T, typename A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    // Создаем:
    for(int i=sz; i<newsize; ++i) alloc.construct(&elem[i],val);
    // Уничтожаем:
    for(int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]);
    sz = newsize;
}
```

Обратите внимание на то, что, поскольку некоторые типы не имеют конструкторов по умолчанию, мы снова предоставляем возможность задавать начальное значение для новых элементов.

Другое новшество — деструкция "избыточных элементов" при уменьшении вектора. Думайте о деструкторе как о чем-то, превращающем объект определенного типа в "просто память".

Непринужденное обращение с распределителями памяти — это довольно сложное и хитроумное искусство. Не старайтесь злоупотреблять им, пока не почувствуете, что готовы стать экспертом.

19.4. Проверка диапазона и исключения

Мы проанализировали текущее состояние нашего класса **vector** и обнаружили (с ужасом?), что при обращении к элементам в нем не предусмотрена проверка выхода за пределы допустимого диапазона. Реализация оператора **operator[]** представляет собой просто

```
template<typename T, typename A>
T& vector<T,A>::operator[](int n)
{
    return elem[n];
}
```

Рассмотрим следующий пример:

```
vector<int> v(100);
v[-200] = v[200]; // Проблема!
int i;
cin>>i;
v[i] = 999;       // Повреждение произвольной ячейки памяти
```

Этот код компилируется и выполняется, обращаясь при этом к памяти, не принадлежащей нашему объекту класса **vector**. Это может привести к большим неприятностям! В реальной программе такой код неприемлем. Попробуем улучшить наш класс **vector**, чтобы решить эту проблему. Простейший способ — добавить в класс операцию проверки доступа с именем **at()**.

```
struct out_of_range { /* ... */ }; // Класс для сообщений об
                                   // ошибках выхода за пределы диапазона

template<typename T, typename A = allocator<T>> class vector {
    // ...
    T& at(int n);                       // Доступ с проверкой
    const T& at(int n) const;           // Доступ с проверкой

    T& operator[](int n);               // Доступ без проверки
    const T& operator[](int n) const; // Доступ без проверки
    // ...
};
```

```
template<typename T, typename A> T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<typename T, typename A>
T& vector<T,A>::operator[](int n)      // Как и раньше
{
    return elem[n];
}
```

Теперь мы можем написать следующий код.

```
void print_some(vector<int>& v)
{
    int i = -1;
    while(cin>>i && i!=-1)
    try {
        cout << "v[" << i << "]==" << v.at(i) << "\n";
    }
    catch(out_of_range) {
        cout << "неверный индекс: " << i << "\n";
    }
}
```

Здесь мы используем функцию `at()`, чтобы обеспечить доступ к элементам с проверкой выхода за пределы допустимого диапазона, и перехватываем исключение `out_of_range` при недопустимом обращении к элементу вектора.

Основная идея заключается в использовании операции индексирования `[]`, если нам заранее известно, что индекс правильный, и функции `at()`, если возможен выход за пределы допустимого диапазона.

19.4.1. Отступление от темы: вопросы проектирования

Пока что все хорошо, но почему бы нам не включить проверку выхода за пределы допустимого диапазона в `operator[]()`? Тем не менее, как показано выше, стандартный класс `vector` содержит отдельную функцию `at()` с проверкой доступа и `operator[]()` без проверки. Попробуем обосновать это решение. Оно основывается на четырех аргументах.

1. *Совместимость*. Люди использовали индексирование без проверки выхода за пределы допустимого диапазона задолго до того, как в языке C++ появились исключения.
2. *Эффективность*. Можно создать оператор с проверкой выхода за пределы допустимого диапазона на основе оптимально эффективного оператора индексирования без такой проверки, но невозможно создать оператор индексирования без проверки выхода за пределы

допустимого диапазона, обладающий оптимальным быстродействием, на основе оператора доступа, выполняющего такую проверку.

3. *Ограничения*. В некоторых средах исключения не допускаются.
4. *Необязательность*. На самом деле стандарт не утверждает, что вы не можете проверить диапазон в классе `vector`, поэтому, если вы хотите выполнять проверку, можете использовать реализацию, которая проверяет выход за пределы диапазона.

19.4.1.1. Совместимость

Люди очень, очень не любят переделывать старый код. Например, если у вас имеется миллион строк кода, то было бы очень дорого переделывать его полностью, чтобы корректно использовать исключения. Мы могли бы сказать, что после такой переделки код станет лучше, но не станем этого делать, поскольку не одобряем излишние затраты времени и денег. Более того, люди, занимающиеся сопровождением существующего кода, обычно утверждают, что, хотя в принципе код без проверки небезопасен, их конкретная программа была протестирована и используется уже многие годы, так что в ней уже выявлены все ошибки. К таким аргументам можно относиться скептически, но в каждом конкретном случае следует принимать взвешенное решение. Естественно, нет никаких программ, которые использовали стандартный класс `vector` до того, как он появился в языке C++, но существуют миллионы строк кода, в которых используются очень похожие классы, но (будучи написанными до принятия стандартов) без исключений. Большинство этих программ впоследствии было переделано с учетом стандарта.

19.4.1.2. Эффективность

Да, проверка выхода за пределы диапазона в экстремальных случаях, таких как буфера сетевых интерфейсов и матрицы в высокопроизводительных научных вычислениях, может оказаться слишком сложной. Однако стоимость проверки выхода за пределы допустимого диапазона редко учитывается при обычных вычислениях, которые выполняются в большинстве случаев. Таким образом, мы рекомендуем использовать в классе `vector` проверку выхода за пределы допустимого диапазона при малейшей возможности.

19.4.1.3. Ограничения

В этом пункте, как и в предыдущем, аргументы нельзя считать универсальными. Несмотря на то что они разделяются практически всеми программистами и не могут быть просто отброшены, если вы начинаете писать новую программу в среде, не связанной с работой в реальном времени

(см. раздел 25.2.1), то используйте обработку ошибок с помощью исключений и векторы с проверкой выхода за пределы допустимого диапазона.

19.4.1.4. Необязательность

Стандарт ISO C++ утверждает, что выход за пределы допустимого диапазона вектора не имеет никакой определенной семантики, а потому его следует избегать. Генерация исключения при попытке выхода за пределы допустимого диапазона идеально вписывается в стандарт. Так что если вы хотите, чтобы класс **vector** генерировал исключения, и три рассмотренных выше пункта не относятся к вашему конкретному приложению, следует использовать класс **vector** с проверкой выхода за пределы допустимого диапазона. Именно этого принципа мы придерживаемся в нашей книге.

Короче говоря, реальный дизайн может оказаться запутаннее, чем хотелось бы, но зато у нас есть возможность справиться с проблемами.

19.4.2. Признание в использовании макроса

Как и наш класс **vector**, большинство реализаций стандартного класса **vector** не гарантирует проверку выхода за пределы допустимого диапазона при использовании оператора индексирования (**[]**), а вместо этого содержит функцию **at()**, выполняющую такую проверку. Так откуда же в наших программах возникали исключения **std::out_of_range**? На самом деле мы выбрали вариант 4 из раздела 19.4.1: реализация класса **vector** не обязана проверять выход за пределы допустимого диапазона с помощью оператора **[]**, но ей не запрещено делать это, и мы решили воспользоваться данной возможностью. То, что вы используете в своей отладочной версии, на самом деле называется **Vector**, и в нем имеется проверка в операторе **[]**. Это позволяет сократить время отладки и поиска ошибок за счет небольшой потери производительности программы.

```
struct Range_error:out_of_range { // Расширенное сообщение о
                                  // выходе за пределы диапазона
    int index;
    Range_error(int i):out_of_range{"Range error"},index{i}{}
};

template<typename T> struct Vector: public std::vector<T> {
    using size_type = typename std::vector<T>::size_type;
    using vector<T>::vector;      // Использование конструкторов
                                  // vector<T> (раздел 20.5)

    T& operator[](size_type i)    // Вместо возврата at(i);
    {
        if (i<0 || this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
```

```
    }

    const T& operator[](size_type i) const
    {
        if (i<0 || this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
};
```

Мы используем класс **Range_error**, чтобы облегчить отладку операции индексирования. Порождение нашего класса из **std::vector** приводит к наличию в нашем классе **Vector** всех функций-членов стандартного вектора. Первая конструкция **using** вводит удобный синоним для **size_type** из **std::vector**; см. раздел 20.5. Вторая конструкция **using** дает в качестве конструкторов **Vector** все конструкторы стандартного вектора.

Класс **Vector** очень полезен для отладки нетривиальных программ. В качестве альтернативы нам пришлось бы использовать реализацию стандартного класса **vector**, предусматривающую систематическую проверку (*возможно*, именно такую реализацию вы и используете); у нас нет никакого способа узнать, насколько высока степень проверок, обеспечиваемых вашими компилятором и библиотекой (поскольку это выходит за рамки стандарта).

В заголовочном файле **std_lib_facilities.h** мы используем ужасный трюк (макроподстановку), переопределяя смысл слова **vector** — теперь оно означает **Vector**.

```
// Отвратительный макрос, чтобы получить вектор
// с проверкой выхода за пределы диапазона
#define vector Vector
```

Это означает, что там, где вы написали слово **vector**, компилятор увидит слово **Vector**. Этот трюк ужасен тем, что вы видите не тот код, который видит компилятор. В реальных программах макросы являются источником довольно большого количества запутанных ошибок (см. разделы 27.8 и A.17.2).

Мы сделали то же самое, чтобы реализовать проверку выхода за пределы допустимого диапазона в классе **string**.

К сожалению, не существует стандартного, переносимого и ясного способа получить проверку выхода за пределы допустимого диапазона из реализации операции **[]** в классе **vector**. Однако эту проверку в классах **vector** и **string** можно реализовать намного точнее и полнее, чем это сделано нами. Правда, обычно это связано с заменой реализации стандартной библиотеки, изменением настроек установки или с вмешательством в исходный текст стандартной библиотеки. Но ни одна из этих возможностей не годится для новичков, приступающих к программированию, а класс **string** мы использовали уже в главе 2.

19.5. Ресурсы и исключения

Таким образом, объект класса **vector** может генерировать исключения, как мы и рекомендуем: если функция не может выполнить требуемое действие, чтобы сообщить вызывающему коду об этом, она должна генерировать исключение (см. главу 5). Теперь настало время подумать, как написать код, обрабатывающий исключения, сгенерированные операторами класса **vector** и другими функциями. Наивный ответ — "для перехвата исключения используйте блок **try**, выводите сообщение об ошибке и прекращайте выполнение программы" — слишком груб для большинства нетривиальных систем.

Один из фундаментальных принципов программирования заключается в том, что, если мы запрашиваем ресурс, то должны — непосредственно или косвенно — вернуть его системе. Примерами ресурсов являются

- память,
- блокировки,
- дескрипторы файлов,
- дескрипторы потоков,
- сокеты,
- окна.

По сути, ресурс — это нечто, что можно получить и необходимо вернуть (освободить) самостоятельно или с помощью некоторого менеджера ресурса. Простейшим примером ресурса является динамическая память, которую мы выделяем, используя оператор **new**, и возвращаем с помощью оператора **delete**, например:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // Выделяем память
    // ...
    delete[] p;          // Освобождаем память
}
```

Как мы видели в разделе 17.4.6, следует помнить о необходимости освободить память, что не всегда просто выполнить. Исключения еще больше усугубляют ситуацию, и в результате из-за невежества или небрежности может возникнуть утечка ресурсов. В качестве примера можно указать функцию **suspicious()**, которая явно использует оператор **new** и присваивает результирующий указатель локальной переменной, создавая опасную ситуацию.

Объект, такой как вектор, который отвечает за освобождение ресурса, называется *владельцем* (owner) или *дескриптором* (handle) ресурса, за который он отвечает.

19.5.1. Потенциальные проблемы управления ресурсами

Рассмотрим одну из опасностей, таящуюся в, казалось бы, безвредном присваивании указателей:

```
int* p = new int[s]; // Выделяем память
```

Она заключается в трудности проверки того, что данному оператору **new** соответствует оператор **delete**. В функции **suspicious()** есть инструкция **delete[]p**, которая могла бы освободить память, но легко представить себе несколько причин, по которым это может и не произойти. Какие инструкции можно было бы вставить в часть, указанную многоточием, чтобы вызвать утечку памяти? Примеры, которые мы подобрали для иллюстрации возникающих проблем, должны натолкнуть вас на размышления и вызвать опасения относительно такого кода. Кроме того, благодаря этим примерам вы оцените простоту и мощь альтернативного решения.

Возможно, указатель **p** больше не указывает на объект, который мы хотим уничтожить с помощью оператора **delete**:

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // Выделение памяти
    // ...
    if (x) p = q;        // Переназначение указателя p
    // ...
    delete[] p;          // Освобождение памяти
}
```

Мы включили в программу инструкцию **if (x)**, чтобы гарантировать, что вы не будете знать заранее, изменилось ли значение указателя **p**. Возможно, программа вообще никогда не выполнит оператор **delete**.

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // Выделение памяти
    // ...
    if (x) return;
    // ...
    delete[] p;          // Освобождение памяти
}
```

Возможно, программа никогда не выполнит оператор **delete**, потому что сгенерирует исключение.

```
void suspicious(int s, int x)
{
```

```
    int* p = new int[s]; // Выделение памяти
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);
    // ...
    delete[] p;          // Освобождение памяти
}
```

Последняя возможность беспокоит нас больше всего. Когда люди впервые сталкиваются с такой проблемой, они считают, что она связана с исключениями, а не с управлением ресурсами. Не понимая истинных причин проблемы, они пытаются перехватывать исключения.

```
void suspicious(int s, int x) // Плохой код!!
{
    int* p = new int[s];      // Выделение памяти
    vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
    // ...
    } catch (...) {           // Перехват всех исключений
        delete[] p;           // Освобождение памяти
        throw;                // Повторная генерация исключения
    }
    // ...
    delete[] p;               // Освобождение памяти
}
```

Этот код решает проблему за счет дополнительных инструкций и дублирования кода, освобождающего ресурсы (в данном случае инструкции `delete[]p;`). Иначе говоря, это некрасивое решение; что еще хуже — его сложно обобщить. Представим, что мы задействовали несколько ресурсов.

```
void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int>v1;
    // ...
    int* q = new int[s];
    vector<double> v2;
    // ...
    delete[] p;
    delete[] q;
}
```

Обратите внимание на то, что, если оператор `new` не сможет выделить динамическую память, он сгенерирует стандартное исключение `bad_alloc`. Метод `try...catch` в этом примере также успешно работает, но нам потребуется несколько блоков `try`, и код станет повторяющимся и просто ужасным. Мы не любим повторяющиеся и запутанные программы, потому что

повторяющийся код сложно сопровождать, а запутанный код не только сложно сопровождать, но и вообще трудно понять.

ПОПРОБУЙТЕ

Добавьте блоки `try` в последний пример и убедитесь, что все ресурсы будут правильно освобождаться при любых исключениях.

19.5.2. Захват ресурса — это инициализация

К счастью, нам не обязательно копировать инструкции `try...catch`, чтобы предотвратить утечку ресурсов. Рассмотрим следующий пример:

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // ...
}
```

Это уже лучше. Что более важно, это *очевидно* лучше. Ресурс (в данном случае динамическая память) выделяется конструктором и освобождается соответствующим деструктором. Теперь мы действительно решили нашу конкретную проблему, связанную с исключениями. Это решение носит универсальный характер; его можно применить ко всем видам ресурсов: конструктор получает ресурсы для объекта, который ими управляет, а соответствующий деструктор их возвращает. Примерами такого подхода могут служить блокировки баз данных, сокеты или буфера ввода-вывода (работу с последними обеспечивает класс `iostream`). Соответствующий принцип обычно формулируется несколько неуклюже: "Захват ресурса есть инициализация" ("Resource Acquisition Is Initialization" — RAII).

Рассмотрим приведенный выше пример. Как только мы выйдем из функции `f()`, будут вызваны деструкторы векторов `p` и `q`: поскольку переменные `p` и `q` не являются указателями, мы не можем присваивать им новые значения, инструкция `return` не может предотвратить вызов деструкторов и никакие исключения не генерируются. Это универсальное правило: когда поток управления покидает область видимости, вызываются деструкторы для каждого полностью созданного объекта и подобъекта. Объект считается полностью созданным, если его конструктор закончил свою работу. Исследование всех следствий, вытекающих из этих двух утверждений, может вызвать головную боль. Будем считать просто, что конструкторы и деструкторы вызываются, когда надо и где надо.

В частности, если хотите выделить в области видимости динамическую память неконстантного размера, мы рекомендуем использовать класс `vector`, а не явные операторы `new` и `delete`.

19.5.3. Гарантии

Что делать, если вектор невозможно ограничить только одной областью (или подобластью) видимости? Рассмотрим пример.

```
vector<int>* make_vec()      // Создание заполненного вектора
{
    vector<int>* p =
        new vector<int>;     // Выделение динамической памяти
    // ... Заполнение вектора данными;
    // возможна генерация исключения ...
    return p;
}
```

Это довольно распространенный пример: мы вызываем функцию, чтобы создать сложную структуру данных, и возвращаем эту структуру как результат. Однако, если при заполнении вектора возникнет исключение, функция **make_vec()** потеряет этот объект класса **vector**. Кроме того, если функция успешно завершит работу, то кто-то должен будет удалить объект, возвращенный функцией **make_vec()** (см. раздел 17.4.6).

Для того чтобы обработать возможность генерации исключения, мы можем добавить блок **try**.

```
vector<int>* make_vec()      // Создание заполненного вектора
{
    vector<int>* p =
        new vector<int>;     // Выделение динамической памяти
    try {
        // ... Заполнение вектора данными;
        // возможна генерация исключения ...
        return p;
    }
    catch(...) {
        delete p;  // Освобождение памяти
        throw;     // повторная генерация исключения, чтобы
                   // вызывающая функция знала, что функция
                   // make_vec() не выполнила свою задачу
    }
}
```

Данная функция **make_vec()** иллюстрирует очень распространенный стиль обработки ошибок: программа пытается выполнить свое задание, а если не может, то освобождает все локальные ресурсы (в данном случае динамическую память, занятую объектом класса **vector**) и сообщает об этом, генерируя исключение. Здесь исключение генерируется некоторой другой функцией (например, **vector::at()**); функция **make_vec()** просто повторяет генерацию с помощью оператора **throw;**. Это простой и эффективный способ обработки ошибок, который можно применять систематически.

- *Базовая гарантия.* Цель кода `try...catch` состоит в том, чтобы гарантировать, что функция `make_vec()` либо завершит работу успешно, либо сгенерирует исключение без утечки ресурсов. Это часто называют *базовой гарантией* (basic guarantee). Весь код, являющийся частью программы, которая должна восстанавливать свою работу после генерации исключения, должен поддерживать базовую гарантию. Базовую гарантию обеспечивает весь код стандартной библиотеки.

- *Строгая гарантия.* Если, кроме базовой гарантии, функция гарантирует, что все наблюдаемые значения (т.е. все значения, не являющиеся локальными по отношению к этой функции) после отказа восстанавливают свои предыдущие значения, то говорят, что такая функция дает *строгую гарантию* (strong guarantee). Строгая гарантия — это идеал для функции: либо функция будет выполнена так, как ожидалось, либо ничего не произойдет, кроме генерации исключения, означающего отказ.

- *Гарантия отсутствия исключений* (no-throw guarantee). Если бы мы не могли выполнять простые операции без какого бы то ни было риска сбоя и генерации исключений, то не могли бы написать код, соответствующий условиям базовой и строгой гарантии. К счастью, практически все встроенные средства языка C++ поддерживают гарантию отсутствия исключений: они просто не могут их генерировать. Для того чтобы избежать генерации исключений, просто избегайте операторов `throw` и `new` и не применяйте оператор `dynamic_cast` к ссылочным типам (раздел A.5.7).

Для анализа правильности программы наиболее полезными являются базовая и строгая гарантии. Принцип RAII играет существенную роль для реализации простого и эффективного кода, написанного в соответствии с этими идеями.

Естественно, всегда следует избегать неопределенных (и обычно опасных) операций, таких как разыменования нулевого указателя, деление на нуль и выход за пределы допустимого диапазона. Перехват исключений не спасает от нарушений фундаментальных правил языка.

19.5.4. Класс `unique_ptr`

Итак, функции, такие как `make_vec()`, подчиняются основным правилам корректного управления ресурсами с использованием исключений. Это обеспечивает выполнение базовой гарантии, которую должны давать все правильные функции при восстановлении работы после исключений. Если с нелокальными данными в той части программы, которая ответственна за заполнение вектора данными, ничего не происходит, то можно

даже утверждать, что такая функция обеспечивает строгую гарантию. Однако блок **try...catch** по-прежнему выглядит ужасно. Решение очевидно: нужно вновь как-то применить идиому RAII; иначе говоря, необходимо предусмотреть объект, который будет владеть объектом класса **vector<int>** и сможет его удалить, если возникнет исключение. В заголовочном файле **<memory>** стандартной библиотеки на этот случай предоставлен класс **unique_ptr**.

```
vector<int>* make_vec() // Создание заполненного вектора
{
    unique_ptr< vector<int>> p{new vector<int>}; // Выделение
                                                 // динамической памяти
    // ... заполнение вектора данными;
    // возможна генерация исключения ...
    return p.release(); // Возврат указателя, которым владеет объект p
}
```

Объект **unique_ptr** представляет собой объект, который хранит указатель. Мы инициализируем его указателем, полученным с помощью оператора **new**. Мы можем использовать операторы **->** и ***** с **unique_ptr** в точности так же, как и со встроенным указателем (например, **p->at(2)** или **(*p).at(2)**), так что можно представить себе **unique_ptr** как разновидность указателя. Однако **unique_ptr** владеет объектом, на который указывает: при уничтожении **unique_ptr** он удаляет с помощью оператора **delete** объект, на который указывает. Это означает, что если во время заполнения вектора будет сгенерировано исключение или произойдет преждевременный выход из функции **make_vec()**, объект **vector<int>** будет корректно уничтожен. Вызов **p.release()** извлекает содержащийся в объекте **p** указатель (на **vector<int>**), так что его можно вернуть из функции, и одновременно делает объект **p** хранящим **nullptr**, так что уничтожение **p** (которое выполняется инструкцией **return**) не уничтожит никакого объекта.

Применение **unique_ptr** существенно упрощает функцию **make_vec()**. По сути, функция **make_vec()** становится такой же простой, как наивная и небезопасная версия. Что важно, наличие **unique_ptr** позволяет нам повторить наши рекомендации смотреть на явные блоки **try** с подозрением; большинство из них — как в функции **make_vec()** — могут быть заменены некоторым вариантом идиомы RAII.

Версия **make_vec()**, использующая **unique_ptr**, прекрасна — с одним исключением: она все еще возвращает указатель, так что кто-то все еще может забыть удалить его с помощью оператора **delete**. Проблема решается путем возврата из функции объекта **unique_ptr**:

```
unique_ptr<vector<int>> make_vec() // Создание заполненного вектора
{
    unique_ptr<vector<int>>
        p{new vector<int>};        // Выделение памяти
```

```
    // ... заполнение вектора данными;
    // возможна генерация исключения ...
    return p;
}
```

Класс **unique_ptr** очень похож на обычный указатель, но он имеет одно важное ограничение: вы не можете присвоить один **unique_ptr** другому, чтобы получить два **unique_ptr**, указывающих на один и тот же объект. Это должно быть именно так, иначе может возникнуть неразбериха в том, какой из **unique_ptr** владеет объектом и должен его удалить:

```
void no_good()
{
    unique_ptr<X> p { new X };
    unique_ptr<X> q {p}; // Ошибка: к счастью
    // ...
}   // Здесь и p, и q удаляют X
```

Если вам нужен "интеллектуальный" указатель, который гарантирует удаление и может быть скопирован, используйте **shared_ptr** (раздел Б.6.5). Однако это более тяжеловесное решение, включающее наличие счетчика, обеспечивающего удаление объекта при удалении последней копии указателя, указывающего на него.

Указатель **unique_ptr** обладает тем интересным свойством, что в нем отсутствуют накладные расходы по сравнению с обычным указателем.

19.5.5. Возврат посредством перемещения

Методика возврата больших объемов информации путем ее размещения в динамической памяти и возврата указателя на эту память весьма распространена. И она же является источником огромного количества ошибок, связанных с управлением памятью: кто удаляет возвращенный функцией указатель и освобождает динамическую память? Будет ли корректно удален указатель при генерации исключения? При отсутствии систематической стратегии работы с указателями (или использования "интеллектуальных" указателей **unique_ptr** и **shared_ptr**) ответы на эти вопросы будут в лучшем случае "ну, я так думаю", чего, конечно же, недостаточно.

К счастью, когда мы добавили в наш **vector** перемещающие операции, мы решили указанную проблему для данного класса: надо просто использовать перемещающий конструктор для передачи владения из функции.

```
vector<int> make_vec() // Создание заполненного вектора
{
    vector<int> res;
    // ... заполнение вектора данными; возможна генерация исключения ...
    return res;         // Перемещающий конструктор эффективно
                        // передает владение
}
```

Эта (последняя) версия функции `make_vec()` является простейшей, и мы рекомендуем именно ее. Перемещающее решение обобщается на все контейнеры и далее — на все дескрипторы ресурсов. Например, `fstream` использует эту методику для отслеживания дескрипторов файлов. Перемещающее решение является простым и общим. Использование дескрипторов ресурсов упрощает код и устраняет огромный источник ошибок. Накладные расходы времени выполнения по сравнению с использованием указателей для таких дескрипторов либо отсутствуют вовсе, либо невелики и предсказуемы.

19.5.6. RAII для класса `vector`

Даже использование интеллектуальных указателей, таких как `unique_ptr`, может показаться недостаточно безопасным. Как мы можем быть уверены, что выявили все указатели, требующие защиты? Как мы можем быть уверены, что освободили все указатели на объекты, которые не должны уничтожаться в конце области видимости? Рассмотрим функцию `reserve()` из раздела 19.3.7.

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;          // Размер не уменьшается
    T* p = alloc.allocate(newalloc);      // Выделение новой памяти

    for (int i=0; i<sz; ++i)
        alloc.construct(&p[i],elem[i]); // Копирование

    for (int i=0; i<sz; ++i)
        alloc.destroy(&elem[i]);          // Уничтожение

    alloc.deallocate(elem,space);         // Освобождение памяти
    elem = p;
    space = newalloc;
}
```

Обратите внимание на то, что операция копирования старого элемента `alloc.construct(&p[i],elem[i])` может генерировать исключение. Следовательно, указатель `p` — это пример проблемы, о которой мы предупреждали в разделе 19.5.1. Да, но ведь мы можем решить проблему с помощью `unique_ptr`? Но еще лучше — вернуться и понять, что память для вектора — это ресурс; иначе говоря, мы можем определить класс `vector_base` для выражения фундаментальной концепции, которую используем все время. Эта концепция изображена на следующем рисунке. на котором показаны три элемента, определяющие использование памяти вектором.

Добавив для полноты картины распределитель памяти, получим следующий код:

```
template<typename T, typename A>
struct vector_base {
    A alloc;     // Распределитель памяти
    T* elem;     // Начало выделенной памяти
    int sz;      // Количество элементов
    int space;   // Размер выделенной памяти

    vector_base(const A& a, int n)
        :alloc{a}, elem{a.allocate(n)},sz{n},space{n}{}
    ~vector_base() { alloc.deallocate(elem,space); }
};
```

Обратите внимание на то, что класс **vector_base** работает с памятью, а не с типизированными объектами. Нашу реализацию класса **vector** можно использовать для хранения объектов, имеющих требуемый тип. По сути, класс **vector** — это просто удобный интерфейс для класса **vector_base**.

```
template<typename T, typename A = allocator<T>>
class vector : private vector_base<T,A> {
public:
    // ...
};
```

Теперь можно переписать функцию **reserve()**, сделав ее более простой и корректной.

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=this->space) return;// Размер не уменьшается
    vector_base<T,A>
        b(this->alloc,newalloc);       // Выделение новой памяти
    uninitialized_copy(b.elem,
        &b.elem[this->sz],this->elem);// Копирование
    for(int i=0; i<this->sz; ++i)
        this->alloc.destroy(
            &this->elem[i]);           // Удаление старых объектов
    swap<vector_base<T,A>>(*this,b);  // Обмен представлений
}
```

Для создания копий элементов из `b` мы использовали функцию стандартной библиотеки `uninitialized_copy`, поскольку она корректно обрабатывает исключения копирующих конструкторов элементов и поскольку вызов функции проще, чем написание цикла. При выходе из функции `reserve()` старая память автоматически освобождается деструктором класса `vector_base`, если операция копирования была успешна. Если же выход осуществился из-за генерации исключения в операции копирования, вновь выделенная память освобождается. Функция `swap()` является алгоритмом из стандартной библиотеки (заголовочный файл `<algorithm>`), обменивающим значения двух объектов. Мы использовали алгоритм `swap<vector_base<T,A>>(*this,b)`, а не более простую функцию `swap(*this,b)`, поскольку объекты `*this` и `b` имеют разные типы (`vector` и `vector_base` соответственно), и мы должны явно указать, какую специализацию алгоритма `swap` следует вызвать. Аналогично мы явно указываем `this->` при обращении к члену базового класса `vector_base<T,A>` из члена порожденного класса `vector_base<T,A>`, такого как `vector<T,A>::reserve()`.

ПОПРОБУЙТЕ

Модифицируйте функцию `reserve`, чтобы она использовала класс `unique_ptr`. Помните о необходимости освободить память перед возвратом из функции. Сравните это решение с классом `vector_base`. Выясните, какое из них проще написать, а какое проще сделать корректно работающим.

Задание

1. Определите класс `template<typename T> struct S {T val;};`.
2. Добавьте конструктор, чтобы можно было инициализировать его типом `T`.
3. Определите переменные типов `S<int>`, `S<char>`, `S<double>`, `S<string>` и `S<vector<int>>`; инициализируйте их значениями по своему выбору.
4. Прочитайте эти значения и выведите их на экран.
5. Добавьте шаблонную функцию `get()`, возвращающую ссылку на переменную `val`.
6. Разместите функцию `get()` за пределами класса.
7. Сделайте член `val` закрытым.
8. Выполните п. 4, используя функцию `get()`.
9. Добавьте шаблонную функцию `set()`, чтобы можно было изменять значение члена `val`.

10. Замените функцию `set()` оператором `S<T>::operator=(const T&)`. *Указание*: это гораздо проще, чем в разделе 19.2.5.
11. Напишите константную и неконстантную версии функции `get()`.
12. Определите функцию `template<typename T> read_val(T&v)`, выполняющую ввод данных из потока `cin` в переменную `v`.
13. Используйте функцию `read_val()`, чтобы считать данные в каждую из переменных, перечисленных в п. 3, за исключением переменной `S<vector<int>>`.
14. Бонус: определите операторы ввода и вывода (`>>` и `<<`) для `vector<T>`. И для ввода, и для вывода используйте формат `{val, val, val}`. Это позволит функции `read_val()` работать с переменной `S<vector<int>>`.

Не забудьте выполнить тестирование после каждого этапа.

Контрольные вопросы

1. Зачем нужно изменять размер вектора?
2. Зачем нужны векторы с разными типами элементов?
3. Почему бы просто не выделить для вектора объем памяти, достаточно большой для любых вычислений?
4. Сколько зарезервированной памяти мы выделяем для нового вектора?
5. Когда мы копируем элементы вектора в новое место в памяти?
6. Какие операции класса `vector` могут изменять размер вектора после его создания?
7. Чему равно значение `vector` после копирования?
8. Какие две операции определяют копирование вектора?
9. Какой смысл имеет копирование объектов класса по умолчанию?
10. Что такое шаблон?
11. Назовите два самых полезных типа аргументов шаблонов?
12. Что такое обобщенное программирование?
13. Чем обобщенное программирование отличается от объектно-ориентированного программирования?
14. Чем класс `array` отличается от класса `vector`?
15. Чем класс `array` отличается от массива встроенного типа?
16. Чем функция `resize()` отличается от функции `reserve()`?
17. Что такое ресурс? Дайте определение и приведите примеры.
18. Что такое утечка ресурсов?
19. Что такое идиома RAII? Какие проблемы она решает?
20. Для чего предназначен класс `unique_ptr`?

Термины

`#define`	`unique_ptr`	параметр шаблона
`at()`	базовая гарантия	перемещение
`push_back()`	владелец	повторная генерация исключения
RAII	гарантии безопасности	ресурс
`resize()`	дескриптор	самоприсваивание
`shared_ptr`	инстанцирование	специализация
`this`	исключение	строгая гарантия
`throw`	макрос	шаблон

Упражнения

В каждом из упражнений создайте и проверьте (с выводом на печать) набор объектов определенных классов и продемонстрируйте, что ваш проект и реализация действительно работают так, как вы ожидали. Там, где задействованы исключения, может потребоваться тщательное обдумывание мест, где могут появиться ошибки.

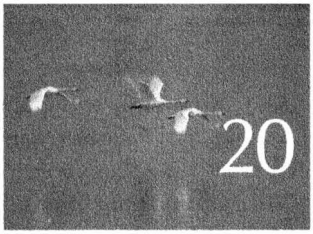
1. Напишите шаблонную функцию `f()`, складывающую элементы одного вектора `vector<T>` с элементами другого такого вектора; например, `f(v1,v2)` должна выполнять суммирование `v1[i]+=v2[i]` для каждого элемента `v1`.
2. Напишите шаблонную функцию, получающую в качестве аргументов объекты типов `vector<T> vt` и `vector<U> vu` и возвращающую сумму всех произведений `vt[i]*vu[i]`.
3. Напишите шаблонный класс `Pair`, содержащий пары значений любого типа. Используйте его для реализации простой таблицы символов, такой как в калькуляторе из раздела 7.8.
4. Превратите класс `Link` из раздела 17.9.3 в шаблонный, в котором аргументом типа является тип элемента. Затем выполните заново упр. 13 из главы 17 с классом `Link<God>`.
5. Определите класс `Int`, содержащий единственный член типа `int`. Определите конструкторы, оператор присваивания и операторы `+`, `-`, `*` и `/`. Протестируйте этот класс и при необходимости усовершенствуйте его дизайн (например, определите операторы `<<` и `>>` для удобства ввода-вывода).
6. Повторите предыдущее упражнение с классом `Number<T>`, где `T` может быть любым числовым типом. Попытайтесь добавить в класс `Number` оператор `%` и посмотрите, что произойдет, когда вы попробуете применить оператор `%` к типам `Number<double>` и `Number<int>`.
7. Примените решение упр. 2 к нескольким объектам типа `Number`.

8. Реализуйте распределитель памяти (см. раздел 19.3.7), используя функции `malloc()` и `free()` (раздел Б.11.4). Создайте класс `vector` так, как описано в конце раздела 19.4, для работы с несколькими простыми тестовыми примерами. *Указание*: поищите "размещающий `new`" и "явный вызов деструктора" в полном справочнике по C++.
9. Повторите реализацию функции `vector::operator=()` (раздел 19.2.5), используя распределитель памяти (раздел 19.3.7) для управления памятью.
10. Реализуйте простой класс `unique_ptr`, содержащий только конструктор, деструктор, операторы `->` и `*`, а также функцию `release()`. В частности, не пытайтесь реализовать присваивание или копирующий конструктор.
11. Разработайте и реализуйте указатель `counted_ptr<T>`, владеющий указателем на объект типа `T`, и указатель на "счетчик ссылок" (типа `int`), совместно используемый всеми указателями на один и тот же объект типа `T`. Счетчик ссылок должен содержать количество указателей, указывающих на данный объект типа `T`. Конструктор `counted_ptr` должен размещать в динамической памяти объект типа `T` и счетчик ссылок. Конструктор `counted_ptr` должен получать аргумент, используемый в качестве начального значения указателя на элемент типа `T`. После уничтожения последнего объекта `counted_ptr` для данного объекта типа `T` его деструктор должен удалить этот объект. Предусмотрите в классе `counted_ptr` операции, позволяющие использовать его как указатель. Это пример так называемого "интеллектуального указателя", который используется для того, чтобы гарантировать, что объект не будет уничтожен, пока последний пользователь не закончит его использовать. Напишите набор тестов для класса `counted_ptr`, используя его объекты в качестве аргументов при вызове функций, в качестве элементов контейнеров и т.д.
12. Определите класс `File_handle`, конструктор которого получает аргумент типа `string` (имя файла) и открывает файл, а деструктор закрывает этот файл.
13. Напишите класс `Tracer`, в котором конструктор и деструктор выводят на экран строки. Передайте эти строки конструктору в качестве аргументов. Используйте этот пример для того, чтобы посмотреть, как работает идиома RAII (например, поэкспериментируйте с объектами класса `Tracer` в качестве локальных объектов, объектов-членов класса, глобальных объектов, объектов, размещенных в динамической памяти с помощью оператора `new`, и т.д. Затем добавьте копирующий конструктор и копирующее присваивание, чтобы можно было увидеть поведение объектов класса `Tracer` в процессе копирования).

14. Разработайте графический пользовательский интерфейс и средства вывода для игры "Охота на Вампуса" (см. главу 18). Предусмотрите ввод данных из окна редактирования и вывод в окне карты той части пещеры, которая в настоящий момент известна игроку.
15. Модифицируйте программу из предыдущего упражнения, чтобы дать пользователю возможность помечать комнаты, основываясь на знаниях и догадках, таких как "могут быть летучие мыши" и "бездонная пропасть".
16. Иногда желательно, чтобы пустой вектор был как можно более маленьким. Например, кто-то может интенсивно использовать класс `vector<vector<vector<int>>>`, в котором большинство векторов пусто. Определите вектор так, чтобы выполнялось условие `sizeof(vector<int>)==sizeof(int*)`, т.е. чтобы класс вектора состоял только из указателя на представление, в котором содержатся массив элементов, количество элементов и указатель `space`.

Послесловие

Шаблоны и исключения представляют собой чрезвычайно мощные языковые возможности. Они поддерживают методы программирования с высокой степенью гибкости — в основном благодаря разделению ответственности, т.е. возможности решать по одной проблеме в каждый конкретный момент времени. Например, используя шаблоны, мы можем определить контейнер, такой как `vector`, отделив его от определения типа элементов. Аналогично исключения позволяют писать код, идентифицирующий ошибки и выдающий сообщения о них, отдельно от кода, предназначенного для их обработки. Третья основная тема данной главы, изменение размера вектора, относительно проста: функции `push_back()`, `resize()` и `reserve()` позволяют отделить определение вектора от указания его размера.



Контейнеры и итераторы

Пишите программы, которые делают что-то одно, но делают это хорошо. Пишите программы, предназначенные для совместной работы.

— Дуг Мак-Илрой (Doug McIlroy)

Эта и следующая главы посвящены библиотеке STL — части стандартной библиотеки языка C++, содержащей контейнеры и алгоритмы. Библиотека STL — это расширяемый каркас для работы с данными в программе на языке C++. После рассмотрения простого примера мы изложим общие идеи и фундаментальные концепции. Мы обсудим понятие итерации, работу со связанными списками и контейнерами из библиотеки STL. Связь между контейнерами (данными) и алгоритмами (обработкой) обеспечивается с помощью ключевых понятий последовательностей и итераторов. В настоящей главе изложены основы для универсальных, эффективных и полезных алгоритмов, описанных в следующей главе. В качестве примера простого приложения рассматривается редактирование текста.

20.1. Хранение и обработка данных

Перед тем как перейти к исследованию крупных коллекций данных, рассмотрим простой пример, иллюстрирующий способы решения большого класса задач, связанных с обработкой данных. Представим себе, что Джек и Джилл измеряют скорость автомобилей, записывая ее в виде чисел с плавающей точкой. Допустим, что Джек программирует на языке C и хранит свои данные в массиве, а Джилл записывает свои измерения в объект класса **vector**. Мы хотели бы использовать их данные в своей программе. Как это сделать?

Потребуем, чтобы программы Джека и Джилл записывали значения в файл, чтобы мы могли считать их в своей программе. В этом случае мы не будем зависеть от выбора структур данных и интерфейсов, сделанных Джеком и Джилл. Довольно часто такая изоляция целиком оправданна. Для ее реализации в наших вычислениях можно использовать приемы ввода, описанные в главах 10 и 11, и класс **vector<double>**.

Однако что делать, если использовать файлы для решения нашей задачи невозможно? Допустим, что код сбора данных оформлен в виде функции, которая каждую секунду поставляет новый набор данных. В таком случае каждую секунду мы будем вызывать функции Джека и Джилл, чтобы получить данные для обработки.

```
double* get_from_jack(int* count); // Джек записывает числа
                                   // типа double в массив,
                                   // возвращая их количество
                                   // в *count
vector<double>* get_from_jill();   // Джилл заполняет вектор
```

```
void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();
    // ... Обработка данных ...
    delete[] jack_data;
    delete jill_data;
}
```

Мы предполагаем, что эти данные хранятся в динамической памяти и их следует удалить после завершения обработки. Другое предположение заключается в том, что мы не можем переписать код, написанный Джеком и Джилл, или не хотим этого делать.

20.1.1. Работа с данными

Очевидно, что этот пример носит слишком упрощенный характер, но он не слишком отличается от множества реальных задач. Если мы сможем элегантно решить эту задачу, то сможем справиться и с огромным множеством других задач программирования. В данной ситуации фундаментальная проблема заключается в том, что мы не можем повлиять на способ хранения данных, который выбрали "поставщики". Наша задача состоит в том, чтобы либо работать с данными в том виде, в котором мы их получаем, либо считать их и сохранить в более удобной форме.

Что мы хотим делать с этими данными? Упорядочить их? Найти наибольшее значение? Вычислить среднее? Найти все значения, большие 65? Сравнить данные Джилл с данными Джека? Определить количество элементов? Возможности бесконечны. Когда мы пишем реальную программу, то просто выполняем требуемые вычисления. В данном случае мы хотим выяснить, как обработать данные и выполнить вычисления с большим массивом чисел. Сначала сделаем нечто совсем простое: найдем наибольший элемент в каждом из наборов данных. Для этого комментарий *...Обработка данных...* надо заменить следующими инструкциями.

```
// ...
double h = -1;
double* jack_high; // Указатель на наибольший элемент
double* jill_high; // Указатель на наибольший элемент

for(int i=0; i<jack_count; ++i)
    if (h<jack_data[i]) {
        jack_high = &jack_data[i];// Адрес наибольшего элемента
        h = jack_data[i];         // Новый "наибольший элемент"
    }

h = -1;
for(int i=0; i< jill_data->size(); ++i)
```

```
    if (h<(*jill_data)[i]) {
        jill_high = &(*jill_data)[i]; // Наибольший элемент
        h = (*jill_data)[i];          // Новый "наибольший элемент"
    }
    cout << "Максимум Джилл: " << *jill_high
         << "; максимум Джека: " << *jack_high;
    // ...
```

Обратите внимание на уродливую конструкцию, используемую для доступа к данным Джилл: `(*jill_data)[i]`. Функция `get_from_jill()` возвращает указатель на вектор `vector<double>*`. Для того чтобы получить данные, мы сначала должны разыменовать указатель, получив доступ к вектору `*jill_data`, а затем применить к нему операцию индексирования. Однако выражение `*jill_data[i]` — не совсем то, что мы хотели; оно означает `*(jill_data[i])`, так как оператор `[]` имеет более высокий приоритет, чем `*`, поэтому нам необходимы скобки вокруг конструкции `*jill_data`, т.е. выражение `(*jill_data)[i]`.

ПОПРОБУЙТЕ

Как бы вы изменили интерфейс, чтобы избежать неуклюжих конструкций, если бы могли изменить код Джилл?

20.1.2. Обобщение кода

Нам нужен единообразный способ доступа и манипуляции данными, чтобы не переписывать программу каждый раз, когда представление данных немного изменяется. Посмотрим на коды Джека и Джилл и попробуем сделать их более абстрактными и единообразными.

Разумеется, все, что мы сделаем с данными Джека, относится и к данным Джилл. Однако между их программами есть два досадных различия: различные способы получения количества данных — `jack_count` и `jill_data->size()`, а также различные конструкции для обращения к данным — `jack_data[i]` и `(*jill_data)[i]`. Последнее различие можно устранить, вводя ссылку:

```
vector<double>& v = *jill_data;
for (int i=0; i<v.size(); ++i)
    if (h<v[i]) {
        jill_high = &v[i];
        h = v[i];
    }
```

Это очень похоже на код для данных Джека. Может быть, имеет смысл написать функцию, которая выполняла бы вычисления как с данными Джилл, так и с данными Джека? Возможны разные пути (см. упр. 3), но по

причинам более высокой степени обобщенности кода, которые станут понятнее после прочтения следующих двух глав, мы выбрали решение, основанное на указателях.

```
double* high(double* first, double* last)
// Возвращает указатель на наибольший
// элемент в диапазоне [first,last)
{
    double h = -1;
    double* high;
    for(double* p = first; p!=last; ++p)
        if (h<*p)
        {
            high = p;
            h = *p;
        }
    return high;
}
```

Теперь можно написать следующий код:

```
double* jack_high = high(jack_data,jack_data+jack_count);
vector<double>& v = *jill_data;
double* jill_high = high(&v[0],&v[0]+v.size());
```

Он выглядит получше. Мы не ввели слишком много переменных и написали только один цикл (в функции `high()`). Если мы хотим найти наибольший элемент, то можем посмотреть на значения `*jack_high` и `*jill_high`:

```
cout << "Максимум Джилл: " << *jill_high
     << "; максимум Джека: " << *jack_high;
```

Обратите внимание на то, что функция `high()` использует тот факт, что вектор хранит данные в массиве, а потому мы можем выразить наш алгоритм поиска максимального элемента в терминах указателей, указывающих на элементы массива.

ПОПРОБУЙТЕ

В этой маленькой программе мы оставили две потенциально опасные ошибки. Одна из них может вызвать катастрофу, а другая приводит к неправильным ответам, если функция `high()` будет использоваться в других программах. Методы обобщенного программирования, описываемые ниже, сделают их очевидными и покажут, как систематически их избегать. Пока же просто найдите их и предложите свои способы их исправления.

Приведенная функция `high()` ограничена в том плане, что она решает одну конкретную задачу.

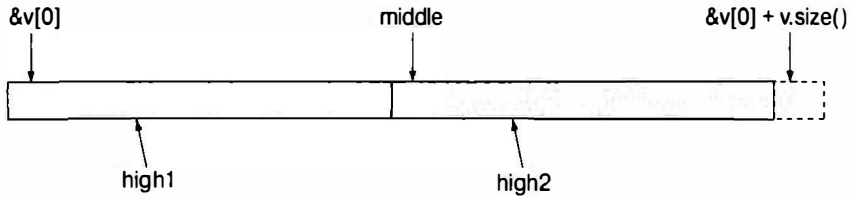
- Она работает только с массивами. Мы считаем, что элементы **vector** хранятся в массиве, но наряду с этим существует множество способов хранения данных, такие как списки и ассоциативные массивы (см. разделы 20.4 и 21.6.1).
- Ее можно применять только к объектам класса **vector** и массивам типа **double**, но не к векторам и массивам с другими типами элементов, например **vector<double*>** или **char[10]**.
- Она находит элемент с наибольшим значением, но с этими данными можно выполнить множество других простых вычислений.

Попробуем обеспечить более высокую общность вычислений над нашими наборами данных.

Обратите внимание на то, что, решив выразить алгоритм поиска наибольшего элемента в терминах указателей, мы "случайно" уже обобщили решение задачи: при желании мы можем найти наибольший элемент массива или вектора, но, помимо этого, можно найти максимальный элемент части массива или вектора. Рассмотрим пример.

```
// ...
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
double* high1 = high(&v[0], middle); // Максимум первой половины
double* high2 = high(middle,
        &v[0]+v.size());             // Максимум второй половины
// ...
```

Здесь указатель **high1** указывает на максимальный элемент первой половины вектора, а указатель **high2** — на максимальный элемент второй половины. Графически это можно изобразить следующим образом.



Мы использовали указатели в качестве аргументов функции **high()**. Это достаточно низкоуровневое обращение к памяти, уязвимое для ошибок. Мы подозреваем, что большинство программистов для поиска максимального элемента в векторе написали бы нечто вроде следующего:

```
double* find_highest(vector<double>& v)
{
    double h = -1;
    double* high = 0;
    for (int i=0; i<v.size(); ++i)
        if (h<v[i])
```

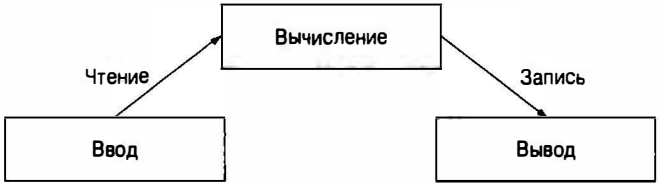
```
        {
            high = &v[i];
            h = v[i];
        }
    return high;
}
```

Однако это не обеспечивает достаточную степень гибкости, которую мы "случайно" уже придали функции `high()`, — мы не сможем использовать функцию `find_highest()` для поиска наибольшего элемента в некоторой части вектора. На самом деле, "связавшись с указателями", мы достигли практической выгоды, получив функцию, которая может работать как с векторами, так и с массивами. Помните: обобщение может привести к функциям, которые применимы для решения большего количества задач.

20.2. Принципы библиотеки STL

Стандартная библиотека языка C++, обеспечивающая основу для работы с данными, представленными в виде последовательности элементов, называется STL. Обычно эту аббревиатуру расшифровывают как "стандартная библиотека шаблонов" ("standard template library"). Библиотека STL является частью стандарта ISO C++. Она содержит контейнеры (такие, как классы `vector`, `list` или `map`) и обобщенные алгоритмы (такие, как `sort`, `find` или `accumulate`). Следовательно, мы имеем право говорить — и говорим, — что такие инструменты, как класс `vector`, являются частью как библиотеки STL, так и стандартной библиотеки. Другие средства стандартной библиотеки, такие как потоки `ostream` (см. главу 10) и функции для работы с С-строками (раздел Б.11.3), не являются частью STL. Чтобы лучше оценить и понять библиотеку STL, сначала рассмотрим проблемы, которые мы должны решать, работая с данными, а также какими должны быть идеальные решения.

Существуют два основных аспекта обработки данных: вычисления и данные. Иногда мы сосредоточиваем внимание на вычислениях и говорим об инструкциях `if`, циклах, функциях, обработке ошибок и пр. В других случаях мы фокусируемся на данных и говорим о массивах, векторах, строках, файлах и пр. Однако, для того чтобы выполнить полезную работу, мы должны учитывать оба аспекта. Большой объем данных невозможно понять без анализа, визуализации и поиска "чего-то интересного". И наоборот, мы можем выполнять вычисления так, как хотим, но такой подход оказывается слишком скучным и "стерильным", пока у нас не будет неких данных, которые свяжут наши вычисления с реальностью. Более того, вычислительная часть программы должна элегантно взаимодействовать с частью "информационной".



Говоря о данных, мы подразумеваем множество разных данных: десятки фигур, сотни значений температуры, тысячи регистрационных записей, миллионы точек, миллиарды веб-страниц и т.д.; иначе говоря, мы говорим об обработке контейнеров данных, потоков данных и т.д. В частности, мы не рассматриваем вопросы, как лучше выбрать набор данных, представляющих небольшой объект, такой как комплексное число, запись о температуре или окружность. Эти типы описаны в главах 9, 11 и 14.

Рассмотрим простые примеры, которые иллюстрируют наше понятие о большом наборе данных.

- Сортировка слов в словаре.
- Поиск номера в телефонной книге по заданному имени.
- Поиск максимальной температуры.
- Поиск всех чисел, превышающих 8800.
- Поиск первого появления числа 17.
- Сортировка телеметрических записей по номерам устройств.
- Сортировка телеметрических записей по временным меткам.
- Поиск первого значения, большего, чем строка "Petersen".
- Поиск наибольшей величины.
- Поиск первого несовпадения между двумя последовательностями.
- Вычисление попарного произведения элементов двух последовательностей.
- Поиск наибольшей температуры для каждого дня месяца.
- Поиск первых десяти лучших продавцов по записям о продажах.
- Подсчет количества появлений слова "Stroustrup" в сети веб.
- Вычисление суммы элементов.

Обратите внимание на то, что каждую из этих задач мы можем описать, не упоминая о способе хранения данных. Очевидно, что мы как-то должны работать со списками, векторами, файлами, потоками ввода и т.д., но мы не обязаны знать, как именно хранятся (и собираются) данные, чтобы говорить о том, что будем делать с ними. Важен лишь тип значений или объектов (тип элементов), способ доступа к этим значениям или объектам, а также что именно мы хотим с ними сделать.

Эти виды задач весьма распространены. Естественно, мы хотели бы написать код, который решал бы эти задачи просто и эффективно. В то же время перед нами как программистами стоят следующие проблемы.

- Существует бесконечное множество вариантов типов данных ("видов данных").
- Существует огромное количество способов хранения коллекций элементов данных.
- Существует громадное количество задач, которые мы хотели бы решить с помощью таких коллекций данных.

Для того чтобы минимизировать влияние этих проблем, мы хотели бы как можно сильнее обобщить наш код, чтобы он с одинаковым успехом мог работать с разными типами данных, разными способами их хранения и решать разные задачи, связанные с обработкой данных. Иначе говоря, мы хотим обобщить наш код так, чтобы он охватывал все возможные варианты. Мы действительно не хотим решать каждую задачу с нуля; это слишком утомительная потеря времени.

Для того чтобы понять, какая поддержка нужна для написания такого кода, рассмотрим более абстрактно, что мы можем делать с данными.

- Собирать данные в контейнерах
  - таких, как **`vector`**, **`list`** и массивы.
- Организовывать данные
  - для вывода;
  - для быстрого доступа.
- Получать данные
  - по индексу (например, найти 42-й элемент);
  - по значению (например, найти первую запись, в которой в поле **`age`** записано число 7);
  - по свойствам (например, все записи, в которых значение поля **`temperature`** больше 32 и меньше 100).
- Модифицировать контейнер
  - добавлять данные;
  - удалять данные;
  - сортировать (в соответствии с некоторым критерием).
- Выполнять простые математические операции (например, умножить все элементы на 1.7).

Мы хотели бы делать все это, не утонув в море информации, касающейся различий между контейнерами, способами доступа к элементам и их

типами. Если нам это удастся, то мы сделаем рывок по направлению к своей цели и получим эффективный метод работы с большими объемами данных.

Оглядываясь назад на методы и инструменты программирования, описанные в предыдущих главах, мы видим, что уже можем писать программы, не зависящие от типа используемых данных. Этот вывод основан на следующих фактах.

- Использование типа `int` мало отличается от использования типа `double`.
- Использование типа `vector<int>` мало отличается от использования типа `vector<string>`.
- Использование массива чисел типа `double` мало отличается от использования вектора `vector<double>`.

Мы бы хотели организовать наш код так, чтобы новый код пришлось бы писать, только если бы нам действительно нужно было сделать что-то совершенно новое и кардинально отличающееся от предыдущих задач. В частности, мы хотели бы иметь код, решающий распространенные задачи программирования, а не переписывать программы каждый раз, когда изменяется способ хранения данных или их интерпретация.

- Поиск значения в объекте класса `vector` не должен отличаться от поиска значения в массиве.
- Поиск объекта класса `string` без учета регистра не должен отличаться от поиска объекта класса `string` с учетом регистра.
- Вывод графического представления экспериментальных данных с точными значениями не должен отличаться от вывода для округленных данных.
- Копирование файла не должно отличаться от копирования вектора.

С учетом сказанного мы хотим писать код, который

- легко читать;
- легко модифицировать;
- имеет систематический характер;
- короткий;
- быстро работает.

Для того чтобы минимизировать наши усилия по программированию, мы должны решить следующие задачи.

- Единообразный доступ к данным:
  - не зависящий от способа хранения данных;
  - не зависящий от типа данных.
- Доступ к данным, безопасный с точки зрения типа.

- Легкий обход данных.
- Компактное хранение данных.
- Скорость выполнения
  - поиска данных;
  - добавления данных;
  - удаления данных.
- Стандартные версии большинства широко распространенных алгоритмов
  - таких, как **copy**, **find**, **search**, **sort**, **sum**, ...

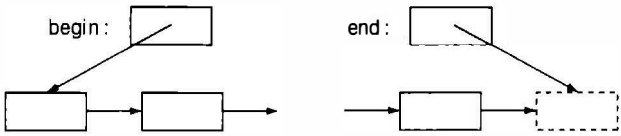
Библиотека STL обеспечивает и другие возможности. Мы изучим эту библиотеку не только потому, что она представляет собой очень полезный набор инструментов, но и потому, что она является примером максимальной гибкости и эффективности. Библиотека STL была разработана Алексом Степановым (Alex Stepanov) как база для универсальных, правильных и эффективных алгоритмов, работающих с разнообразными структурами данных. Ее идеал — простота, универсальность и элегантность математики.

Если бы в нашем распоряжении не было библиотеки с ясно выраженными идеями и принципами, то каждый программист должен был бы разрабатывать каждую программу, используя лишь фундаментальные языковые конструкции и придерживаясь идей, которые в данный момент кажутся хорошими. Для этого приходилось бы выполнять много лишней работы. Более того, в результате часто получалась бы беспринципная путаница; часто такие программы не может понять никто, кроме их авторов, и очень сомнительно, что эти программы можно использовать в каком-то ином контексте.

Теперь, рассмотрев мотивы и цели, перейдем к описанию основных определений из библиотеки STL, а затем изучим примеры их применения для более простого создания более совершенного кода обработки данных.

20.3. Последовательности и итераторы

Центральной концепцией библиотеки STL является последовательность. С точки зрения STL любая коллекция данных представляет собой последовательность. Последовательность имеет начало и конец. Мы можем перемещаться по последовательности от начала к концу, при необходимости считывая или записывая значение элементов. Начало и конец последовательности идентифицируются парой итераторов. *Итератор* (iterator) — это объект, идентифицирующий элемент последовательности. Последовательность можно представить себе следующим образом.



Здесь **begin** и **end** — итераторы; они идентифицируют начало и конец последовательности. Последовательность STL является тем, что называют "полуоткрытой" ("half-open") последовательностью; иначе говоря, элемент, идентифицированный итератором **begin**, является частью последовательности, а итератор **end** указывает на ячейку, следующую за концом последовательности. Обычно такие последовательности (диапазоны) обозначаются следующим образом: [**begin**,**end**). Стрелки, направленные от одного элемента к другому, означают, что если у нас есть итератор, указывающий на один элемент, то мы можем получить итератор, указывающий на элемент, следующий за ним.

Что такое итератор? Это довольно абстрактное понятие.

- Итератор указывает (ссылается) на элемент последовательности (или за последний элемент).
- Два итератора можно сравнивать с помощью операторов == и **!=**.
- Значение элемента, на который указывает итератор, можно получить с помощью унарного оператора * ("разыменование").
- Итератор на следующий элемент можно получить с помощью оператора ++.

Допустим, что **p** и **q** — итераторы, указывающие на элементы одной и той же последовательности.

Основные операции над стандартными итераторами	
p==q	Равно **true** тогда и только тогда, когда оба итератора, **p** и **q**, указывают на один и тот же элемент или оба указывают за последний элемент
p!=q	**! (p==q)**
***p**	Ссылка на элемент, на который указывает итератор **p**
***p=val**	Присваивает значение **val** элементу, на который указывает итератор **p**
val=*p	Присваивает переменной **val** значение элемента, на который указывает итератор **p**
++p	Устанавливает итератор **p** указывающим на следующий элемент последовательности или на элемент, следующий за последним элементом последовательности

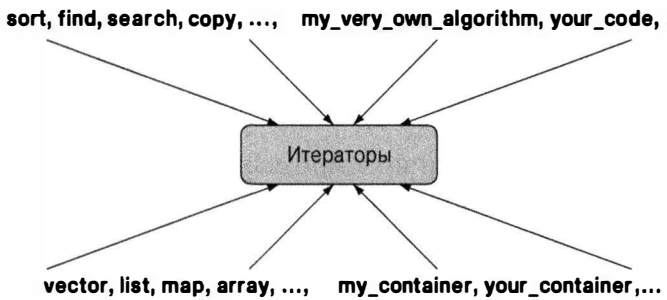
Очевидно, что идея итератора тесно связана с идеей указателя (см. раздел 17.4). Фактически указатель на элемент массива является итератором. Однако многие итераторы являются не просто указателями; например, мы могли бы определить итератор с проверкой выхода за пределы допустимого диапазона, который генерирует исключение при попытке указать за

пределы последовательности [`begin`,`end`) или разыменовать итератор `end`. Оказывается, итератор обеспечивает огромную гибкость и универсальность именно как абстрактное понятие, а не как конкретный тип. В этой и следующей главах вы увидите несколько примеров работы с итераторами.

ПОПРОБУЙТЕ

Напишите функцию `void copy(int*f1, int*e1, int*f2)`, копирующую элементы массива типа `int`, определенного последовательностью [`f1`,`e1`) в другую последовательность [`f2`,`f2+(e1-f1)`). Используйте только упомянутые выше операции с итераторами (а не индексирование).

Итераторы используются в качестве средства связи между нашим кодом (алгоритмами) и нашими данными. Автор кода знает о существовании итераторов (но не знает, как именно они обращаются к данным), а поставщик данных предоставляет итераторы, не раскрывая пользователям детали механизма хранения данных. В результате получаем достаточно независимые друг от друга алгоритмы и контейнеры. Процитируем Алекса Степанова: "Алгоритмы и контейнеры STL потому так хорошо работают друг с другом, что ничего не знают друг о друге". Вместо этого и алгоритмы, и контейнеры знают о последовательностях, определенных парами итераторов.



Иначе говоря, мои алгоритмы больше не обязаны ничего знать о разнообразных способах хранения данных и обеспечения доступа к ним; им достаточно просто знать об итераторах. И наоборот, как поставщик данных я больше не обязан писать код для обслуживания огромного количества разнообразных пользователей; мне достаточно реализовать итератор для данных. На самом фундаментальном уровне итератор определяется только операторами `*`, `++`, `==` и `!=`. Это обеспечивает его простоту и быстродействие.

Библиотека STL содержит около десяти контейнеров и шестидесяти алгоритмов, связанных с итераторами (см. главу 21). Кроме того, многие организации и отдельные лица создают контейнеры и алгоритмы в стиле STL. Вероятно, STL в настоящее время является наиболее широко известным и широко используемым примером обобщенного программирования

(см. раздел 19.3.2). Если вы знаете основные концепции и несколько примеров, то сможете использовать и все остальное.

20.3.1. Вернемся к примерам

Посмотрим, как можно решить задачу "найти максимальный элемент" с помощью последовательности STL.

```
template<typename Iterator>
Iterator high(Iterator first, Iterator last)
// Возвращает итератор, указывающий максимальный
// элемент диапазона [first,last)
{
    Iterator high = first;
    for(Iterator p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

Обратите внимание, что мы исключили локальную переменную `h`, которую до сих пор использовали для хранения максимального значения. Если вам неизвестен реальный тип элементов последовательности, то инициализация значением `-1` выглядит совершенно произвольной и странной. И она действительно является произвольной и странной! Кроме того, такая инициализация представляет собой ошибку: в нашем примере число `-1` оправдывает себя только потому, что отрицательных скоростей не бывает. Мы знаем, что "магические константы", такие как `-1`, препятствуют сопровождению кода (см. разделы 4.3.1, 7.6.1, 10.11.1 и др.). Здесь мы видим, что такие константы могут ограничить применимость функции и свидетельствовать о неполноте решения; иначе говоря, "магические константы" могут быть — и часто бывают! — свидетельством небрежности.

Обобщенную функцию `high()` можно использовать для любых типов элементов, которые можно сравнивать с помощью операции `<`. Например, мы могли бы использовать функцию `high()` для поиска лексикографически последней строки в контейнере `vector<string>` (см. упр. 7).

Шаблонную функцию `high()` можно применять к любой последовательности, определенной парой итераторов. Например, мы можем в точности повторить нашу демонстрационную программу, приведенную в начале главы.

```
double* get_from_jack(int* count); // Джек записывает числа
                                   // типа double в массив,
                                   // возвращая их количество
                                   // в *count
vector<double>* get_from_jill();   // Джилл заполняет вектор
```

```
void fct()
{
    int jack_count = 0;
    double* jack_data = get_from_jack(&jack_count);
    vector<double>* jill_data = get_from_jill();

    double* jack_high = high(jack_data,jack_data+jack_count);
    vector<double>& v = *jill_data;
    double* jill_high = high(&v[0],&v[0]+v.size());

    cout << "Максимум Джилл: " << *jill_high
         << "; максимум Джека: " << *jack_high;

    // ...
    delete[] jack_data;
    delete jill_data;
}
```

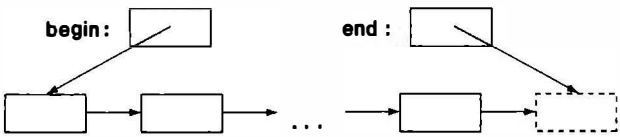
Здесь в двух вызовах функции `high()` типом аргумента шаблона `Iterator` является тип `double*`. Код ничем не отличается от нашего предыдущего решения. Точнее, выполнение кодов этих программ ничем не отличаются друг от друга, хотя степень общности этих кодов разнится существенно. Шаблонная версия функции `high()` может применяться к последовательности любого вида, лишь бы она могла быть определена парой итераторов. Прежде чем углубляться в принципы библиотеки STL и полезные стандартные алгоритмы, реализующие эти принципы, и для того чтобы избежать создания сложных кодов, рассмотрим несколько способов хранения коллекций элементов данных.

ПОПРОБУЙТЕ

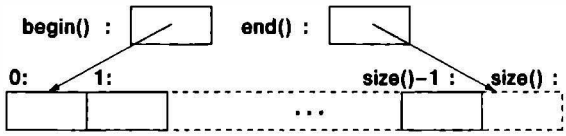
В этой программе снова сделана серьезная ошибка. Найдите ее, исправьте и предложите универсальный способ устранения таких проблем.

20.4. Связанные списки

Еще раз рассмотрим графическое представление понятия последовательности.



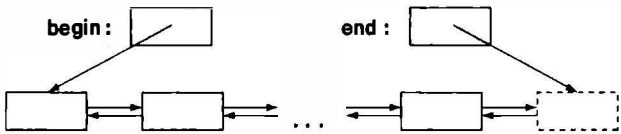
Сравним его с визуализацией вектора, хранящегося в памяти.



По существу, индекс `0` означает тот же элемент, что и итератор `v.begin()`, а индекс `v.size()` идентифицирует элемент, следующий за последним, на который можно также указать с помощью итератора `v.end()`.

Элементы в векторе располагаются в памяти последовательно. Понятие последовательности в STL такого размещения не требует. Более того, имеется много алгоритмов, в которых нам хотелось бы вставлять новые элементы между существующими без перемещения последних. Графическое представление абстрактного понятия последовательности предполагает возможность вставки (и удаления) элементов без перемещения остальных элементов. Понятие итераторов в STL поддерживает эту концепцию.

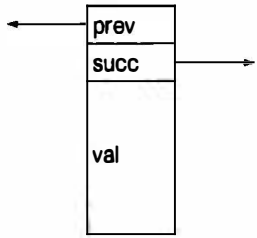
Структуру данных, которая точнее всех соответствует диаграмме последовательности в STL, называют *связанным списком* (linked list). Стрелки в абстрактной модели обычно реализуются с помощью указателей. Элемент связанного списка — это часть узла, состоящего из собственно элемента и одного или нескольких указателей. Связанный список, в котором узел содержит только один указатель (на следующий узел), называют *односвязным* (singly-linked list), а список, в котором узел указывает как на предыдущий, так и на следующий за ним узлы, — *двусвязным списком* (doubly-linked list). Мы схематично рассмотрим реализацию двухсвязных списков, которые стандартная библиотека C++ предоставляет под именем `list`. Графически такой список можно изобразить следующим образом.



В виде кода его можно представить следующим образом:

```
template<typename Elem> struct Link {
    Link* prev; // Предыдущий узел
    Link* succ; // Следующий узел
    Elem val;   // Значение
};

template<typename Elem> struct list {
    Link<Elem>* first;
    Link<Elem>* last; // Узел, находящийся за последним узлом
};
```

Графически схема класса `Link` приведена ниже.



Существует много способов реализации и представления связанных списков. Описание списка, реализованного в стандартной библиотеке, приведено в приложении Б. Здесь мы лишь кратко перечислим основные свойства списка — возможность вставлять и удалять элементы, не затрагивая при этом существующие элементы, а также покажем, как перемещаться по списку с помощью итератора, и приведем пример использования списка.

Мы настоятельно рекомендуем вам, размышляя о списках, рисовать диаграммы, иллюстрирующие рассматриваемые вами операции. Манипуляции связанным списком — это тема, в которой один рисунок может заменить тысячу слов.

20.4.1. Операции над списками

Какие операции необходимы для списка?

- Операции, которые имеются у вектора (конструкторы, определение размера и т.д.), за исключением индексирования.
- Вставка (добавление) элемента в список и удаление элемента из него.
- Нечто, что можно использовать для ссылки на элементы и обхода списка: итератор.

В STL тип итератора является членом класса контейнера, поэтому и мы поступим так же.

```
template<typename Elem> class list {
// Детали представления и реализации
public:
    class iterator;   // Тип — член класса: iterator

    iterator begin(); // Итератор, указывающий на первый элемент
    iterator end();   // Итератор, указывающий на элемент,
                      // следующий за последним

    iterator insert(iterator p,      // Вставка v в список
                    const Elem& v); // после элемента, на который
                                    // указывает итератор p
    iterator erase(iterator p);      // Удаление из списка элемента,
                                    // на который указывает итератор p
```

```
    void push_back(const Elem& v);  // Вставка v в конец списка
    void push_front(const Elem& v); // Вставка v в начало списка
    void pop_front();               // Удаление первого элемента
    void pop_back();                // Удаление последнего элемента

    Elem& front();                  // Первый элемент
    Elem& back();                   // Последний элемент
    // ...
};
```

Так же, как наш класс **vector** не совпадал с полной версией стандартного вектора, так и класс **list** — это далеко не полное определение стандартного библиотечного списка. В этом определении все правильно; оно просто неполное. Цель "нашего" класса **list** — объяснить устройство связанных списков, продемонстрировать их реализацию и показать способы использования их основных возможностей. Более подробная информация имеется в приложении Б и в книгах о языке C++, предназначенных для экспертов.

Итератор играет главную роль в определении класса **list** в библиотеке STL. Итераторы используются для идентификации места вставки или удаления элементов. Кроме того, их используют для "навигации" по списку вместо оператора индексирования. Такое применение итераторов очень похоже на использование указателей при обходе массивов и векторов, описанное в разделах 20.1 и 20.3.1. Этот стиль итераторов является ключом к стандартным алгоритмам (разделы 21.1–21.3).

Почему в классе **list** не используется индексирование? Мы могли бы проиндексировать узлы, но эта операция удивительно медленная: для того чтобы достичь элемента **lst[1000]**, нам пришлось бы начинать с первого элемента и пройти все элементы по очереди, пока мы не достигли бы элемента с номером **1000**. Если вы хотите этого, то можете реализовать эту операцию сами (или применить алгоритм **advance()**; см. раздел 20.6.2). По этой причине стандартный класс **list** не предоставляет пользователям операцию индексирования, выглядящую столь невинно.

Мы сделали тип итератора списка членом класса (вложенным классом), потому что нет никаких причин делать его глобальным. Он используется только со списками. Кроме того, это позволяет нам называть каждый тип итератора в контейнере одним и тем же именем **iterator**. В стандартной библиотеке есть **list<T>::iterator**, **vector<T>::iterator**, **map<K,V>::iterator** и т.д.

20.4.2. Итерация

Итератор списка должен обеспечивать выполнение операций *****, **++**, **==** и **!=**. Поскольку стандартный список является двухсвязным, в нем также есть операция **--** для перемещения назад, к началу списка.

```
template<typename Elem> class list<Elem>::iterator {
    Link<Elem>* curr; // Текущий узел
public:
    iterator(Link* p) :curr(p) {}

    // Вперед:
    iterator& operator++() {curr = curr->succ; return *this; }

    // Назад:
    iterator& operator--() { curr = curr->prev; return *this; }

    // Разыменование (получение значения):
    Elem& operator*() { return curr->val; }

    bool operator==(const iterator& b) const
        { return curr == b.curr; }
    bool operator!= (const iterator& b) const
        { return curr != b.curr; }
};
```

Эти функции короткие, простые и, очевидно, эффективные: в них нет циклов, сложных выражений и "подозрительных" вызовов других функций. Если эта реализация вам не понятна, то посмотрите на диаграммы, приведенные ранее. Этот итератор списка представляет собой просто указатель на узел с необходимыми операциями. Несмотря на то что реализация (код) класса `list<Elem>::iterator` сильно отличается от обычного указателя, который использовался в качестве итератора для векторов и массивов, семантика их операций одинакова. По существу, итератор списка обеспечивает подходящие операции `++`, `--`, `*`, `==` и `!=` для указателя на узел.

Посмотрим на функцию `high()` еще раз.

```
template<typename Iter> // Требует Input_iterator<Iter>()
                        // (см. раздел 19.3.3)
Iter high(Iter first, Iter last)
// Возвращает итератор, указывающий на элемент из
// диапазона [first,last), имеющий наибольшее значение
{
    Iter high = first;
    for (Iter p = first; p!=last; ++p)
        if (*high<*p) high = p;
    return high;
}
```

Мы можем использовать ее с объектом класса `list`.

```
void f()
{
    list<int> lst; for (int x; cin >> x; ) lst.push_front(x);

    list<int>::iterator p = high(lst.begin(), lst.end());
    cout << "Наибольшее значение равно " << *p << '\n';
}
```

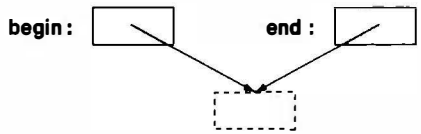
Здесь "значением" аргумента `Iter` является объект типа `list<int>::iterator`, а реализация операций `++`, `*` и `!=` совершенно иная, чем в случае массива, хотя смысл их остается неизменным. Шаблонная функция `high()` по-прежнему обходит данные (в данном случае список) и находит среди них максимальное значение. Мы можем вставлять элементы в любое место списка, так что мы использовали функцию `push_front()` для добавления элементов в начало списка просто для иллюстрации своих возможностей. С таким же успехом мы могли бы использовать и функцию `push_back()`, как делали это для объектов класса `vector`.

ПОПРОБУЙТЕ

В классе `vector` стандартной библиотеки нет функции `push_front()`. Почему? Реализуйте функцию `push_front()` для класса `vector` и сравните ее с функцией `push_back()`.

Наконец пришло время задаться вопросом "А что если список окажется пустым?" Иначе говоря, "Что если `lst.begin()==lst.end()`?" В таком случае выполнение инструкции `*p` будет попыткой разыменования (несуществующего) элемента, следующего за последним, т.е. `lst.end()`. Это катастрофа! Или, что еще хуже, результатом может быть случайная величина, которая приведет к неверному ответу.

Последняя формулировка вопроса содержит явную подсказку: мы можем проверить, пуст ли список, сравнив итераторы `begin()` и `end()`, — фактически последовательность пуста, когда ее начало совпадает с ее концом.



Это и есть самая главная причина, по которой итератор `end()` указывает на элемент, следующий за последним, а не на сам последний элемент: при этом пустая последовательность не является особым случаем. Мы не любим особые случаи, потому что — по определению — для каждого из них приходится писать особый код.

В нашем примере можно поступить следующим образом:

```
list<int>::iterator p = high(lst.begin(), lst.end());
if (p == lst.end())  // Мы достигли конца последовательности?
    cout << "Список пуст";
else
    cout << "Наибольшее значение равно " << *p << endl;
```

Работая с алгоритмами из библиотеки STL, мы систематически используем такую проверку на равенство итератора значению `end()`.

Поскольку стандартная библиотека предоставляет свой класс списка, мы не будем углубляться здесь в детали реализации. Вместо этого кратко рассмотрим, для чего удобно их применять (если вас интересуют детали реализации списков, выполните упр. 12–14).

20.5. Очередное обобщение класса `vector`

Очевидно, из примеров, приведенных в разделах 20.3 и 20.4, следует, что стандартный **vector** (как и класс **std::list**) имеет тип-член класса **iterator**, а также функции-члены **begin()** и **end()**. Однако мы не предоставили их в нашем классе **vector** в главе 19. Что в действительности делает разные контейнеры более или менее взаимозаменяемыми в STL-стиле обобщенного программирования, представленном в разделе 20.3? Сначала набросаем схему решения (игнорируя для простоты распределители памяти), а затем объясним ее.

```
template<typename T>  // Требует Element<T>() (раздел 19.3.3)
class vector {
public:
    using size_type = unsigned long;
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    size_type size();

    // ...
};
```

Объявление **using** создает синоним типа; иначе говоря, для нашего класса **vector** имя **iterator** — это синоним, т.е. другое имя типа, который мы решили использовать в качестве итератора: **T***. Теперь для объекта **v** класса **vector** можно написать следующие инструкции:

```
vector<int>::iterator p = find(v.begin(), v.end(),32);
```

и

```
for (vector<int>::size_type i = 0; i<v.size(); ++i)
    cout << v[i] << '\n';
```

Суть в том, что для того, чтобы написать эти инструкции, нам на самом деле не обязательно знать, какие именно типы имеют имена **iterator** и

size_type. В частности, в приведенном выше коде, выраженном через типы **iterator** и **size_type**, мы будем работать с векторами, в которых тип **size_type** — это не **unsigned long** (как во многих процессорах встроенных систем), а тип **iterator** — не простой указатель, а класс (как во многих широко известных реализациях языка C++).

В стандарте класс **list** и другие стандартные контейнеры определены аналогично, например так:

```
template<typename T>  // Требует Element<T>() (раздел 19.3.3)
class list {
public:
    class Link;
    using size_type  = unsigned long;
    using value_type = T;
    class iterator;       // См. раздел 20.4.2
    class const_iterator; // Аналог iterator, но не допускает
                          // изменения элементов

    // ...

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    size_type size();

    // ...
};
```

При этом можно писать код, не беспокоясь о том, что он использует: класс **list** или **vector**. Все стандартные алгоритмы определены в терминах таких имен типов, как **iterator** и **size_type**, поэтому они не должны обязательно зависеть от реализации контейнеров или от того, с контейнером какого вида приходится работать (подробнее об этом — в главе 21).

В качестве альтернативы записи **C::iterator** для некоторого контейнера **C** мы зачастую предпочитаем запись **Iterator<C>**. Этого можно достичь с помощью простого псевдонима шаблона:

```
template<typename C>
using Iterator = typename C::iterator; // Iterator<C> означает
                                       // typename C::iterator
```

Причиной нашего предпочтения записи **Iterator<C>** служит тот факт, что по техническим причинам нам приходится предварять запись **C::iterator** префиксом **typename**, чтобы указать, что **iterator** является типом. Аналогично мы определяем

```
template<typename C>
using Value_type = typename C::value_type;
```

Теперь мы можем писать `Value_type<C>`. Эти псевдонимы типов отсутствуют в стандартной библиотеке, но вы можете найти их в заголовочном файле `std_lib_facilities.h`.

Объявление `using` является обозначением стандарта C++11 и обобщением того, что в C и C++ известно как конструкция `typedef` (раздел A.16).

20.5.1. Обход контейнера

Используя функцию `size()`, мы можем проходить по нашим векторам от первого элемента до последнего:

```
void print1(const vector<double>& v)
{
    for (int i = 0; i<v.size(); ++i)
    cout << v[i] << '\n';
}
```

Но этот способ не работает со списком, поскольку класс `list` не предоставляет операции индексации. Однако и вектор, и список из стандартной библиотеки можно обойти, если воспользоваться простым циклом `for` для диапазона (раздел 4.6.1):

```
void print2(const vector<double>& v, const list<double>& lst)
{
    for (double x : v)
    cout << x << '\n';
    for (double x : lst)
    cout << x << '\n';
}
```

Этот способ работает как для обоих контейнеров стандартной библиотеки, так и для "наших" `vector` и `list`. Как? Фокус в том, что цикл `for` для диапазона определен в терминах функций `begin()` и `end()`, возвращающих итераторы на первый и следующий за последним элементы нашего вектора `vector`. Цикл `for` для диапазона — просто синтаксическое сокращение для цикла по всей последовательности с использованием итераторов. Когда мы определяем `begin()` и `end()` для наших классов `vector` и `list`, мы "случайно" предоставляем все необходимое для работы цикла `for` для диапазона.

20.5.2 Ключевое слово `auto`

Когда мы должны написать цикл для прохода по обобщенной структуре, имена итераторов могут стать серьезной неприятностью. Судите сами:

```
template<typename T>    // Требует Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
        cout << *p << '\n';
```

```
    list<T>::iterator q = find(lst.begin(), lst.end(),T{42});
}
```

Наибольшее раздражение вызывает тот факт, что компилятору, очевидно, известны как тип `iterator` для класса `list`, так и тип `size_type` для класса `vector`. Почему мы должны говорить компилятору то, что ему и так хорошо известно? Это раздражает тех, кто не любит работать машинисткой, и открывает широкие возможности для ошибок. К счастью, теперь мы и не обязаны это делать: можно объявить переменную с помощью ключевого слова `auto`, означающего, что ее типом будет тип соответствующего итератора:

```
template<typename T>  // Требует Element<T>()
void user(vector<T>& v, list<T>& lst)
{
    for (auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << '\n';
    auto q = find(lst.begin(), lst.end(),T{42});
}
```

Здесь `p` представляет собой `vector<T>::iterator`, а `q` — `list<T>::iterator`. Мы можем использовать ключевое слово `auto` почти в любом определении, включающем инициализатор:

```
auto x = 123; // x имеет тип int
auto c = 'y'; // c имеет тип char
auto& r = x;  // r имеет тип int&
auto y = r;   // Тип y – int (ссылка неявно разыменовывается)
```

Обратите внимание, что строковый литерал имеет тип `const char*`, так что применение `auto` к строковым литералам может приводить к неприятным сюрпризам:

```
auto s1   = "San Antonio";     // s1 имеет тип const char*
                               // (Сюрприз?!)
string s2 = "Fredericksburg"; // s2 имеет тип string
```

Когда мы точно знаем, какой тип нам нужен, его часто можно написать так же легко, как и ключевое слово `auto`.

Одним из наиболее распространенных применений ключевого слова `auto` является применение его для определения типа переменной цикла `for` для диапазона:

```
template<typename C>   // Требует Container<T>
void print3(const C& cont)
{
    for (const auto& x : cont) cout << x << '\n';
}
```

Здесь мы использовали `auto`, поскольку именовать тип элемента контейнера `cont` не так-то просто. Мы использовали ключевое слово `const`,

поскольку мы не записываем элементы контейнера, и использовали `&` (ссылку) на тот случай, если элементы настолько велики, что их копирования желательно избежать.

20.6. Пример: простой текстовый редактор

Существенным свойством списка является возможность вставлять и удалять элементы без перемещения других элементов списка. Рассмотрим простой пример, иллюстрирующий этот факт. Посмотрим, как представить символы в текстовом документе в простом текстовом редакторе. Это представление должно быть таким, чтобы операции над документом стали простыми и по возможности эффективными.

Какие именно операции? Предположим, что документ будет храниться в основной памяти компьютера. Следовательно, можно выбрать любое удобное представление и просто превратить его в поток байтов, когда нам потребуется сохранить его в файле. Аналогично мы можем читать поток байтов из файла и превращать их в соответствующее представление в памяти компьютера. Решив этот вопрос, мы можем сконцентрироваться на выборе подходящего представления документа в памяти компьютера. По сути, это представление должно хорошо поддерживать пять операций.

- Создание документа из входного потока байтов.
- Вставка одного или нескольких символов.
- Удаление одного или нескольких символов.
- Поиск строки.
- Генерация потока байтов для вывода в файл или на экран.

В качестве простейшего представления можно выбрать класс **`vector<char>`**. Однако, чтобы добавить или удалить символ в векторе, нам пришлось бы перемещать все последующие символы документа. Рассмотрим пример.

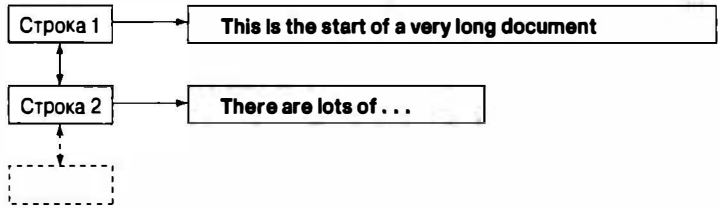
```
This is he start of a very long document.
There are lots of ...
```

Мы могли бы добавить недостающий символ `t` и получить следующий текст:

```
This is the start of a very long document.
There are lots of ...
```

Однако, если бы эти символы хранились в отдельном объекте класса **`vector<char>`**, мы должны были бы переместить все символы, начиная с буквы `h`, на одну позицию вправо. Для этого пришлось бы копировать много символов. Так, для документа, состоящего из 80 тыс. символов (примерный размер этой главы), при вставке или удалении символа в среднем нам пришлось бы переместить 40 тыс. символов. В результате временная

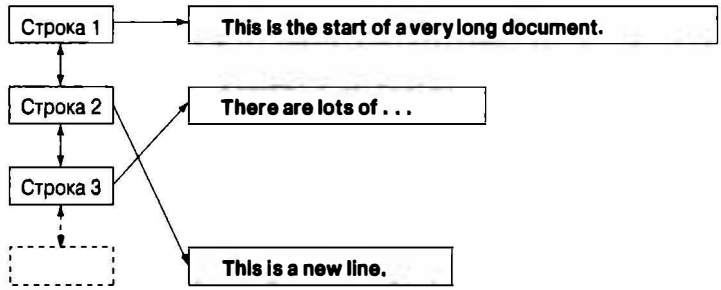
задержка стала бы заметной и досадной для пользователей. Вследствие этого мы решили разбить наше представление на части и изменять части документа так, чтобы не перемещать большие массивы символов. Мы представим документ в виде списка строк с помощью класса `list<Line>`, где шаблонный параметр `Line` — это класс `vector<char>`.



Теперь для вставки символа `t` достаточно переместить только остальные символы из этой строки. Более того, при необходимости можно добавить новую строку без перемещения каких-либо символов. В качестве примера рассмотрим вставку строки "`This is a new line.`" после слова "`document.`".

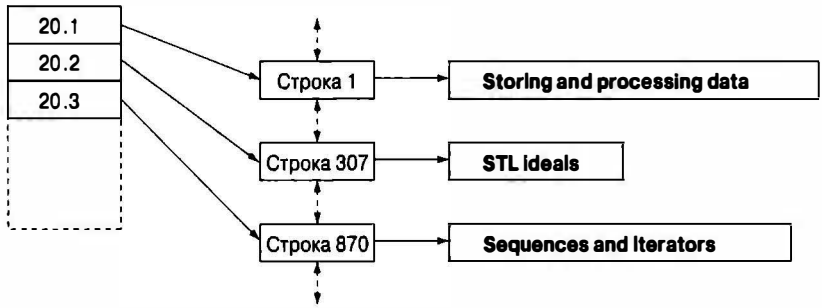
```
This is the start of a very long document.
This is a new line.
There are lots of ...
```

Все, что нам для этого нужно, — добавить новую строку между двумя другими.



Важность вставки новых узлов в список без перемещения существующих узлов объясняется тем, что у нас могут быть итераторы, указывающие на узлы, или указатели, которые указывают на объекты в этих узлах. Вставка и удаление строк на такие итераторы и указатели не влияет. Например, в текстовом процессоре может использоваться объект класса `vector<list<Line>::iterator>`, в котором хранятся итераторы, установленные на начало каждого заголовка и подзаголовка текущего документа.
Мы можем добавить строки в раздел 20.2, не нарушая целостности итератора, указывающего на раздел 20.3.

В заключение отметим, что использование списка строк вместо вектора строк или вектора всех символов имеет как логические, так и практические причины. Однако следует подчеркнуть, что ситуации, когда эти при-



чины становятся важными, являются весьма редкими, так что правило "по умолчанию используйте класс **vector**" по-прежнему действует. Нужна особая причина, чтобы предпочесть класс `list` классу **vector**, даже если вы мысленно представляете свои данные только в виде списка! (См. раздел 20.7.) Список — это логическая концепция, которую в вашей программе можно представить с помощью как (связанного) списка, так и вектора. В библиотеке STL ближайшим аналогом нашего повседневного представления о списке (например, список дел, товаров или расписание) является последовательность, а большинство последовательностей лучше всего представлять с помощью класса **vector**.

20.6.1. Строки

Как решить, что такое строка в нашем документе? Есть три очевидные альтернативы.

1. Положиться на индикатор новой строки (например, символ `'\n'`) в пользовательском вводе.
2. Провести тем или иным образом анализ документа и использовать "естественную" пунктуацию (например, точки).
3. Разделять любую строку, длина которой превышает некий заданный порог (например, 50 символов), на две.

Кроме этого, несомненно, существуют менее очевидные варианты. Для простоты выберем первую альтернативу.

Представим документ в нашем редакторе в виде объекта класса **Document**. Схематически наш тип должен выглядеть примерно так:

```
using Line = vector<char>; // Строка – это вектор символов

struct Document {
     list<Line> line;       // Документ – это список строк
     Document() { line.push_back(Line{}); }
};
```

Каждый объект класса **Document** начинается с пустой строки: конструктор класса **Document** создает пустую строку и вносит ее в список строк.

Чтение и разделение на строки можно выполнить следующим образом:

```
istream& operator>>(istream& is, Document& d)
{
    for (char ch; is.get(ch); ) {
        d.line.back().push_back(ch);  // Добавление символа
        if (ch=='\n')
            d.line.push_back(Line{}); // Добавление строки
    }
    if (d.line.back().size())         // Добавление последней
        d.line.push_back(Line{});     // пустой строки
    return is;
}
```

Классы **vector** и **list** имеют функцию-член **back()**, возвращающую ссылку на последний элемент. Для ее использования вы должны быть уверены, что она действительно указывает на последний элемент, — функцию **back()** нельзя применять к пустому контейнеру! Вот почему в соответствии с определением каждый объект класса **Document** должен завершаться пустой строкой. Обратите внимание на то, что мы храним каждый введенный символ, даже символы перехода на новую строку (**'\n'**). Хранение символов перехода на новую строку существенно упрощает вывод, но при подсчете символов вам следует быть осторожным (простой подсчет символов будет учитывать пробелы и символы перехода на новую строку).

20.6.2. Итерация

Если бы документ хранился как объект класса **vector<char>**, перемещаться по нему было бы просто. Как перемещать итератор по списку строк? Очевидно, что перемещаться по списку можно с помощью **list<Line>::iterator**. Однако, что если мы хотим пройтись по символам один за другим, не думая о разбиении на строки? Мы могли бы использовать итератор, специально разработанный для нашего класса **Document**.

```
class Text_iterator {     // Отслеживает строку и
                          // позицию символа в строке
    list<Line>::iterator ln;
    Line::iterator pos;
public:
    // Устанавливает итератор на позицию pp в ll-й строке
    Text_iterator(list<Line>::iterator ll, Line::iterator pp)
        :ln{ll}, pos{pp} { }

    char& operator*() { return *pos; }
    Text_iterator& operator++();

    bool operator==(const Text_iterator& other) const
```

```
        { return ln==other.ln && pos==other.pos; }

    bool operator!=(const Text_iterator& other) const
        { return !(*this==other); }
};

Text_iterator& Text_iterator::operator++()
{
    ++pos;                      // Переход к новому символу
    if (pos==(*ln).end()) {
        ++ln;                   // Переход на новую строку
        pos = (*ln).begin();
    }
    return *this;
}
```

Для того чтобы класс **Text_iterator** стал полезным, необходимо снабдить класс **Document** традиционными функциями **begin()** и **end()**.

```
struct Document {
    list<Line> line;

    Text_iterator begin()  // Первый символ первой строки
        { return Text_iterator(line.begin(),
                 (*line.begin()).begin()); }
     Text_iterator end()   // За последним символом документа
         { return(line.end(), (*line.end()).end());}
};
```

Мы использовали забавную конструкцию **(*line.begin()).begin()**, потому что хотим начинать перемещение итератора с начала строки, на которую указывает итератор **line.begin()**; в качестве альтернативы можно было бы использовать вызов **line.begin()->begin()**, так как стандартные итераторы поддерживают операцию **->**.

Теперь мы можем проходить по символам документа:

```
void print(Document& d)
{
    for(auto p : d) cout << *p;
}

print(my_doc);
```

Представление документа в виде последовательности символов полезно по многим причинам, но обычно мы перемещаемся по документам в поисках некоторой более конкретной информации, чем символ. Например, рассмотрим фрагмент кода, удаляющий строку **n**.

```
void erase_line(Document& d, int n)
{
    if (n<0 || d.line.size()-1<=n) return;
    auto p = d.line.begin();
```

```
    advance(p,n);
    d.line.erase(p);
}
```

Вызов `advance(p,n)` перемещает итератор `p` на `n` элементов вперед; функция `advance()` — это функция стандартной библиотеки, но мы можем сами написать подобный код.

```
template<typename Iter> // Требует Forward_iterator<Iter>
void advance(Iter& p, int n)
{
    while (0<n) { ++p; --n; }
}
```

Обратите внимание, что функцию `advance()` можно использовать для имитации индексирования. Фактически для объекта класса `vector` с именем `v` выражение `*advance(v.begin(),n)` почти эквивалентно конструкции `v[n]`. Здесь слово "почти" означает, что функция `advance()` старательно проходит по каждому из первых `n-1` элементов шаг за шагом, в то время как операция индексирования сразу обращается к `n`-му элементу. Для класса `list` мы вынуждены использовать этот неэффективный метод. Это цена, которую мы должны заплатить за гибкость списка.

Если итератор может перемещаться вперед и назад, как, например, в классе `list`, то отрицательный аргумент стандартной библиотечной функции `advance()` означает перемещение назад. Если итератор допускает индексирование, как, например, в классе `vector`, стандартная библиотечная функция `advance()` сразу направит его на требуемый элемент и не будет медленно перемещаться по всем элементам с помощью оператора `++`.

Очевидно, что стандартная функция `advance()` немного интеллектуальнее нашей. Это стоит запомнить: как правило, стандартные средства создаются более тщательно, и на них затрачивается больше времени, чем мы могли бы затратить на самостоятельную разработку, поэтому лучше использовать стандартные, а не кустарные инструменты.

ПОПРОБУЙТЕ

Перепишите нашу функцию `advance()` так, чтобы, получив отрицательный аргумент, она выполняла перемещение назад.

Вероятно, поиск — наиболее очевидный для пользователя вид итерации. Мы ищем отдельные слова (например, `milkshake` или `Gavin`), последовательности букв, которые нельзя рассматривать как слово (например, `secret\nhomestead`, т.е. строка, заканчивающаяся словом `secret`, за которым следует строка, начинающаяся словом `homestead`), регулярные выражения (например, `[bB]\w*ne`, т.е. буква `B` в верхнем или нижнем регистре, за которой следует 0 или больше букв, за которыми следуют буквы `ne`; см.

главу 23) и т.п. Покажем, как решить вторую задачу: найдем строку, используя нашу схему хранения объекта класса `Document`. Будем использовать простой — не оптимальный! — алгоритм.

- Найдем первый символ искомой строки в документе.
- Проверим, совпадают ли эти и следующие символы с символами искомой строки.
- Если совпадают, то задача решена; если нет, будем искать следующее появление первого символа.

Для общности примем соглашения STL по определению текста, в котором выполняется поиск, в виде пары итераторов. Это позволит нам применить функцию поиска не только ко всему документу, но и к любой его части. Если мы найдем нашу строку в документе, то вернем итератор, указывающий на ее первый символ; если не найдем, то вернем итератор, указывающий на конец последовательности.

```
Text_iterator find_txt(Text_iterator first,
                       Text_iterator last, const string& s)
{
    if (s.size()==0) return last; // Нельзя искать пустую строку
    char first_char = s[0];
    while (true) {
        auto p = find(first,last,first_char);
        if (p==last || match(p,last,s)) return p;
        first = ++p;          // Ищем следующее вхождение символа
    }
}
```

Возврат конца строки в качестве признака неудачного поиска является важным соглашением, принятым в библиотеке STL. Функция `match()` тривиальна; она просто сравнивает две последовательности символов. Попробуйте написать ее самостоятельно. Функция `find()`, используемая для поиска символа в последовательности, вероятно, является простейшим стандартным алгоритмом (раздел 21.2). Нашу функцию `find_txt()` можно использовать следующим образом:

```
auto p = find_txt(my_doc.begin(), my_doc.end(),
                  "secret\nhomestead");
if (p==my_doc.end())
    cout << "not found";
else {
    // Некоторые действия
}
```

Наш "текстовый редактор" и его операции очень просты. Очевидно, что мы хотим создать простой и достаточно эффективный, а не "навороченный" редактор. Однако не следует ошибочно думать, что *эффективные* вставка, удаление и поиск произвольного символа — тривиальные задачи.

Мы выбрали этот пример для того, чтобы продемонстрировать мощь и универсальность концепций последовательности, итератора и контейнера STL (таких, как `list` и `vector`) в сочетании с правилами программирования (приемами), принятыми в STL, согласно которым возврат итератора, установленного на конец последовательности, является признаком неудачи. Обратите внимание, что если бы мы захотели, то могли бы превратить класс `Document` в контейнер STL; снабдив его итератором `Text_iterator`, мы сделали ключевую часть работы по представлению `Document` в виде последовательности значений.

20.7. Классы `vector`, `list` и `string`

Почему для хранения строк мы используем класс `list`, а для символов — класс `vector`? Точнее, почему для хранения последовательности строк мы используем класс `list`, а для хранения последовательности символов — класс `vector`? Более того, почему для хранения строки мы не используем класс `string`?

Сформулируем немного более общий вариант этого вопроса. Для хранения последовательности символов у нас есть четыре способа.

- `char[]` (массив символов)
- `vector<char>`
- `string`
- `list<char>`

Какой из этих вариантов выбрать для решения конкретной задачи? Для действительно простой задачи все эти варианты являются взаимозаменяемыми; иначе говоря, у них очень похожие интерфейсы. Например, имея итератор, мы можем перемещаться по элементам с помощью операции `++` и использовать оператор `*` для доступа к символам. Если посмотреть на примеры кода, связанного с классом `Document`, то мы действительно можем заменить наш класс `vector<char>` классом `list<char>` или `string` без

каких-либо проблем. Такая взаимозаменяемость является фундаментальным преимуществом, потому что она позволяет нам сделать выбор, ориентируясь на эффективность. Но, перед тем как рассматривать вопросы эффективности, мы должны рассмотреть логические возможности этих типов: что такого может делать каждый из них, чего не могут другие?

- `Elem[]`. Не знает своего размера. Не имеет функций `begin()`, `end()` и других контейнерных функций-членов. Не может систематически проверять выход за пределы допустимого диапазона. Может передаваться функциям, написанным на языке C или в стиле языка C. Элементы в памяти располагаются последовательно в

смежных ячейках. Размер массива фиксируется на этапе компиляции. Операции сравнения (== и !=) и вывода (<<) используют указатель на первый элемент массива, а не на элементы.

- **vector<Elem>**. Может выполнять практически все, включая функции **insert()** и **erase()**. Предусматривает индексирование. Операции над списками, такие как **insert()** и **erase()**, как правило, связаны с перемещением элементов (что может оказаться неэффективным для крупных элементов и при большом количестве элементов). Может проверять выход за пределы допустимого диапазона. Элементы в памяти располагаются последовательно в смежных ячейках. Объект класса **vector** может увеличиваться (например, используя функцию **push_back()**). Элементы вектора хранятся в массиве (в виде непрерывной последовательности). Сравнение элементов осуществляется с помощью операторов ==, !=, <, <=, > и >=.
- **string**. Предусматривает все обычные и полезные операции, а также специфические функции для работы с текстами, такие как конкатенация (+ и +=). Элементы гарантированно хранятся в смежных ячейках памяти. Строки расширяемы. Сравнение элементов осуществляется с помощью операторов ==, !=, <, <=, > и >=.
- **list<Elem>**. Предусматривает все обычные и полезные операции, за исключением индексирования. Операции **insert()** и **erase()** можно выполнять без перемещения остальных элементов. Для хранения каждого элемента необходимы два дополнительных слова (для указателей на узлы). Списки расширяемы. Сравнение элементов осуществляется с помощью операторов ==, !=, <, <=, > и >=.

Как мы уже видели (в разделах 17.2 и 18.6), массивы полезны и необходимы для управления памятью на самом нижнем уровне, а также для обеспечения взаимодействия с программами, написанными на языке C (см. разделы 27.1.2 и 27.5). По сравнению с ними класс **vector** является более предпочтительным, потому что его легче использовать, к тому же он более гибкий и безопасный.

ПОПРОБУЙТЕ

Что означает этот список отличий в реальном коде? Определите массив символов **char**, **vector<char>**, **list<char>** и **string** со значением **"Hello"** и передайте каждый из этих объектов в функцию в качестве аргумента. В функции следует вывести количество символов в переданной строке, попытаться сравнить ее со строкой **"Hello"** (чтобы убедиться, что в функцию действительно передана строка **"Hello"**), а затем сравнить аргумент со строкой **"Howdy"**, чтобы выяснить, какое из этих слов появляется в словаре первым. Скопируйте аргумент в другую переменную того же типа.

ПОПРОБУЙТЕ

Выполните предыдущее задание для массива `int`, `vector<int>` и `list<int>` со значением `{1,2,3,4,5}`.

20.7.1. Операции `insert` и `erase`

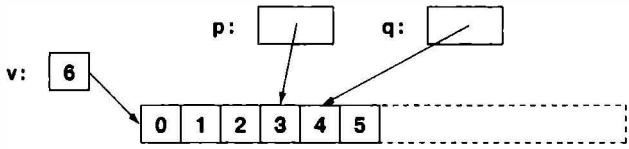
В качестве контейнера по умолчанию мы выбираем класс `vector` из стандартной библиотеки. Он имеет большинство из необходимых свойств, поэтому альтернативу ему следует использовать только при необходимости. Основной недостаток вектора заключается в том, что при выполнении операций, характерных для списка (`insert()` и `erase()`), в векторе происходит перемещение остальных элементов; если вектор содержит большое количество элементов или элементы вектора сами являются крупными объектами, цена такого перемещения может оказаться неприемлемой. Однако слишком беспокоиться об этом не следует. Мы для эксперимента без заметных проблем считали полмиллиона значений с плавающей точкой в вектор, используя функцию `push_back()`. Измерения подтвердили, что предварительное выделение памяти не приводит к заметным последствиям.

Всегда проводите измерения перед тем, как вносить в код значительные изменения для повышения эффективности; точно предположить, как изменится эффективность кода при том или ином изменении, трудно даже экспертам.

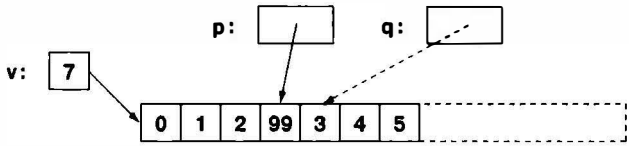
Как указывалось в разделе 20.6, перемещение элементов связано с логическим ограничением: выполняя операции, характерные для списков (такие, как `insert()`, `erase()` или `push_back()`), не следует сохранять итераторы или указатели на элементы вектора; ведь, если элемент будет перемещен, ваш итератор или указатель будет указывать на неправильный элемент (или вообще может не указывать на элемент вектора). В этом заключается принципиальное преимущество класса `list` (и класса `map`; см. раздел 21.6) над классом `vector`. Если вам необходима коллекция крупных объектов или объектов, на которые приходится указывать из многих частей программы, лучше использовать класс `list`.

Сравним функции `insert()` и `erase()` в классах `vector` и `list`. Сначала рассмотрим пример, разработанный только для того, чтобы продемонстрировать ключевые моменты.

```
vector<int>::iterator p = v.begin();// Берем вектор,
++p; ++p; ++p;                      // итератор устанавливаем
                                    // на 4-й элемент
vector<int>::iterator q = p;
++q;                                // Итератор указывает
                                    // на 5-й элемент
```

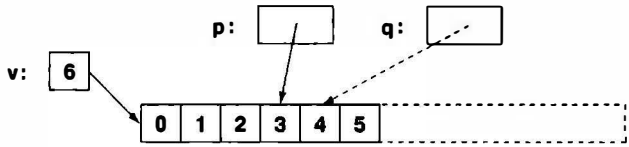



```
p = v.insert(p,99); // p указывает на вставленный элемент
```



Теперь итератор `q` становится неверным. При увеличении размера вектора элементы могут быть перемещены в другое место. Если вектор `v` имеет запас памяти, то при увеличении выделенная для элементов память останется на том же самом месте, а итератор `q`, скорее всего, будет указывать на элемент со значением 3, а не на элемент со значением 4.

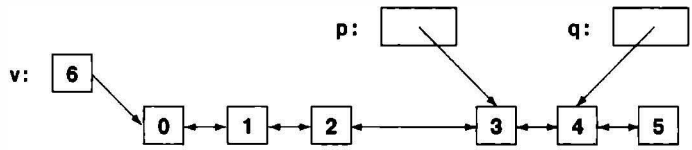
```
p = v.erase(p); // Итератор p указывает на элемент,
                // следующий за удаленным
```



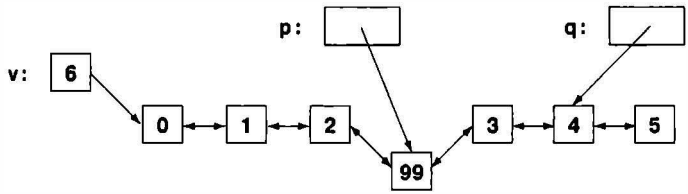
Итак, вызов функции `erase()` после функции `insert()` приводит нас к тому же вектору, что и ранее, но итератор `q` вновь указывает на новый элемент. Более того, возможна ситуация, когда при вставке элемента в выделенной для массива памяти не хватит места для перемещения всех элементов вправо, и тогда будет выделена новая память, все элементы будут перемещены в новое местоположение, а старая память освобождена. В таком случае итератор `q` после вставки элемента будет указывать на некоторое место в памяти, не имеющее никакого отношения к вектору.

Для сравнения проделаем то же самое со списком `list`.

```
list<int>::iterator p = v.begin(); // Берем список,
++p; ++p; ++p;                     // устанавливаем итератор
                                   // на 4-й элемент
list<int>::iterator q = p;
++q;                               // Итератор указывает
                                   // на 5-й элемент
```

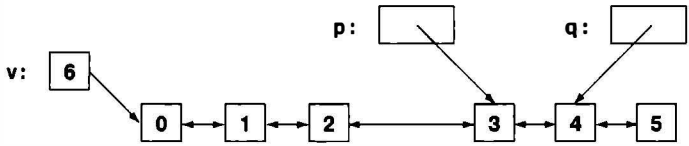


```
p = v.insert(p,99); // p указывает на вставленный элемент
```



Обратите внимание на то, что итератор **q** по-прежнему указывает на элемент, имеющий значение **4**.

```
p = v.erase(p); // Итератор p указывает на элемент, следующий
                // за удаленным
```



И снова мы оказались там, откуда начинали. Однако, в отличие от класса **vector**, работая с классом **list**, мы не перемещали элементы, и итератор **q** всегда оставался корректным.

Объект класса **list<char>** занимает по меньшей мере в три раза больше памяти, чем остальные три альтернативы; обычно объект **list<char>** использует 12 байтов на элемент, объект класса **vector<char>** — один байт на элемент. Для большого количества символов это обстоятельство может оказаться важным.

А в чем заключается преимущество класса **vector** перед классом **string**? На первый взгляд, список их возможностей свидетельствует о том, что класс **string** может делать все то же самое, что и класс **vector**, и даже больше. В этом и заключается часть проблемы: поскольку класс **string** может делать намного больше, его труднее оптимизировать. Фактически класс **vector** можно оптимизировать для операций, работающих с памятью, таких как **push_back()**, а класс **string** — нет. В то же время в классе **string** можно оптимизировать копирование, работу с короткими строками и С-строками. В примере с текстовым редактором мы выбрали класс **vector**, так как использовали функции **insert()** и **erase()**. Это решение объяснялось вопросами эффективности. Основное логическое отличие векторов от строк заключается в том, что мы можем создавать векторы, содержащие элементы практически любых типов. Возможность выбора появляется только тогда, когда мы работаем с символами. В заключение мы рекомендуем использовать класс **vector**, а не **string**, если только вам не нужны специфичные строковые операции, такие как конкатенация или чтение слов, разделенных пробельными символами.

20.8. Адаптация нашего класса `vector` к STL

После добавления функций `begin()`, `end()` и псевдонимов типов в разделе 20.5 классу `vector` не достает только функций `insert()` и `erase()`, чтобы стать близким аналогом класса `std::vector`.

```
template<typename T, typename A = allocator<T>>
// Требует Element<T>() и Allocator<A>() (раздел 19.3.3)
class vector {
    int sz;     // Размер
    T* elem;    // Указатель на элементы
    int space; // Количество элементов плюс
               // количество свободных ячеек
    A alloc;    // Распределитель памяти для элементов
public:
    // ... Все остальное описано в главе 19 и разделе 20.5 ...
    using iterator = T*; // T* – максимально простой итератор

    iterator insert(iterator p, const T& val);
    iterator erase(iterator p);
};
```

Здесь мы вновь использовали в качестве типа итератора указатель на элемент `T*`. Это простейшее из всех возможных решений. Разработку итератора, проверяющего выход за пределы допустимого диапазона, читатели могут выполнить в качестве упражнения (упр. 18).

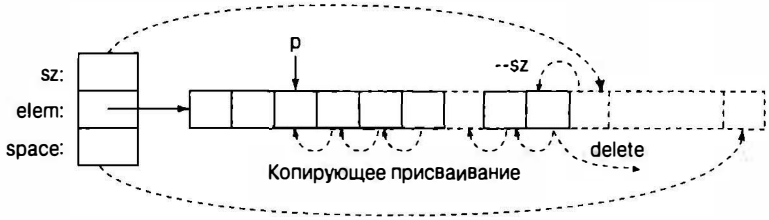
Как правило, операции над списками, такие как `insert()` и `erase()`, типами данных, хранящих элементы в смежных ячейках памяти (такими, как `vector`) не предоставляются. Однако операции над списками наподобие `insert()` и `erase()` оказываются, несомненно, полезными и удивительно эффективными при работе с небольшими векторами или при небольшом количестве элементов. Мы регулярно сталкиваемся с полезностью как функции `push_back()`, как и других традиционных операций над списками.

По сути мы реализуем функцию `vector<T,A>::erase()` путем копирования всех элементов, расположенных после удаляемого. Используя определение класса `vector` из раздела 19.3.7 с указанными дополнениями, мы получаем следующий код.

```
template<typename T, typename A>
// Требует Element<T>() и Allocator<A>() (раздел 19.3.3)
vector<T,A>::iterator vector<T,A>::erase(iterator p)
{
    if (p==end()) return p;
    for (auto pos = p+1; pos!=end(); ++pos)
        *(pos-1) = *pos;          // Копируем элементы на
                                  // одну позицию влево
    alloc.destroy(&*(end()-1)); // Уничтожаем лишнюю копию
                                  // последнего элемента
```

```
    --sz;
    return p;
}
```

Этот код легче понять, если представить его в графическом виде.



Код функции **erase()** довольно прост, но, пожалуй имеет смысл попытаться выполнить несколько примеров на бумаге. Правильно ли обрабатывается пустой объект класса **vector**? Почему нужна проверка **p==end()**? Что если мы хотим удалить последний элемент вектора? Не был бы этот код проще, если бы мы использовали индексирование?

Реализация функции **vector<T,A>::insert()** является немного более сложной.

```
template<typename T, typename A>
// Требует Element<T>() и Allocator<A>() (раздел 19.3.3)
vector<T,A>::iterator
    vector<T,A>::insert(iterator p, const T& val)
{
    int index = p-begin();
    if (size()==capacity())              // Обеспечиваем
        reserve(size()==0?8:2*size()); // достаточное место

    // Сначала копируем последний элемент в
    // неинициализированную память:
    alloc.construct(elem+sz,*back());
    ++sz;
    iterator pp = begin()+index; // Место для размещения val
    for (auto pos = end()-1; pos!=pp; --pos)
        *pos = *(pos-1) ;          // Копируем элементы на одну
                                   // позицию вправо
    *(begin()+index) = val;        // "Вставка" val
    return pp;
}
```

Обратите внимание на следующее.

- Итератор не может указывать на что-то вне последовательности; поэтому мы используем указатели, такие как **elem+space**. Это одна из причин, по которым распределители памяти реализованы на основе указателей, а не итераторов.

- Когда мы используем функцию `reserve()`, элементы вектора могут быть перенесены в новую область памяти. Следовательно, мы должны запомнить индекс вставляемого элемента, а не итератор, указывающий на него. Когда элементы вектора переносятся в памяти на другое место, итераторы, указывающие на них, становятся некорректными — их можно интерпретировать как указатели на старые адреса.
- Наше использование распределителя памяти `A` является интуитивным, но не точным. Если вам когда-либо придется реализовывать контейнер, внимательно изучите стандарт.
- Тонкости, подобные указанным, позволяют избежать непосредственной работы с памятью на низком уровне. Естественно, стандартный класс `vector`, как и остальные контейнеры стандартной библиотеки, корректно реализует эти важные семантические тонкости. Это одна из причин, по которым мы настоятельно рекомендуем использовать стандартную библиотеку, а не "кустарные" решения.

По причинам, связанным с эффективностью, мы не должны применять функции `insert()` и `erase()` в средине больших векторов; в этом случае лучше использовать класс `list` (или класс `map`; см. раздел 21.6). Однако операции `insert()` и `erase()` доступны во всех векторах, а их производительность при перемещении небольшого количества данных является непревзойденной, поскольку современные компьютеры выполняют такое копирование очень быстро (см. упр. 20). Избегайте (связанных) списков, состоящих из небольшого количества маленьких элементов.

20.9. Адаптация встроенных массивов к библиотеке STL

Мы многократно указывали на недостатки встроенных массивов: они неявно преобразуют указатели при малейшем поводе, их нельзя скопировать с помощью присваивания, они не знают своего размера (см. раздел 18.6.2) и т.д. Мы также отмечали и их основное преимущество: они превосходно моделируют физическую память.

Для того чтобы использовать преимущества как массивов, так и контейнеров, мы можем создать контейнер типа `array`, обладающий достоинствами массивов, но не имеющий их недостатков. Вариант класса `array` был включен в стандарт как часть технического отчета Комитета по стандартизации языка C++. Поскольку возможности, включенные в этот отчет, не обязательны для реализации во всех компиляторах, класс `array` в вашей стандартной библиотеке может отсутствовать. Однако его идея проста и полезна.

```
template <typename T, int N>   // Требует Element<T>()
// Не вполне стандартный массив
struct array {
    using value_type = T;
    using iterator = T*;
    using const_iterator = const T*;
    using size_type = unsigned int; // Тип индекса

    T elems[N];
    // Явное создание/копирование/уничтожение не требуется

    iterator begin() { return elems; }
    const_iterator begin() const { return elems; }
    iterator end() { return elems+N; }
    const_iterator end() const { return elems+N; }

    size_type size() const;

    T& operator[](int n) { return elems[n]; }
    const T& operator[](int n) const { return elems[n]; }

    const T& at(int n) const; // Доступ с проверкой диапазона
    T& at(int n);             // Доступ с проверкой диапазона

    T* data() { return elems; }
    const T* data() const { return elems; }
};
```

Это определение не полное и не совсем соответствует стандарту, но оно хорошо иллюстрирует основную идею. Кроме того, оно позволяет использовать класс **array**, если его нет в вашей стандартной библиотеке. Если же он есть, то искать его следует в заголовке **<array>**. Обратите внимание на то, что поскольку объекту класса **array<T,N>** известен его размер **N**, мы можем (и должны) предусмотреть операторы =, ==, != так же, как и для класса **vector**.

В качестве примера используем массив со стандартной функцией **high()** из раздела 20.4.2:

```
void f()
{
    array<double,6> a = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
    array<double,6>::iterator p = high(a.begin(), a.end());
    cout << "Максимальное значение равно " << *p << endl;
}
```

Обратите внимание на то, что мы совершенно не думали о классе **array**, когда писали функцию **high()**. Возможность применять функцию **high()** к объекту класса **array** является простым следствием того, что в обоих случаях мы придерживались стандартных соглашений.

20.10. Обзор контейнеров

Библиотека STL предоставляет программисту ряд контейнеров.

Стандартные контейнеры	
`vector`	Непрерывная последовательность элементов. Рекомендуем использовать как контейнер по умолчанию
`list`	Двухсвязный список. Рекомендуем использовать, когда возникает необходимость вставить и удалить элементы без перемещения существующих элементов
`deque`	Объединение списка и вектора. Рекомендуем не использовать, пока вы не станете экспертом в области алгоритмов и машинной архитектуры
`map`	Сбалансированное упорядоченное дерево. Рекомендуем использовать, когда нужен доступ к элементам по значению ключа (см. разделы 21.6.1–21.6.3)
`multimap`	Сбалансированное упорядоченное дерево, в котором может храниться несколько копий ключа. Рекомендуем использовать, когда нужен доступ к элементам по значению ключа (см. разделы 21.6.1–21.6.3)
`unordered_map`	Хеш-таблица; оптимизированная версия класса `map`. Рекомендуем использовать, если нужна высокая производительность и вы можете разработать хорошую хеш-функцию (см. раздел 21.6.4)
`unordered_multimap`	Хеш-таблица, в которой может храниться несколько копий ключа; оптимизированная версия класса `multimap`. Рекомендуем использовать, если вам нужна высокая производительность и вы можете разработать хорошую хеш-функцию (см. раздел 21.6.4)
`set`	Сбалансированное упорядоченное дерево. Рекомендуем использовать при необходимости отслеживать отдельные значения (см. раздел 21.6.5)
`multiset`	Сбалансированное упорядоченное дерево, в котором может храниться несколько копий ключа. Рекомендуем использовать при необходимости отслеживать отдельные значения (см. раздел 21.6.5)
`unordered_set`	Похож на класс `unordered_map`, но для значений, а не для пар (ключ,значение)
`unordered_multiset`	Похож на класс `unordered_multimap`, но для значений, а не для пар (ключ,значение)
`array`	Массив фиксированного размера, в котором нет большинства недостатков встроенных массивов (см. раздел 20.9)

Огромный массив дополнительной информации об этих контейнерах и их использовании можно найти в книгах и документации, размещенной в Интернете. Перечислим несколько источников, заслуживающих доверия.

1. Josuttis, Nicholai M. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 2012. ISBN 978-0321623218. Используйте только второе издание![1]
2. Lippman, Stanley B., Jose Lajoie, and Barbara E. Moo. *The C++ Primer.* Addison-Wesley, 2012. ISBN 978-0-321-71411-4. Используйте только пятое издание![2]
3. Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley, 2012. ISBN 978-0321714114. Используйте только четвертое издание!
4. Документацию о реализации библиотеки STL и библиотеки потоков ввода-вывода компании SGI (Silicon Graphics International) можно найти на веб-странице `www.sgi.com/tech/stl`. Обратите внимание, что на этой веб-странице имеются законченные программы.

Вы чувствуете себя обманутым? Полагаете, что мы должны были описать все контейнеры и показать, как их использовать? Это невозможно. Существует слишком много стандартных возможностей, полезных приемов и библиотек, чтобы описать их в одной книге. Программирование слишком

богато возможностями, чтобы их мог освоить один человек. Кроме того, зачастую программирование — это искусство. Как программист вы должны привыкнуть искать информацию о возможностях языка, библиотеках и методах программирования. Программирование — динамичная и быстро развивающаяся отрасль, поэтому необходимо довольствоваться тем, что вы знаете, и спокойно относиться к тому, что существуют вещи, которых вы не знаете. "Искать в справочнике" — это вполне разумный ответ на многие вопросы. По мере увеличения вашего опыта вы будете поступать так все чаще.

С другой стороны, вы обнаружите, что, освоив классы `vector`, `list` и `map`, а также стандартные алгоритмы, описанные в главе 21, вы легко научитесь работать с остальными контейнерами из библиотеки STL. Вы обнаружите также, что знаете все, что требуется для работы с нестандартными контейнерами, и сможете программировать их сами.

Что такое контейнер? Определение этого понятия можно найти в любом из указанных выше источников. Здесь лишь дадим неформальное определение. Контейнер библиотеки STL обладает следующими свойствами.

- Контейнер представляет собой последовательность элементов [`begin()`:`end()`).

---

[1] Имеется русский перевод: Николаи М. Джосаттис. *Стандартная библиотека C++: справочное руководство, 2-е изд.* — М.: ООО "И.Д. Вильямс", 2014.

[2] Имеется русский перевод: Стенли Б. Липпман, Жози Лажойе, Барбара Э. Му. *Язык программирования C++. Базовый курс, 5-е издание* — М.: ООО "И.Д. Вильямс", 2015.

- Контейнер предоставляет операцию копирования, которая копирует элементы. Копирование можно выполнить с помощью присваивания или конструктора копирования.
- Тип элементов имеет имя `value_type`.
- Контейнер имеет итераторы с именами `iterator` и `const_iterator`. Итераторы предоставляют возможность операций `*`, `++` (как префиксный, так и постфиксный), `==` и `!=` с соответствующей семантикой. Итераторы класса `list` предоставляют также оператор `--` для перемещения по последовательности в обратном направлении; такие итераторы называют *двунаправленными* (bidirectional iterator). Итераторы класса `vector` также предоставляют операции `--`, `[]`, `+` и `-`. Такие итераторы называются *итераторами с произвольным доступом* (random-access iterators) (см. раздел 20.10.1).
- Контейнеры обеспечивают функции `insert()` и `erase()`, `front()` и `back()`, `push_back()` и `pop_back()`, `size()` и т.д.; классы `vector` и `map` обеспечивают также операцию индексирования (оператор `[]`).
- Контейнеры предоставляют операторы сравнения (`==`, `!=`, `<`, `<=`, `>` и `>=`) для сравнения элементов. Для операций `<`, `<=`, `>` и `>=` контейнеры используют лексикографическое упорядочивание; иначе говоря, они сравнивают элементы, начиная с первого.

Цель этого списка — небольшой обзор для читателей. Более детальная информация приведена в приложении Б. Более точная спецификация и полный список операций приведены в упомянутой выше книге *The C++ Programming Language* или в стандарте языка.

Некоторые типы данных обладают многими свойствами стандартных контейнеров, но не всеми. Мы иногда называем их "почти контейнерами". Наиболее интересными среди них являются следующие.

"Почти контейнеры"	
Встроенный массив `T[n]`	Не имеет функции `size()` и других функций-членов. Если есть возможность выбора, рекомендуем вместо него использовать контейнер, такой как `vector`, `string` или `array`
`string`	Хранит только символы, зато обеспечивает полезные операции для работы с текстом, такие как конкатенация (`+` и `+=`). Рекомендуем использовать стандартный класс `string` вместо других видов строк
`valarray`	Числовой вектор с векторными операциями, но с множеством ограничений, нацеленных на получение высокой производительности. Используйте его только при необходимости выполнять много векторных вычислений

Кроме того, многие люди и организации разрабатывают собственные контейнеры, удовлетворяющие (или почти удовлетворяющие) требованиям стандарта.

Если у вас есть сомнения, используйте класс **vector**. Если только у вас нет весомых причин поступать иначе, используйте класс **vector**.

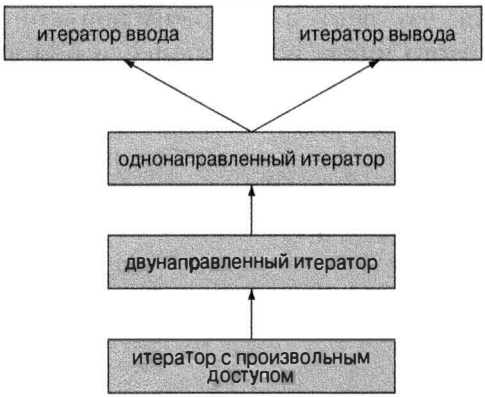
20.10.1. Категории итераторов

Мы говорили об итераторах так, будто все они являются взаимозаменяемыми. Однако они взаимозаменяемы только с точки зрения простейших операций, таких как перемещение по последовательности с однократным считыванием каждого элемента. Если вы хотите большего, например, перемещаться в обратном направлении или обеспечить произвольный доступ, то вам нужны более совершенные итераторы.

Категории итераторов	
Входной итератор (итератор ввода)	Можно перемещаться вперед, используя оператор ++, и считывать значения элементов с помощью оператора *. Именно такой вид итераторов предлагает поток ввода **istream** (см. раздел 21.7.2). Если выражение **(*p).m** является корректным, то в качестве сокращения можно использовать запись **p->m**
Выходной итератор (итератор вывода)	Можно перемещаться вперед, используя оператор ++, и выводить значения элементов с помощью оператора *. Именно такой вид итераторов предлагает поток вывода **ostream** (см. раздел 21.7.2)
Однонаправленный итератор	Можно перемещаться вперед, используя оператор ++, считывая или записывая (если элементы не являются константными) значения элементов с помощью оператора *. Если выражение **(*p).m** является корректным, то в качестве сокращения можно использовать запись **p->m**
Двунаправленный итератор	Можно перемещаться вперед (используя оператор ++) и назад (используя оператор --), считывая или записывая (если элементы не являются константными) значения элементов с помощью оператора *. Если выражение **(*p).m** является корректным, то в качестве сокращения можно использовать запись **p->m**
Итератор с произвольным доступом	Можно перемещаться вперед (используя оператор ++) и назад (используя оператор --), считывая или записывая (если элементы не являются константными) значения элементов с помощью оператора * или **[]**. Можно индексировать итератор с произвольным доступом, а также добавлять к нему или вычитать из него целое число, используя операторы + и –. Можно вычислить расстояние между двумя итераторами с произвольным доступом для одной и той же последовательности, просто вычитая один из другого. (Именно такой вид итераторов обеспечивает класс **vector**.) Если выражение **(*p).m** является корректным, то в качестве сокращения можно использовать запись **p->m**

Глядя на предоставляемые итераторами операции, легко убедиться в том, что вместо итераторов для записи или чтения можно использовать двунаправленный итератор. Кроме того, двунаправленный итератор одно-

временно является однонаправленным, а итератор с произвольным доступом — двунаправленным. В графическом виде категории итераторов можно изобразить следующим образом.



Обратите внимание на то, что категории итераторов не являются классами. Иерархия категорий итераторов не является иерархией классов, реализованной с помощью наследования.

Задание

1. Определите массив чисел типа `int` с десятью элементами { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }.
2. Определите объект класса `vector<int>` с этими же десятью элементами.
3. Определите объект класса `list<int>` с этими же десятью элементами.
4. Определите второй массив, вектор и список, каждый из которых инициализируется первым массивом, вектором или списком соответственно.
5. Увеличьте значение каждого элемента в массиве на 2; увеличьте значение каждого элемента в массиве на 3; увеличьте значение каждого элемента в списке на 5.
6. Напишите простую операцию `copy()`

```
template<typename Iter1, typename Iter2>
// Требует Input_iterator<Iter1>() и Output_iterator<Iter2>()
Iter2 copy(Iter f1, Iter1 e1, Iter2 f2);
```

   копирующую последовательность [`f1`,`e1`) в последовательность [`f2`,`f2+(e1-f1)`), и точно так же, как и стандартная библиотечная функция копирования, возвращающую итератор `f2+(e1-f1)`. Обратите внимание на то, что если `f1==e1`, то последовательность пуста и копировать нечего.

7. Используйте вашу функцию `copy()` для копирования массива в вектор, а списка — в массив.
8. Используйте стандартную библиотечную функцию `find()` для того, чтобы убедиться, что вектор содержит значение 3, и выведите на экран позицию этого числа в векторе (если оно в нем есть). Используйте стандартную библиотечную функцию `find()`, чтобы убедиться, что список содержит значение 27, и выведите на экран позицию этого числа в списке (если оно в нем есть). "Позиция" первого элемента равна нулю, позиция второго элемента равна единице и т.д. Если функция `find()` возвращает итератор, установленный на конец последовательности, значит, искомое значение в последовательности не найдено.

Не забывайте тестировать программу после каждого этапа.

Контрольные вопросы

1. Почему программы, написанные разными людьми, выглядят по-разному? Приведите примеры.
2. Какие простые вопросы мы обычно задаем, думая о данных?
3. Перечислите разные способы хранения данных.
4. Какие основные операции можно выполнить с коллекцией данных?
5. Каких принципов следует придерживаться при хранении данных?
6. Что такое последовательность STL?
7. Что такое итератор STL? Какие операции поддерживают итераторы?
8. Как переместить итератор на следующий элемент?
9. Как переместить итератор на предыдущий элемент?
10. Что произойдет, если вы попытаетесь переместить итератор за конец последовательности?
11. Какие виды итераторов могут перемещаться к предыдущему элементу?
12. Почему полезно отделять данные от алгоритмов?
13. Что такое STL?
14. Что такое связанный список? Чем он фундаментально отличается от вектора?
15. Что такое узел (в связанном списке)?
16. Что делает функция `insert()`? Что делает функция `erase()`?
17. Как определить, что последовательность пуста?
18. Какие операции предусмотрены в итераторе класса `list`?
19. Как обеспечить обход контейнера, используя STL?
20. В каких ситуациях лучше использовать класс `string`, а не `vector`?

21. В каких ситуациях лучше использовать класс `list`, а не `vector`?
22. Что такое контейнер?
23. Что должны делать функции `begin()` и `end()` в контейнере?
24. Какие контейнеры предоставляет библиотека STL?
25. Что такое категория итератора? Какие категории итераторов предлагает библиотека STL?
26. Какие операции предусмотрены в итераторе с произвольным доступом, но не поддерживаются двунаправленным итератором?

Термины

`auto`	алгоритм	обход
`begin()`	двусвязный список	односвязный список
`end()`	итератор	последовательность
`erase()`	итерация	псевдоним типа
`insert()`	контейнер	пустая последовательность
`using`	контейнер array	связанный список
`value_type`	непрерывная память	элемент

Упражнения

1. Если вы еще не выполнили задания из врезок **Попробуйте**, то сделайте это сейчас.
2. Попробуйте запрограммировать пример с Джеком и Джилл из раздела 20.1.2. Для тестирования используйте несколько небольших файлов.
3. Проанализируйте пример с палиндромом (см. раздел 18.7); еще раз выполните задание из п. 2, используя разные приемы.
4. Найдите и исправьте ошибки, сделанные в примере с Джеком и Джилл в разделе 20.3.1, используя приемы библиотеки STL.
5. Определите операторы ввода и вывода (`>>` и `<<`) для класса `vector`.
6. Напишите операцию "найти и заменить" для класса `Document`, используя информацию из раздела 20.6.2.
7. Определите лексикографически последнюю строку в неупорядоченном классе `vector<string>`.
8. Напишите функцию, подсчитывающую количество символов в объекте класса `Document`.
9. Напишите программу, подсчитывающую количество слов в объекте класса `Document`. Предусмотрите две версии: одну, в которой слово — это "последовательность символов, разделенных пробельными символами", и вторую, в которой слово — это "неразрывная после-

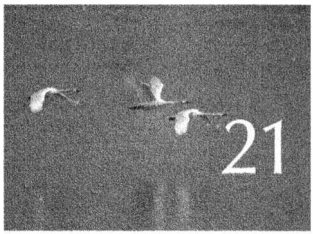
довательность алфавитных символов". Например, при первом определении выражения `alpha.numeric` и `as12b` это слова, а при втором каждое из них рассматривается как два слова.

10. Напишите подсчитывающую слова программу, в которой пользователь мог бы сам задавать набор символов-разделителей.
11. Создайте объект класса `vector<double>` и скопируйте в него элементы списка типа `list<int>`, передавая его как параметр (по ссылке). Убедитесь, что копия полна и верна. Затем выведите на экран элементы в порядке возрастания их значений.
12. Завершите определение класса `list` из разделов 20.4.1 и 20.4.2 и продемонстрируйте работу функции `high()`. Выделите память для объекта класса `Link`, представляющего узел, следующий за концом списка.
13. На самом деле в классе `list` нам не нужен реальный объект класса `Link`, расположенный за последним элементом. Модифицируйте свое решение из предыдущего упражнения так, чтобы в качестве указателя на несуществующий объект класса `Link` (`list<Elem>::end()`) использовалось значение 0; иначе говоря, размер пустого списка может быть равен размеру отдельного указателя.
14. Определите односвязный список `slist`, ориентируясь на стиль класса `std::list`. Какие операции из класса `list` стоило бы исключить из класса `slist`, поскольку он не содержит указателя на предыдущий элемент?
15. Определите класс `pvector`, похожий на вектор указателей, с тем отличием, что он содержит указатели на объекты, а его деструктор уничтожает все объекты, на которые указывают указатели-элементы.
16. Определите класс `ovector`, похожий на класс `pvector`, с тем отличием, что операции `[]` и `*` возвращают не указатели, а ссылки на объекты, на которые указывают соответствующие элементы.
17. Определите класс `ownership_vector`, хранящий указатели на объект, как и класс `pvector`, но предусматривающий механизм, позволяющий пользователю решить, какие из объектов принадлежат вектору (т.е. какие объекты должны быть удалены деструктором класса). *Указание*: это простое упражнение, если вы вспомните главу 13.
18. Определите итератор с проверкой выхода за пределы допустимого диапазона для класса `vector` (итератор с произвольным доступом).
19. Определите итератор с проверкой выхода за пределы допустимого диапазона для класса `list` (двунаправленный итератор).
20. Проведите эксперимент по сравнению временных затрат при работе с классами `vector` и `list`. Способ измерения длительности работы программы изложен в разделе 26.6.1. Сгенерируйте *N* случайных

целых чисел в диапазоне [0,*N*). Вставьте каждое сгенерированное число в вектор `vector<int>` (после каждой вставки увеличивающийся на один элемент). Поддерживайте вектор в отсортированном виде; иначе говоря, значение должно быть вставлено так, чтобы все предыдущие значения были меньше или равны ему, а все последующие значения должны быть больше него. Выполните те же действия, используя для хранения целых чисел класс `list<int>`. При каких значениях *N* класс `list` обеспечивает более высокое быстродействие, чем класс `vector`? Попробуйте объяснить результаты эксперимента. Впервые этот эксперимент был предложен Джоном Бентли (John Bentley).

Послесловие

Если бы у нас было *N* видов контейнеров для данных и *M* операций, которые мы хотели бы над ними выполнить, то мы могли бы легко написать *N*M* фрагментов кода. Если бы данные были *K* разных типов, то нам пришлось бы написать *N*M*K* фрагментов кода. Библиотека STL решает эту проблему, разрешая задавать тип элемента в виде параметра (устраняя множитель *K*) и отделяя доступ к данным от алгоритмов. Используя итераторы для доступа к данным в любом контейнере и в любом алгоритме, мы можем ограничиться *N+M* фрагментами кода. Это огромное облегчение. Например, если бы у нас было 12 контейнеров и 60 алгоритмов, то прямолинейный подход потребовал бы создания 720 функций, в то время как стратегия, принятая в библиотеке STL, требует только 60 функций и 12 определений итераторов: тем самым мы экономим 90% работы. На самом деле это заниженная оценка экономии усилий, потому что многие алгоритмы принимают две пары итераторов, и эти пары могут быть разных типов (см., например, упр. 6). Кроме того, в библиотеке STL приняты соглашения, касающиеся определения алгоритмов, которые упрощают создание корректного кода и его объединение с другими кодами, что также экономит много времени.



21

Алгоритмы и ассоциативные массивы

Теоретически практика проста.
— Трюве Реенскауг (Trygve Reenskaug)

В этой главе мы завершаем представление фундаментальных идей, лежащих в основе библиотеки STL, и обзор возможностей этой библиотеки. Здесь мы сосредоточим свое внимание на алгоритмах. Наша главная цель — ознакомить читателей с десятками весьма полезных алгоритмов, которые сэкономят им дни, если не месяцы, работы. Описание каждого алгоритма сопровождается примерами его использования и указанием методов программирования, которые его поддерживают. Вторая цель, которую мы преследуем, — научить читателей писать элегантные и эффективные алгоритмы в тех случаях, когда ни стандартная, ни другие доступные библиотеки не могут удовлетворить их потребности. Кроме того, мы рассмотрим еще три контейнера: **map**, **set** и **unordered_map**.

21.1. Алгоритмы стандартной библиотеки

Стандартная библиотека содержит около шестидесяти алгоритмов. Все они иногда оказываются кому-то полезными. Мы сосредоточимся на алгоритмах, которые полезны многим, а также на тех, которые иногда очень полезны некоторым.

Избранные стандартные алгоритмы	
`r=find(b,e,v)`	Итератор `r` указывает на первое вхождение элемента `v` в последовательность `[b,e)`
`r=find_if(b,e,p)`	Итератор `r` ссылается на первое вхождение элемента `x`, такого, что предикат `p(x)` имеет значение `true`, в последовательность `[b,e)`
`x=count(b,e,v)`	`x` — количество вхождений элемента `v` в диапазон `[b,e)`
`x=count_if(b,e,p)`	`x` — количество элементов в последовательности `[b,e)`, таких, что предикат `p(x)` имеет значение `true`
`sort(b,e)`	Упорядочивает последовательность `[b,e)` с помощью оператора `<`
`sort(b,e,p)`	Упорядочивает последовательность `[b,e)` с помощью предиката `p`
`copy(b,e,b2)`	Копирует последовательность `[b,e)` в последовательность `[b2,b2+(e-b))`; требуется достаточное количество ячеек после итератора `b2`

Избранные стандартные алгоритмы	
`unique_copy(b,e,b2)`	Копирует последовательность [`b`,`e`) в последовательность [`b2`,`b2+(e-b)`); смежные дубли не копируются
`merge(b,e,b2,e2,r)`	Объединяет две упорядоченные последовательности [`b2`,`e2`) и [`b`,`e`) в последовательность [`r`,`r+(e-b)+(e2-b2)`)
`r=equal_range(b,e,v)`	`r` — подпоследовательность упорядоченного диапазона [`b`,`e`) со значением `v`, по сути — бинарный поиск элемента `v`
`equal(b,e,b2)`	Проверяет, равны ли все соответствующие элементы последовательностей [`b`,`e`) и [`b2`,`b2+(e-b)`)
`x=accumulate(b,e,i)`	`x` — сумма числа `i` и элементов последовательности [`b`,`e`)
`x=accumulate(b,e,i,op)`	Аналогичен предыдущему `accumulate`, но сумма вычисляется с помощью операции `op`
`x=inner_product(b,e,b2,i)`	`x` — скалярное произведение [`b`,`e`) и [`b2`,`b2+(e-b)`); `i` — значение, инициализирующее сумму произведений
`x=inner_product(b,e,b2,i,op,op2)`	Аналогичен предыдущему `inner_product`, но вместо операций `+` и `*` использует операции `op` и `op2`

По умолчанию проверка равенства выполняется с помощью оператора `==`, а упорядочение — на основе оператора сравнения `<` (меньше). Алгоритмы стандартной библиотеки определены в заголовочном файле `<algorithm>`. Более подробную информацию можно найти в приложении Б.5 и в источниках в разделах 21.2–21.5. Эти алгоритмы получают одну или две последовательности. Входная последовательность определяется парой итераторов; результирующая последовательность — итератором, установленным на ее первый элемент. Как правило, алгоритм параметризуется одной или несколькими операциями, которые можно определить либо с помощью функциональных объектов, либо с помощью собственно функций. Алгоритмы обычно сообщают о сбоях, возвращая итератор, указывающий на конец входной последовательности. Например, алгоритм `find(b,e,v)` вернет элемент `e`, если не найдет значение `v`.

21.2. Простейший алгоритм: `find()`

Вероятно, простейшим из полезных алгоритмов является алгоритм `find()`. Он находит элемент последовательности с заданным значением.

```
template<typename In, typename T>
// Требует Input_iterator<In>()
// и Equality_comparable<Value_type<T>>() (раздел 19.3.3)
In find(In first, In last, const T& val)
// Находит первый элемент в [first,last), равный val
{
    while (first!=last && *first != val) ++first;
    return first;
}
```

Посмотрим на определение алгоритма `find()`. Естественно, вы можете использовать алгоритм `find()`, не зная, как именно он реализован, — фактически мы его уже применяли (например, в разделе 20.6.2). Однако определение алгоритма `find()` иллюстрирует много полезных проектных идей, поэтому оно достойно изучения.

Прежде всего, алгоритм `find()` применяется к последовательности, определенной парой итераторов. Мы ищем значение `val` в полуоткрытой последовательности [`first`,`last`). Результат, возвращаемый функцией `find()`, является итератором. Он указывает либо на первый элемент последовательности, равный значению `val`, либо на элемент `last`. Возвращение итератора на элемент, следующий за последним элементом последовательности, — самый распространенный способ, с помощью которого алгоритмы STL сообщают о том, что элемент не найден. Так что мы можем использовать алгоритм `find()` следующим образом.

```
void f(vector<int>& v, int x)
{
    auto p = find(v.begin(),v.end(),x);
    if (p!=v.end()) {
        // Мы нашли x в v
    }
    else {
        // В v нет элемента, равного x
    }
    // ...
}
```

В этом примере, как в большинстве случаев, последовательность содержит все элементы контейнера (в данном случае — вектора). Мы сравниваем возвращенный итератор с концом последовательности, чтобы узнать, найден ли искомый элемент. Типом возвращаемого значения является итератор, переданный в качестве аргумента.

Чтобы избежать именования возвращаемого типа, мы используем ключевое слово `auto`. Объект, определенный как имеющий "тип" `auto`, получает свой тип от инициализатора, например:

```
auto ch = 'c'; // ch имеет тип char
auto d = 2.1;  // d имеет тип double
```

Спецификатор типа `auto` особенно полезен в обобщенном коде, таком как алгоритм `find()`, где записать фактический тип может быть весьма утомительно (в данном случае это тип `vector<int>::iterator`).

Теперь мы знаем, как используется алгоритм `find()`, а следовательно, как использовать группу аналогичных алгоритмов, следующих тем же соглашениям. Однако, прежде чем переходить к другим алгоритмам, внимательнее посмотрим на определение алгоритма `find()`.

```
template<typename In, typename T>
// Требует Input_iterator<In>()
// и Equality_comparable<Value_type<T>>() (раздел 19.3.3)
In find(In first, In last, const T& val)
// Находит первый элемент в [first,last), равный val
{
    while (first!=last && *first != val) ++first;
    return first;
}
```

Вы полагаете, что этот цикл вполне тривиален? Мы так не думаем. На самом деле это минимальное, эффективное и непосредственное представление фундаментального алгоритма. Однако, пока мы не рассмотрим несколько примеров, это далеко не очевидно. Давайте напишем другую версию алгоритма и сравним с предыдущей.

```
template<typename In, typename T>
// Требует Input_iterator<In>()
// и Equality_comparable<Value_type<T>>() (раздел 19.3.3)
In find(In first, In last, const T& val)
// Находит первый элемент в [first,last), равный val
{
    for (In p = first; p!=last; ++p)
        if (*p == val) return p;
    return last;
}
```

Эти два определения логически эквивалентны, и действительно хороший компилятор сгенерирует для них обоих одинаковый код. Однако на практике многие компиляторы не настолько хороши, чтобы устранить излишнюю переменную (`p`) и перестроить код так, чтобы все проверки выполнялись в одном месте. Зачем это нужно? Частично затем, что стиль первой (предпочтительной) версии алгоритма `find()` стал очень популярным, и мы должны понимать его, чтобы читать чужие программы, а частично затем, что для небольших, часто используемых функций, работающих с большими объемами данных, большее значение имеет эффективность.

ПОПРОБУЙТЕ

Уверены ли вы, что эти два определения являются логически эквивалентными? Почему? Попробуйте привести аргументы в пользу их эквивалентности. Затем примените оба алгоритма к одному и тому же набору данных. Знаменитый специалист по компьютерным наукам Дональд Кнут (Donald Knuth) однажды сказал: "Я всего лишь доказал, что алгоритм является правильным, но не проверял его". Даже математические доказательства могут содержать ошибки. Для того чтобы убедиться в своей правоте, нужно иметь как доказательства, так и результаты тестирования.

21.2.1. Примеры использования обобщенных алгоритмов

Алгоритм `find()` является обобщенным. Это значит, что его можно применять к разным типам данных. Фактически его обобщенная природа носит двойственный характер.

- Алгоритм `find()` можно применять к любой последовательности в стиле библиотеки STL.
- Алгоритм `find()` можно применять к любому типу элементов.

Рассмотрим несколько примеров (если они покажутся вам сложными, обратитесь к диаграммам из раздела 20.4).

```
void f(vector<int>& v, int x) // Работает с векторами int
{
    auto p = find(v.begin(),v.end(),x);
    if (p != v.end()) { /* Мы нашли x */ }
    // ...
}
```

Здесь операции над итераторами, использованные в алгоритме `find()`, являются операциями над итераторами типа `vector<int>::iterator`; т.е. оператор ++ (в выражении `++first`) просто перемещает указатель на следующую ячейку памяти (где хранится следующий элемент вектора), а операция `*` (в выражении `*first`) разыменовывает этот указатель. Сравнение итераторов (в выражении `first!=last`) является сравнением указателей, а сравнение значений (в выражении `*first!=val`) — обычным сравнением целых чисел.

Попробуем применить алгоритм к объекту класса `list`.

```
void f(list<string>& v, string x) // Работа со списком строк
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* Мы нашли x */ }
    // ...
}
```

Здесь операции над итераторами, использованные в алгоритме `find()`, являются операциями над итераторами класса `list<string>::iterator`. Эти операторы имеют соответствующий смысл, так что логика их работы совпадает с логикой работы операторов из предыдущего примера (для класса `vector<int>`). В то же время они реализованы совершенно иначе; другими словами, оператор ++ (в выражении `++first`) просто следует за указателем, установленным на следующий узел списка, а оператор `*` (в выражении `*first`) находит значение в узле. Сравнение итераторов (в выражении `first!=last`) сводится к сравнению указателей типа `Link*`, а сравнение значений (в выражении `*first!=val`) означает сравнение строк с помощью оператора `!=` из класса `string`.

Итак, алгоритм `find()` чрезвычайно гибкий: если мы будем соблюдать простые правила при работе с итераторами, то сможем использовать алгоритм `find()` для поиска элементов в любой последовательности любого контейнера. Например, с помощью алгоритма `find()` мы можем искать символ в объекте класса `Document`, определенного в разделе 20.6:

```
void f(Document& v, char x) // Работает с классом Document
{
    Text_iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* Мы нашли x */ }
    // ...
}
```

Эта гибкость является отличительной особенностью алгоритмов из библиотеки STL и делает их более полезными, чем многие люди могут себе представить, когда впервые встречаются с ними.

21.3. Универсальный поиск: `find_if()`

Нам редко приходится искать какое-то конкретное значение. Куда чаще нас интересует значение, удовлетворяющее определенным критериям. Мы смогли бы выполнять намного более полезную операцию `find`, если бы имели возможность определять собственные критерии поиска. Например, мы могли бы найти число, превышающее `42`. Мы могли бы также сравнивать строки без учета регистра. А может быть, нам захотелось бы найти первое нечетное число или запись с адресом `"17 Cherry Tree Lane"`.

Стандартным алгоритмом поиска на основе критерия, заданного пользователем, является алгоритм `find_if()`.

```
template<typename In, typename Pred>
// Требует Input_iterator<In>()
// и Predicate<Pred,Value_type<In>>()
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

Очевидно (если сравнить исходные коды), что он похож на алгоритм `find()`, с тем отличием, что в нем используется условие `!pred(*first)`, а не `*first!=val`; иначе говоря, алгоритм останавливает поиск, как только предикат `pred()` оказывается истинным, а не когда обнаружен элемент с заданным значением.

Предикат (predicate) — это функция, возвращающая значение `true` или `false`. Очевидно, что алгоритм `find_if()` требует предиката, принимающего один аргумент, чтобы можно было записать выражение `pred(*first)`. Мы можем без труда написать предикат, проверяющий какое-то свойство

значения, например "содержит ли строка букву *x*?", "превышает ли число значение 42?" или "является ли число нечетным?". Например, мы можем найти первое нечетное число в целочисленном векторе следующим образом.

```
bool odd(int x) { return x%2; } // % — остаток от деления

void f(vector<int>& v)
{
    auto p = find_if(v.begin(), v.end(), odd);
    if (p!=v.end()) { /* Мы нашли нечетное число */ }
    // ...
}
```

При таком вызове алгоритм `find_if()` будет применять функцию `odd()` к каждому элементу, пока не найдет первое нечетное число. Обратите внимание, что при передаче функции в качестве аргумента круглые скобки после ее имени не добавляются, иначе это был бы вызов функции. Аналогично можно найти первый элемент списка, значение которого превышает 42.

```
bool larger_than_42(double x) { return x>42; }

void f(list<double>& v)
{
    auto p = find_if(v.begin(), v.end(), larger_than_42);
    if (p!=v.end()) { /* Мы нашли значение, большее 42 */ }
    // ...
}
```

Однако последний пример не вполне удовлетворительный. А что если мы после этого захотим найти элемент, который больше 41? Нам придется написать новую функцию. Хотите найти элемент, который больше 19? Пишите еще одну функцию. Должен быть более удобный способ!

Если мы хотим сравнивать элемент с произвольным значением `v`, то должны как-то сделать это значение неявным аргументом предиката алгоритма `find_if()`. Мы могли бы попробовать (выбрав в качестве удобного имени идентификатор `v_val`) решить задачу с помощью глобальной переменной.

```
double v_val; // Значение, с которым предикат larger_than_v()
              // сравнивает свой аргумент
bool larger_than_v(double x) { return x>v_val; }

void f(list<double>& v, int x)
{
    v_val = 31; // Устанавливаем v_val равной 31
    auto p = find_if(v.begin(), v.end(),larger_than_v);
    if (p!=v.end()) { /* Мы нашли значение, превышающее 31 */ }

    v_val = x;  // устанавливаем v_val равной x
    auto q = find_if(v.begin(), v.end(),larger_than_v);
```

```
    if (q!=v.end()) { /* Мы нашли значение, превышающее x */ }

    // ...
}
```

Какая гадость! Мы убеждены, что люди, написавшие такую программу, в конце концов получат по заслугам, но заранее сочувствуем пользователям и вообще любому, кто столкнется с таким кодом. Повторим: должен быть более удобный способ!

ПОПРОБУЙТЕ

Почему приведенное решение вызывает у нас такое отвращение? Назовите по крайней мере три возможности, которыми этот код приведет к непонятным ошибкам. Назовите три приложения, в которых такой код совершенно недопустим.

21.4. Функциональные объекты

Итак, мы хотим передавать предикат алгоритму **find_if()** и хотим, чтобы этот предикат сравнивал элементы со значением, которое мы зададим как его аргумент. В частности, мы хотим написать примерно такой код:

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* Мы нашли число, превышающее 31 */ }

    auto q = find_if(v.begin(), v.end(),Larger_than(x));
    if (q!=v.end()) { /* Мы нашли число, превышающее x */ }
    // ...
}
```

Очевидно, что объект **Larger_than** должен удовлетворять двум условиям.

- Его можно вызывать как предикат, например **pred(*first)**.
- Он может хранить значение, например **31** или **x**, передаваемое при вызове.

Для того чтобы выполнить эти условия, нам нужен функциональный объект, т.е. объект, который ведет себя как функция. Нам нужен объект, поскольку объекты могут хранить данные, например значение для сравнения. Рассмотрим следующий пример.

```
class Larger_than {
    int v;
public:
    Larger_than(int vv) : v(vv) { }              // Аргумент
    bool operator()(int x) const {return x>v;} // Сравнение
};
```

Следует отметить, что это определение представляет собой именно то, что мы требовали от предиката. Теперь осталось понять, как это работает. Написав выражение `Larger_than(31)`, мы (очевидно) создаем объект класса `Larger_than`, хранящий число 31 в члене `v`:

```
find_if(v.begin(),v.end(),Larger_than(31))
```

Здесь мы передаем объект `Larger_than(31)` алгоритму `find_if()` как параметр `pred`. Для каждого элемента `v` алгоритм `find_if()` осуществляет вызов

```
pred(*first)
```

Это приводит к выполнению оператора вызова функции, т.е. функции-члена `operator()` функционального объекта, с передачей ей аргумента `*first`. В результате происходит сравнение значения элемента, т.е. `*first`, с числом 31.

Мы видим, что вызов функции можно рассматривать как результат работы оператора `()`, аналогично любому другому оператору. Оператор `()` называют также *оператором вызова функции* (function call operator) или *оператором приложения* (application operator). Таким образом, скобки `()` в выражении `pred(*first)` представляют собой вызов `Larger_than::operator()`, точно так же, как квадратные скобки `[]` в выражении `v[i]` представляют собой вызов `vector::operator[]`.

21.4.1. Абстрактная точка зрения на функциональные объекты

Таким образом, мы имеем механизм, позволяющий "функции" хранить необходимые для работы данные. Очевидно, что функциональные объекты представляют собой универсальный, мощный и удобный механизм. Рассмотрим понятие функционального объекта подробнее.

```
class F {    // Абстрактный пример функционального объекта
    S s;     // Состояние
public:
    F(const S& ss) :s(ss) { /* Начальное состояние */ }
    T operator() (const S& ss) const
    {
        // Делает что-то с аргументом ss
        // Возвращает значение типа T (T зачастую
        // представляет собой void, bool или S)
    }

    // Получение состояния:
    const S& state() const { return s; }
    // Сброс состояния:
    void reset(const S& ss) { s = ss; }
};
```

Объект класса `F` хранит данные в своем члене `s`. При необходимости функциональный объект может иметь много данных-членов. Иногда вместо фразы "что-то хранит данные" говорят "нечто пребывает в состоянии". Создавая объект класса `F`, мы можем инициализировать это состояние. При необходимости мы можем прочитать это состояние. В классе `F` для считывания состояния предусмотрена операция `state()`, а для записи состояния — операция `reset()`. Однако при разработке функционального объекта мы свободны в выборе способа доступа к его состоянию. Разумеется, мы можем прямо или косвенно вызывать функциональный объект, используя обычную систему обозначений. При вызове функциональный объект `F` получает один аргумент, но мы можем определять функциональные объекты, получающие столько параметров, сколько потребуется.

Использование функциональных объектов является основным способом параметризации в STL. Мы используем функциональные объекты для того, чтобы указать алгоритму поиска, что именно мы ищем (см. раздел 21.3), для определения критериев сортировки (раздел 21.4.2), для указания арифметических операций в численных алгоритмах (раздел 21.5), для указания, какие значения мы считаем равными (раздел 21.8), а также для многого другого. Использование функциональных объектов — основной источник гибкости и универсальности алгоритмов.

Функциональные объекты, как правило, очень эффективны. В частности, передача по значению небольшого функционального объекта в качестве аргумента шаблонной функции обеспечивает оптимальную производительность. Причина проста, но удивительна для людей, хорошо знающих механизм передачи функций в качестве аргументов: обычно передача функционального объекта приводит к созданию значительно меньшего и более быстрого кода, чем при передаче функции как таковой! Это утверждение оказывается истинным, только если объект мал (например, если он содержит одно-два слова данных или вообще не хранит данные) или передается по ссылке, а также если оператор вызова функции невелик (например, простое сравнение с помощью оператора `<`) и определен как встраиваемая функция (например, когда его определение содержится в теле класса). Большинство примеров в этой главе — и в книге в целом — соответствует этому правилу. Основная причина высокой производительности небольших и простых функциональных объектов состоит в том, что они предоставляют компилятору объем информации о типе, достаточный для того, чтобы сгенерировать оптимальный код. Даже устаревшие компиляторы с несложными оптимизаторами могут сгенерировать простую машинную команду "больше" для сравнения в классе `Larger_than` вместо вызова функции. Вызов функции обычно выполняется в 10–50 раз дольше, чем простая операция сравнения. Кроме того, код для вызова функции в несколько раз больше, чем код простого сравнения.

21.4.2. Предикаты на членах класса

Как мы уже видели, стандартные алгоритмы хорошо работают с последовательностями элементов базовых типов, таких как **int** и **double**. Однако в некоторых предметных областях более широко используются контейнеры объектов пользовательских классов. Рассмотрим пример, играющий ключевую роль во многих предметных областях, — сортировка записей по нескольким критериям.

```
struct Record {
    string name;    // Для простоты — стандартная строка
    char addr[24]; // Стиль C для согласованности
                   // с базами данных
    // ...
};

vector<Record> vr;
```

Иногда мы хотим сортировать вектор **vr** по имени, а иногда — по адресам. Если мы не стремимся одновременно к элегантности и эффективности, наши методы ограничены практической целесообразностью. Мы можем написать следующий код:

```
// ...
sort(vr.begin(),vr.end(),Cmp_by_name()); // Сортировка по имени
// ...
sort(vr.begin(),vr.end(),Cmp_by_addr()); // Сортировка по адресу
// ...
```

Cmp_by_name — это функциональный объект, сравнивающий два объекта класса **Record** по членам **name**. Для того чтобы дать пользователю возможность задавать критерий сравнения, в стандартном алгоритме **sort** предусмотрен необязательный третий аргумент, указывающий критерий сортировки. Запись **Cmp_by_name()** создает объект типа **Cmp_by_name** для алгоритма **sort()**, который будет использовать его для сравнения объектов **Record**. Все выглядит отлично, и все, что мы должны сделать, — это определить классы **Cmp_by_name** и **Cmp_by_addr**.

```
// Разные сравнения объектов класса Record:

struct Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
        { return a.name < b.name; }
};

struct Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
        { return strncmp(a.addr, b.addr, 24) < 0; } // !!!
};
```

Класс `Cmp_by_name` совершенно очевиден. Оператор вызова функции `operator()` просто сравнивает строки `name`, используя оператор `<` стандартного класса `string`. Однако сравнение в классе `Cmp_by_addr` выглядит ужасно. Это объясняется тем, что мы выбрали неудачное представление адреса — в виде массива, состоящего из 24 символов (и не завершающегося нулевым символом). Мы сделали этот выбор частично для того, чтобы показать, как функциональный объект можно использовать для сокрытия некрасивого и уязвимого для ошибок кода, а частично — как вызов: уродливая, но важная с практической точки зрения задача, с которой не в состоянии справиться STL. Сравнение использует стандартную функцию `strncmp()`, которая сравнивает массивы символов фиксированной длины и возвращает отрицательное число, если вторая строка лексикографически больше, чем первая. Как только вам потребуется выполнить такое устаревшее сравнение, вспомните об этой функции (см., например, раздел Б.11.3).

21.4.3. Лямбда-выражения

Определение функционального объекта (или функции) в одном месте программы и использование его в другом может оказаться неудобным и утомительным. В частности, достаточно неприятно, когда такое действие оказывается очень просто определить, просто понять и оно никогда больше не потребуется в дальнейшем. В таком случае можно прибегнуть к лямбда-выражениям (раздел 15.3.3). Вероятно, наилучший способ — рассматривать лямбда-выражения как сокращенную запись для определения функционального объекта (класса с оператором `()`), а затем тут же использовать созданный объект. Например, мы можем написать следующий код.

```
// ...
sort(vr.begin(), vr.end(), // Сортировка по имени
    [ ] (const Record& a, const Record& b)
    { return a.name < b.name; }
);
// ...
sort(vr.begin(), vr.end(), // Сортировка по адресу
    [ ] (const Record& a, const Record& b)
    { return strncmp(a.addr, b.addr, 24) < 0; }
);
// ...
```

В этом случае мы можем задаться вопросом "А не дало ли бы применение именованных функциональных объектов более поддерживаемый и понятный код?" Ведь, возможно, `Cmp_by_name` и `Cmp_by_addr` имеют и другие применения?

Рассмотрим, однако, пример с алгоритмом `find_if()` из раздела 21.4. Здесь нам надо передать в качестве аргумента операцию, а эта операция нуждается в данных для своей работы:

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(), Larger_than(31));
    if (p!=v.end()) { /* Найдено значение > 31 */ }

    auto q = find_if(v.begin(), v.end(), Larger_than(x));
    if (q!=v.end()) { /* Найдено значение > x */ }

    // ...
}
```

В качестве эквивалентного альтернативного решения можно написать

```
void f(list<double>& v, int x)
{
    auto p = find_if(v.begin(), v.end(),
                     [](double a) { return a>31; });
    if (p!=v.end()) { /* Найдено значение > 31 */ }

    auto q = find_if(v.begin(), v.end(),
                     [&](double a) { return a>x; });
    if (q!=v.end()) { /* Найдено значение > x */ }
    // ...
}
```

Сравнение с локальной переменной **x** делает версию с лямбда-выражением более привлекательной.

21.5. Численные алгоритмы

Большинство стандартных алгоритмов из библиотеки STL связаны с обработкой данных: они их копируют, сортируют, выполняют в них поиск и т.д. В то же время некоторые из алгоритмов предназначены для вычислений. Они могут оказаться полезными как для решения конкретных задач, так и для демонстрации общих принципов реализации численных алгоритмов в библиотеке STL. Существуют всего четыре таких алгоритма.

Численные алгоритмы	
`x=accumulate(b,e,i)`	Суммирует последовательности; например, для последовательности {a,b,c,d} результат равен i+a+b+c+d. Тип результата **x** совпадает с типом начального значения `i`
`x=inner_product (b,e,b2,i)`	Перемножает пары значений из двух последовательностей и суммирует результаты; например, для последовательностей {a,b,c,d} и {e,f,g,h} результат равен i+a*e+b*f+c*g+d*h. Тип результата **x** совпадает с типом начального значения `i`
`r=partial_sum(b,e,r)`	Создает последовательность, состоящую из сумм первых *n* элементов заданной последовательности; например, для последовательности {a,b,c,d} результат равен {a, a+b, a+b+c, a+b+c+d}

Численные алгоритмы	
`r=adjacent_difference (b,e,b2,r)`	Создает последовательность разностей между элементами заданной последовательности; например, для последовательности {a,b,c,d} результат равен {a,b–a,c–b,d–c}

Эти алгоритмы определены в заголовочном файле **<numeric>**. Мы опишем первые два из них, а остальные при необходимости читатели могут изучить самостоятельно.

21.5.1. Алгоритм accumulate()

Простейшим и наиболее полезным численным алгоритмом является алгоритм **accumulate()**. В простейшем варианте он суммирует значения, принадлежащие последовательности.

```
template<typename In, typename T>
// Требует Input_iterator<In>() и Number<T>()
T accumulate(In first, In last, T init)
{
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Получив начальное значение **init**, он просто добавляет к нему каждое значение из последовательности [**first**,**last**) и возвращает сумму. Переменную **init**, в которой накапливается сумма, часто называют *аккумулятором* (accumulator). Рассмотрим пример.

```
int a[] = { 1, 2, 3, 4, 5 };
cout << accumulate(a, a+sizeof(a)/sizeof(int), 0);
```

Этот фрагмент кода выводит на экран число 15, т.е. 0+1+2+3+4+5 (0 является начальным значением). Очевидно, что алгоритм **accumulate()** можно использовать для всех видов последовательностей.

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0);
    int sum2 = accumulate(p,p+n,0);
}
```

Тип результата (суммы) совпадает с типом переменной, которую алгоритм **accumulate()** использует в качестве аккумулятора. Это обеспечивает высокую степень гибкости которая может играть важную роль. Рассмотрим пример.

```
void f(int* p, int n)
{
    // Суммирование целых чисел типа int:
    int s1 = accumulate(p, p+n, 0);        // В int
    long s1 = accumulate(p, p+n, long(0)); // В long
    double s2 = accumulate(p, p+n, 0.0);   // В double
}
```

На некоторых компьютерах переменная типа **long** имеет больше значащих цифр, чем переменная типа **int**. Переменная типа **double** может представить большие (и меньшие) числа по сравнению с переменной типа **int**, но, возможно, с меньшей точностью. В главе 24 мы еще вернемся к вопросу о диапазоне и точности в вычислениях.

Использование в качестве аккумулятора инициализирующей переменной представляет собой распространенную идиому, позволяющую задать тип аккумулятора.

```
void f(vector<double>& vd, int* p, int n)
{
    double s1 = 0;
    s1 = accumulate(vd.begin(), vd.end(), s1);
    int s2 = accumulate(vd.begin(), vd.end(), s2); // Ой
    float s3 = 0;
    accumulate(vd.begin(), vd.end(), s3);          // Ой
}
```

Не забывайте инициализировать аккумулятор и присваивать результат работы алгоритма **accumulate()** какой-нибудь переменной. В данном примере в качестве инициализатора использовалась переменная **s2**, которая до вызова алгоритма сама еще не получила начальное значение; результат такого вызова будет непредсказуем. Мы передали переменную **s3** алгоритму **accumulate()** (по значению; см. раздел 8.5.3), но результат ничему не присвоили; такая компиляция представляет собой простую трату времени.

21.5.2. Обобщение алгоритма accumulate()

Таким образом, базовый алгоритм **accumulate()** с тремя аргументами выполняет суммирование. Однако существует много других полезных операций, которые можно выполнять над последовательностями — например, умножение и вычитание, — поэтому в STL предусмотрена версия алгоритма **accumulate()** с четырьмя аргументами, позволяющая указать используемую операцию.

```
template<typename In, typename T, typename BinOp>
// Требует Input_iterator<In>(), Number<T>() и
// Binary_operator<BinOp,Value_type<In>,T>()
T accumulate(In first, In last, T init, BinOp op)
{
```

```
    while (first!=last) {
        init = op(init, *first);
        ++first;
    }
    return init;
}
```

Здесь можно использовать любую бинарную операцию, получающую два аргумента с типом аккумулятора. Рассмотрим пример.

```
array<double,4> a = { 1.1, 2.2, 3.3, 4.4 };
cout << accumulate(a.begin(),a.end(),1.0,multiplies<double>());
```

Этот фрагмент кода выводит число 35.1384, т.е. 1.0*1.1*2.2*3.3*4.4 (1.0 — начальное значение). Бинарный оператор `multiplies<double>()`, передаваемый в качестве аргумента, представляет собой функциональный объект стандартной библиотеки, выполняющий умножение; функциональный объект `multiplies<double>` перемножает числа типа `double`, функциональный объект `multiplies<int>` перемножает числа типа `int` и т.д. Существуют и другие бинарные функциональные объекты: `plus` (сложение), `minus` (вычитание), `divides` и `modulus` (деление и вычисление остатка от деления). Все они определены в заголовочном файле `<functional>` (раздел Б.6.2).

Обратите внимание на то, что для умножения чисел с плавающей точкой естественным начальным значением является число `1.0`.

Как и в примере с алгоритмом `sort()` (см. раздел 21.4.2), нас часто интересуют данные, хранящиеся в объектах классов, а не обычные данные встроенных типов. Например, мы могли бы вычислить общую стоимость товаров, зная стоимость их единицы и общее количество.

```
struct Record {
    double unit_price;
    int units; // Количество проданных единиц
    // ...
};
```

Мы можем поручить оператору алгоритма `accumulate` извлекать данные `units` из соответствующего элемента класса `Record`, умножать на значение цены и прибавлять к аккумулятору.

```
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units; // Вычисляет стоимость
                                       // и накапливает итог
}

void f(const vector<Record>& vr)
{
    double total = accumulate(vr.begin(),vr.end(),0.0,price);
    // ...
}
```

Мы "поленились" и использовали для вычисления стоимости функцию, а не функциональный объект — просто чтобы показать, что так тоже можно делать. Мы рекомендуем использовать функциональные объекты в следующих ситуациях:

- если между вызовами необходимо сохранять данные,
- если они настолько короткие, что их можно сделать встраиваемыми (по крайней мере, для некоторых примитивных операций).

В приведенном примере мы могли бы использовать функциональный объект, руководствуясь вторым пунктом этого списка.

ПОПРОБУЙТЕ

Определите класс **vector<Record>**, проинициализируйте его четырьмя записями по своему выбору и вычислите общую стоимость, используя приведенные выше функции.

21.5.3. Алгоритм inner_product()

Возьмем два вектора, попарно перемножим их компоненты и сложим полученные произведения. Результат этих вычислений называется *скалярным произведением* (inner product) двух векторов и является широко используемой операцией во многих областях (например, в физике и линейной алгебре; см. раздел 24.6). Если словам вы предпочитаете программу, то взгляните на версию этого алгоритма из библиотеки STL.

```
template<typename In, typename In2, typename T>
//Требует Input_iterator<In> и Input_iterator<In2>
// и Number<T> (раздел 19.3.3)
T inner_product(In first, In last, In2 first2, T init)
// Примечание: вычисляет скалярное произведение двух векторов
{
    while(first!=last) {
        init = init + (*first)*(*first2); // Умножение пар
        ++first;
        ++first2;
    }
    return init;
}
```

Эта версия алгоритма обобщает понятие скалярного произведения для любого вида последовательностей с любым типом элементов. Рассмотрим в качестве примера биржевой индекс. Он вычисляется путем присваивания компаниям неких весов. Например, индекс Доу–Джонса Alcoa на момент написания книги составлял 2.4808. Для того чтобы определить текущее значение индекса, умножаем цену акции каждой компании на ее вес

и складываем полученные результаты. Очевидно, что такой индекс представляет собой скалярное произведение цен и весов.

```
// Вычисление индекса Доу-Джонса
vector<double> dow_price = { // Цена акции каждой компании
    81.86, 34.69, 54.45,
    // ...
};

list<double> dow_weight = {  // Вес каждой компании в индексе
    5.8549, 2.4808, 3.8940,
    // ...
};

double dji_index = inner_product(// Умножаем пары (weight,value) и суммируем
        dow_price.begin(), dow_price.end(),
        dow_weight.begin(), 0.0);

cout << "Значение DJI равно " << dji_index << '\n';
```

Обратите внимание на то, что алгоритм `inner_product()` получает две последовательности. В то же время он получает только четыре аргумента: у второй последовательности задается только начало. Предполагается, что вторая последовательность содержит не меньше элементов, чем первая. В противном случае мы получим ошибку времени выполнения. В алгоритме `inner_product()` вторая последовательность вполне может содержать больше элементов, чем первая; лишние элементы просто не будут использоваться.

Две последовательности не обязательно должны иметь одинаковый тип или содержать элементы одинаковых типов. Для того чтобы проиллюстрировать это утверждение, мы записали цены в объект класса `vector`, а веса — в объект класса `list`.

21.5.4. Обобщение алгоритма `inner_product()`

Алгоритм `inner_product()` можно обобщить так же, как и алгоритм `accumulate()`. Однако в отличие от предыдущего обобщения алгоритму `inner_product()` нужны два дополнительных аргумента: первый — для обновления аккумулятора новым значением, точно так же как в алгоритме `accumulate()`, а второй — для действия с парами значений.

```
template<typename In, typename In2, typename T,
         typename BinOp, typename BinOp2>
// Требует Input_iterator<In>, Input_iterator<In2>,
// Number<T>, Binary_operation<BinOp,T,Value_type<In>(),
// и Binary_operation<BinOp2,T, Value_type<In2>()
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while(first!=last) {
```

```
        init = op(init, op2(*first, *first2));
        ++first;
        ++first2;
    }
    return init;
}
```

В разделе 21.6.3 мы еще вернемся к примеру с индексом Доу–Джонса и используем обобщенную версию алгоритма `inner_product()` как часть более элегантного решения поставленной задачи.

21.6. Ассоциативные контейнеры

После класса `vector` вторым по частоте использования, вероятно, является стандартный контейнер `map`, представляющий собой упорядоченную последовательность пар (ключ, значение) и позволяющий находить значение по ключу; например, элемент `my_phone_book["Nicholas"]` может быть телефонным номером Николаса. Единственным потенциально достойным конкурентом класса `map` по популярности является класс `unordered_map` (см. раздел 21.6.4). Структуры данных, подобные контейнерам `map` и `unordered_map`, известны под разными названиями, например *ассоциативные массивы* (associative arrays), *хеш-таблицы* (hash tables) и *красно-черные деревья* (red-black trees). Популярные и полезные концепции всегда имеют много названий. Мы будем называть их *ассоциативными контейнерами* (associative containers).

Стандартная библиотека предоставляет восемь ассоциативных контейнеров.

Ассоциативные контейнеры	
`map`	Отображение — упорядоченный контейнер пар (ключ, значение)
`set`	Множество — упорядоченный контейнер ключей
`unordered_map`	Неупорядоченное отображение — контейнер пар (ключ, значение)
`unordered_set`	Неупорядоченное множество — контейнер ключей
`multimap`	Мультиотображение — контейнер `map`, в котором ключ может встречаться несколько раз
`multiset`	Мультимножество — контейнер `set`, в котором ключ может встречаться несколько раз
`unordered_multimap`	Неупорядоченное мультиотображение — контейнер `unordered_map`, в котором ключ может встречаться несколько раз
`unordered_multiset`	Неупорядоченное мультимножество — контейнер `unordered_set`, в котором ключ может встречаться несколько раз

Эти контейнеры определены в заголовочных файлах **<map>**, **<set>**, **<unordered_map>** и **<unordered_set>**.

21.6.1. Контейнер map

Рассмотрим концептуально простую задачу: создадим список количеств вхождений слов в текст. Очевидным решением является поддержка списка слов вместе с количеством их вхождений в текст. Считывая новое слово, мы проверяем, не появлялось ли оно ранее. Если нет, вставляем его в список и связываем с ним число 1; если да — увеличиваем связанный с ним счетчик на единицу. Для этого можно было бы использовать объект типа **list** или **vector**, но тогда мы должны были бы выполнять поиск каждого считанного слова. Такое решение было бы слишком медленным. Класс отображения **map** хранит свои ключи так, чтобы было легко обнаружить искомый ключ, если он там есть. В этом случае поиск становится тривиальной задачей.

```
int main()
{
    map<string,int> words; // Поддержка пар (слово, частота)

    for (string s; cin>>s; )
        ++words[s];        // words индексируется строками

    for (const auto& p : words)
        cout << p. first << ": " << p.second << '\n';
}
```

Самой интересной частью этой программы является выражение **++words[s]**. Как видно из первой строки функции **main()**, переменная **words** — это объект класса **map**, хранящего пары (**string**, **int**); иными словами, контейнер **words** отображает строки **string** на целые числа **int**. Иначе говоря, по заданной строке типа **string** контейнер **words** предоставляет нам доступ к соответствующему числу типа **int**. Так что, когда мы индексируем контейнер **words** строкой **s** (содержащей слово, считанное из потока ввода), элемент **words[s]** представляет собой ссылку на переменную типа **int**, соответствующую строке **s**. Рассмотрим конкретный пример.

```
words["sultan"]
```

Если строка **"sultan"** еще не встречалась, то она вставляется в контейнер **words** вместе со значением, заданным по умолчанию для типа **int**, т.е. **0**. Теперь контейнер **words** содержит элемент (**"sultan"**,**0**). Таким образом, если строка **"sultan"** ранее в контейнер не вносилась, то выражение **++words["sultan"]** свяжет со строкой **"sultan"** значение **1**. Точнее говоря, объект класса **map** выяснит, что строки **"sultan"** в нем нет, вставит пару **("sultan",0)**, а затем оператор **++** увеличит это значение на единицу, в итоге оно станет равным **1**.

Посмотрим на текст программы еще раз: выражение `++words[s]` получает слово из потока ввода и увеличивает соответствующее ему целое значение на единицу. При первом вводе каждое слово получает значение `1`. Теперь смысл цикла становится понятным.

```
for (string s; cin>>s; )
    ++words[s];        // words индексируется строками
```

Он считывает каждое слово (отделенное пробельным символом) из потока ввода и вычисляет количество его вхождений в контейнер. Теперь нам достаточно просто вывести результат. Обойти контейнер `map` можно точно так же, как и любой другой контейнер из STL. Элементы контейнера `map<string, int>` имеют тип `pair<string,int>`. Первый член объекта класса `pair` имеет имя `first`, второй — `second`. Цикл вывода выглядит следующим образом:

```
for (const auto& p : words)
    cout << p. first << ": " << p.second << '\n';
```

В качестве теста мы ввели в программу вступительный текст из первого издания книги *The C++ Programming Language*.

> C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.

Результат работы программы приведен ниже.

```
C: 1
C++: 3
C,: 1
Except: 1
In: 1
a: 2
addition: 1
and: 1
by: 1
defining: 1
designed: 1
details,: 1
efficient: 1
enjoyable: 1
facilities: 2
flexible: 1
for: 3
general: 1
is: 2
language: 1
language.: 1
make: 1
minor: 1
```

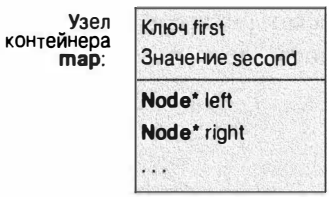
```
more: 1
new: 1
of: 1
programmer.: 1
programming: 3
provided: 1
provides: 1
purpose: 1
serious: 1
superset: 1
the: 3
to: 2
types.: 1
```

Если не хотите различать верхний и нижний регистры букв или учитывать знаки пунктуации, то можно решить и эту задачу (см. упр. 13).

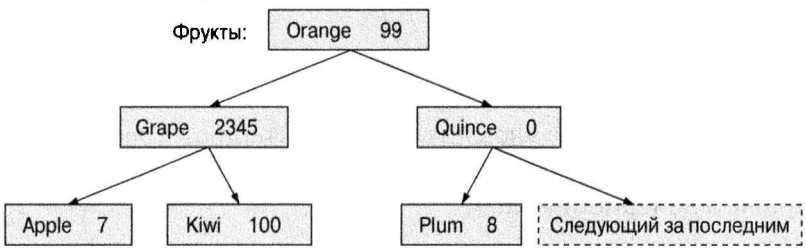
21.6.2. Обзор контейнера map

Так что же собой представляет контейнер **map**? Существует много способов реализации класса **map**, но, как правило, в STL он реализован на основе сбалансированных бинарных деревьев, точнее говоря — на основе красно-черных деревьев. Мы не будем вдаваться в детали, но, поскольку теперь вы знаете это название, не составит труда найти всю необходимую информацию в книгах или в веб.

Дерево состоит из узлов (подобно тому как из узлов состоит список; см. раздел 20.4). В объекте класса **Node** хранятся ключ, соответствующее ему значение и указатели на два узла-наследника.



Вот как может выглядеть объект класса **map<Fruit,int>** в памяти компьютера, после того как мы вставили в него пары (Kiwi,100), (Quince,0), (Plum,8), (Apple,7), (Grape,2345) и (Orange,99).



Поскольку ключ хранится в члене класса `Node` с именем `first`, основное правило организации бинарного дерева поиска имеет следующий вид:

```
left->first < first && first < right->first
```

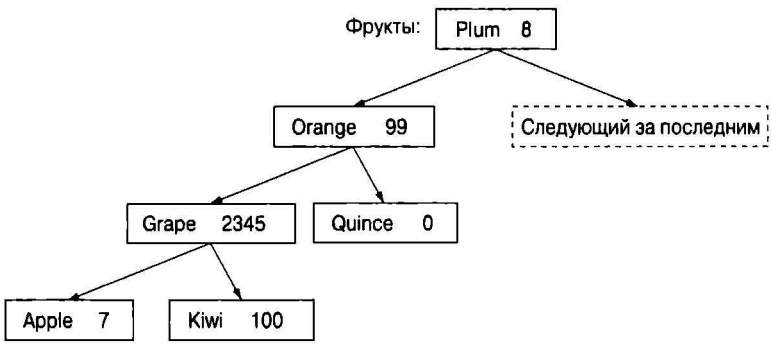
Иначе говоря, для каждого узла выполняются два условия:

- ключ его левого дочернего узла меньше ключа узла;
- ключ узла меньше, чем ключ его правого дочернего узла.

Можете убедиться, что эти условия выполняются для каждого узла дерева. Это позволяет нам выполнять поиск вниз по дереву, начиная с корня. Забавно, что в литературе по компьютерным наукам деревья растут вниз. Корневым узлом является узел, содержащий пару (Orange, 99). Мы просто перемещаемся по дереву вниз, пока не находим подходящее место. Дерево называется *сбалансированным* (balanced), если (как в приведенном выше примере) все одинаково удаленные от корня поддеревья содержат примерно одинаковое количество узлов. В сбалансированном дереве среднее количество узлов, которые мы должны пройти, пока не достигнем заданного узла, минимально.

В узле могут также храниться дополнительные данные, которые контейнер может использовать для поддержки сбалансированности. Дерево считается сбалансированным, если каждый узел имеет примерно одинаковое количество наследников как слева, так и справа. Если дерево, состоящее из N узлов, сбалансировано, то для обнаружения узла необходимо просмотреть не больше $\log_2 N$ узлов. Это намного лучше, чем $N/2$ узлов в среднем, которые мы должны были бы просмотреть, если бы ключи хранились в списке, а поиск выполнялся с его начала (при линейном поиске в худшем случае нам пришлось бы просмотреть все N узлов). (См. также раздел 21.6.4.)

Вот пример того, как может выглядеть несбалансированное дерево.



Это дерево по-прежнему удовлетворяет критерию, требующему, чтобы ключ каждого узла был больше ключа левого дочернего узла и меньше ключа правого дочернего узла.

```
left->first < first && first < right->first
```

И все же это дерево является несбалансированным, поэтому нам придется совершить три перехода, чтобы найти узлы Apple и Kiwi, вместо двух переходов в сбалансированном дереве. Для деревьев, содержащих большое количество узлов, эта разница может оказаться существенной, поэтому для реализации контейнеров **map** используются именно сбалансированные деревья.

Разбираться в принципах организации деревьев, используемых для реализации контейнера **map**, необязательно. Достаточно того, что профессионалы как минимум понимают принципы их работы. Все, что нам нужно, — это интерфейс класса **map**, предоставляемый стандартной библиотекой. Вот как выглядит его несколько упрощенная версия.

```
template<typename Key, typename Value, typename Cmp = less<Key>>
// Требует Binary_operation<Cmp,Value>() (раздел 19.3.3)
class map {
    // ...
    using value_type = pair<Key,Value>; // map работает с парами (Key,Value)
    using iterator = sometype1;         // Аналог указателя
    using const_iterator = sometype2;   // на узел дерева

    iterator begin();           // Указатель на первый элемент
    iterator end();             // Указатель за последний элемент

    Value& operator[](const Key& k); // Индексация значением k

    iterator find(const Key& k);     // Есть ли ключ k в map?

    void erase(iterator p);     // Удаление элемента с помощью p
                                // Вставка пары (key,value):
    pair<iterator, bool> insert(const value_type&);
    // ...
};
```

Реальную версию контейнера **map** можно найти в заголовочном файле **<map>**. Можно представить себе итератор в виде указателя на узел **Node***, но нельзя полагаться на то, что библиотека вашего компилятора использует определенный тип для реализации итератора.

Сходство рассмотренного интерфейса с интерфейсами классов **vector** и **list** (см. разделы 20.5 и B.4) очевидно. Основное отличие заключается в том, что при итерировании элементами являются пары типа **pair <Key,Value>**. Этот тип является еще одним очень полезным типом в STL.

```
template<typename T1, typename T2>
struct pair { // Упрощенная версия std::pair
    using first_type = T1;
    using second_type = T2;

    T1 first;
    T2 second;
    // ...
};
```

```
template<typename T1, typename T2>
pair<T1,T2> make_pair(T1 x, T2 y)
{
    return {x,y};
}
```

Здесь мы скопировали полное определение класса **pair** и его полезную вспомогательную функцию **make_pair()** из стандарта.

Заметим, что при перемещении по контейнеру **map** элементы перебираются в порядке, определенном ключом. Например, если мы перемещаемся по контейнеру, описанному в примере с фруктами, то получаем следующий порядок обхода:

```
(Apple,7) (Grape,2345) (Kiwi,100) (Orange,99) (Plum,8) (Quince,0)
```

Порядок вставки узлов значения не имеет.

Операция **insert()** имеет странное возвращаемое значение, которое в простых программах, как правило, игнорируется. Это пара, состоящая из итератора, указывающего на пару (ключ, значение), и переменной типа **bool**, принимающей значение **true**, если данная пара (ключ, значение) была вставлена с помощью вызова функции **insert()**. Если ключ уже был в контейнере, то вставка игнорируется, а переменная типа **bool** принимает значение **false**.

Заметим, что упорядочение элементов, используемое отображением, можно определить с помощью третьего аргумента (предикат **Cmp** в объявлении класса **map**). Рассмотрим пример.

```
map<string, double, No_case> m;
```

Предикат **No_case** определяет сравнение символов без учета регистра (см. раздел 21.8). По умолчанию порядок обхода определяется предикатом **less<Key>**, т.е. отношением "меньше".

21.6.3. Еще один пример использования map

Для того чтобы оценить полезность отображения **map**, вернемся к примеру с индексом Доу–Джонса из раздела 21.5.3. Описанный там код работает правильно, только если все веса записаны в объекте класса **vector** в тех же позициях, что и соответствующие имена. Это требование носит неявный характер и легко может стать источником малопонятных ошибок. Существует много способов решения этой проблемы, но наиболее привлекательным является хранение всех весов вместе с аббревиатурой названия компании — ("AA",2.4808). Аналогично аббревиатуру компании можно хранить вместе с ценой ее акции, например ("AA",34.69). В заключение для людей, редко сталкивающихся с фондовым рынком США, мы можем записывать аббревиатуру вместе с полным названием компании, например

("AA","Alcoa Inc."); иначе говоря, мы можем поддерживать три отображения соответствующих значений.

Сначала создадим отображение, содержащее пары (аббревиатура, цена).

```
map<string,double> dow_price = {  // Индекс Доу-Джонса
        //  (аббревиатура, цена);
        //текущие котировки см. на www.djindexes.com
    {"MMM",81.86},
    {"AA",34.69},
    {"MO",54.45},
    // ...
};
```

Отображение, содержащее пары (аббревиатура, вес), объявляется следующим образом.

```
map<string,double> dow_weight = { // (аббревиатура, вес)
    {"MMM", 5.8549},
    {"AA",2.4808},
    {"MO",3.8940},
    // ...
};
```

Наконец создадим отображение с парами (аббревиатура, название).

```
map<string,string> dow_name = { // (аббревиатура, название)
    {"MMM","3M Co."},
    {"AA","Alcoa Inc."},
    {"MO","Altria Group Inc."},
    // ...
};
```

С помощью этих отображений легко извлечь любую информацию. Рассмотрим пример.

```
double alcoa_price = dow_price ["AAA"]; // Считываем значения
double boeing_price = dow_price ["BA"]; // из отображения

// Находим запись в отображении:
if (dow_price.find("INTC") != dow_price.end())
    cout << "Intel найден\n";
```

Перемещаться по ассоциативному массиву легко. Мы просто должны помнить, что ключ называется `first`, а значение — `second`.

```
// Вывод цен акций каждой компании в индексе Доу-Джонса
for (const auto& p : dow_price) {
    const string& symbol = p.first;      // Аббревиатура
    cout << symbol << '\t'
         << p.second << '\t'
         << dow_name[symbol] << '\n';
}
```

Используя отображение, мы можем даже выполнить некоторые вычисления. В частности, можно вычислить индекс, как мы делали это в разделе 21.5.3. Мы должны извлечь цены акций и веса из соответствующих отображений и перемножить их. Можно без труда написать функцию, выполняющую эти вычисления с любыми двумя отображениями **map<string,double>**.

```
double weighted_value(
    const pair<string,double>& a,
    const pair<string,double>& b
    )       // Извлечение значений и перемножение
    {
        return a.second * b.second;
    }
```

Теперь просто подставим эту функцию в обобщенную версию алгоритма **inner_product()** и получим значение индекса.

```
double dji_index =
    inner_product(
        dow_price.begin(), dow_price.end(), // Все компании
        dow_weight.begin(),  // Их веса
        0.0,                 // Начальное значение
        plus<double>(),      // Обычное сложение
        weighted_value);     // Перемножение значений
```

Почему целесообразно хранить такие данные в отображениях, а не в векторах? Мы использовали класс **map**, чтобы связь между разными значениями стала явной. Это одна из причин. Кроме того, контейнер **map** хранит элементы в порядке, определяемом их ключами. Так, при обходе контейнера **dow** мы выводили символы в алфавитном порядке; если бы мы использовали класс **vector**, то были бы должны его сортировать. Чаще всего класс **map** используют просто потому, что в задаче требуется поиск значений по их ключам. Для больших последовательностей поиск элементов с помощью алгоритма **find()** оказывается намного медленнее, чем поиск в упорядоченной структуре, такой как отображение **map**.

ПОПРОБУЙТЕ

Доведите приведенный пример до работоспособного состояния. Затем по своему выбору добавьте несколько компаний и задайте их веса.

21.6.4. Контейнер unordered_map

Для того чтобы найти элемент в контейнере **vector**, алгоритм **find()** должен проверить все элементы, начиная с первого и заканчивая искомым или последним элементом вектора. Среднее

время работы этого поиска пропорционально длине вектора N; в таком случае говорят, что алгоритм имеет сложность $O(N)$.

Для того чтобы найти элемент в контейнере **map**, оператор индексирования должен проверить все элементы на пути от корня дерева и до искомого значения или до листа дерева. Среднее время этого поиска пропорционально глубине дерева. Максимальная глубина сбалансированного бинарного дерева, содержащего N элементов, равна $\log_2 N$, так что сложность поиска равна $O(\log_2 N)$, т.е. пропорциональна величине $\log_2 N$,. Это намного лучше, чем $O(N)$.

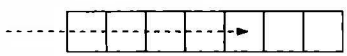
N	16	128	1024	16384	1048576	1073741824
$\log_2 N$	4	7	10	14	20	30

Реальная сложность поиска зависит от того, насколько быстро нам удастся найти искомые значения и какие затраты связаны с выполнением операции сравнения и итераций. Обычно следование по указателям (при поиске в контейнере **map**) несколько сложнее, чем инкремент указателя (при поиске в контейнере **vector** с помощью алгоритма **find()**).

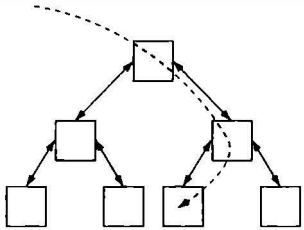
Для некоторых типов, особенно для целых чисел и символьных строк, можно достичь еще более высокой скорости поиска, чем при поиске по дереву контейнера **map**. Не вдаваясь в подробности, скажем, что идея заключается в том, что для данного ключа мы можем вычислить индекс в контейнере **vector**. Этот индекс называется *хеш-значением* (hash value), а контейнер, в котором используется этот метод, — *хеш-таблицей* (hash table). Количество возможных ключей намного больше, чем количество ячеек в хеш-таблице. Например, хеш-функция часто используется для того, чтобы отобразить миллиарды возможных строк в индексы вектора, состоящего из тысячи элементов. Такая задача достаточно сложна, но разрешима. Основное преимущество хеш-таблицы заключается в том, что среднее время поиска в ней является (почти) постоянным и не зависит от количества элементов в таблице, т.е. имеет порядок $O(1)$. Очевидно, что это значительное преимущество для больших массивов данных, например для 500 тысяч веб-адресов. Более подробную информацию о хеш-поиске читатели могут найти в документации по контейнеру **unordered_map** (доступной в сети веб) или в любом учебнике по структурам данных и алгоритмам (ищите в предметном указателе *хеш-таблицы* или *хеширование*).

Проиллюстрируем графически поиск в (неупорядоченном) векторе, сбалансированном бинарном дереве и хеш-таблице.

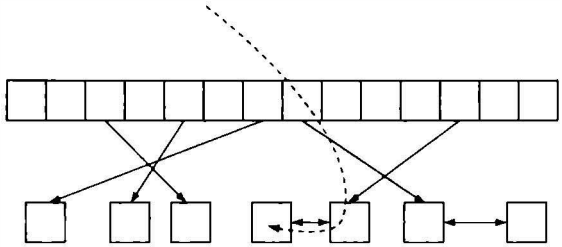
- Поиск в неупорядоченном контейнере **vector**.



- Поиск в отображении **map** (сбалансированном бинарном дереве).



- Поиск в отображении **unordered_map** (хеш-таблица).



Контейнер **unordered_map** из библиотеки STL реализуется с помощью хеш-таблицы, контейнер **map** — на основе сбалансированного бинарного дерева, а контейнер **vector** — в виде массива. Полезность библиотеки STL частично объясняется тем, что она позволила объединить в одно целое разные способы хранения данных и доступа к ним, с одной стороны, и алгоритмы — с другой. Эмпирическое правило гласит следующее.

- Используйте контейнер **vector**, если у вас нет веских оснований не делать этого.

- Используйте контейнер **map**, если вам необходимо выполнять поиск по значению (и если тип ключа позволяет эффективно выполнять операцию "меньше").
- Используйте контейнер **unordered_map**, если вам необходимо часто выполнять поиск в большом ассоциативном массиве и вам не нужен упорядоченный обход (и если тип вашего ключа допускает эффективное использование хеш-функций).

Мы не будем подробно описывать контейнер **unordered_map**. Его можно использовать с ключом типа **string** или **int** точно так же, как и контейнер **map**, с тем отличием, что при обходе элементов они не будут упорядочены. Например, мы могли бы переписать фрагмент кода для вычисления индекса- Доу–Джонса из раздела 21.6.3 следующим образом:

```
unordered_map<string,double> dow_price;

for (const auto& p : dow_price) {
    const string& symbol = p.first; // Аббревиатура
    cout << symbol << '\t'
```

```
        << p.second << '\t'
        << dow_name[symbol] << '\n';
}
```

Теперь поиск в контейнере **dow** можно выполнять быстрее. Однако это ускорение может оказаться незаметным, поскольку в данный индекс включены только около тридцати компаний. Если бы мы учли цены акций всех компаний, котирующихся на нью-йоркской фондовой бирже, то сразу почувствовали бы разницу в производительности работы программы. Отметим пока лишь логическое отличие: данные в этом случае выводятся не в алфавитном порядке.

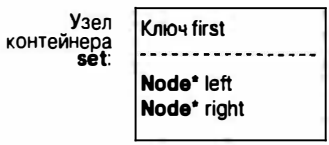
Неупорядоченные ассоциативные массивы в стандарте языка C++ являются новшеством и еще не стали полноправным его элементом, поскольку описаны в техническом отчете Комиссии по стандартизации языка C++ (Technical Report), а не в тексте самого стандарта. Тем не менее они широко распространены, а там, где их нет, часто можно обнаружить их аналоги, например, что-нибудь вроде класса **hash_map**.

ПОПРОБУЙТЕ

Напишите программу с директивой **#include <unordered_map>**. Если она не работает, значит, класс **unordered_map** не включен в вашу реализацию языка C++. Если вам действительно нужен контейнер **unordered_map**, можете загрузить одну из его доступных реализаций из веба (см., например, сайт **www.boost.org**).

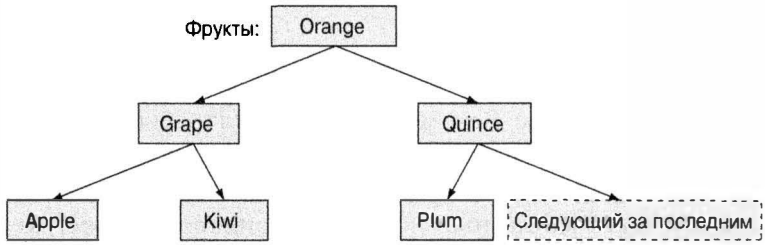
21.6.5. Контейнер set

Контейнер **set** можно рассматривать как отображение, в котором значения не имеют значения, или как отображение без значений. Визуализировать узел множества **set** можно следующим образом.



Контейнер **set**, в котором перечислены фрукты (см. раздел 21.6.2), можно представить следующим образом:

Чем полезны контейнеры **set**? Оказывается, существует много проблем, при решении которых следует помнить, встречалось ли нам уже какое-то значение. Один из примеров — перечисление имеющихся фруктов (независимо от цены); второй пример — составление словарей. Другой способ использования этого контейнера — множество "записей", элементы кото-



рого являются объектами, потенциально содержащими много информации, и при этом роль ключа играет один из их членов. Рассмотрим пример.

```
struct Fruit {
    string name;
    int count;
    double unit_price;
    Date last_sale_date;
    // ...
};

struct Fruit_order {
    bool operator()(const Fruit& a, const Fruit& b) const
    {
        return a.name<b.name;
    }
};

set<Fruit, Fruit_order> inventory; // Для сравнения объектов
     // Fruit используется функциональный объект Fruit_Order
```

Здесь мы снова видим, как функциональный объект значительно расширяет спектр задач, которые можно решить с помощью библиотеки STL.

Поскольку контейнер **set** не имеет значений, он не поддерживает операцию индексирования (**operator[]()**). Следовательно, вместо нее мы должны использовать такие "операции над списками", как **insert()** и **erase()**. К сожалению, контейнеры **map** и **set** не поддерживают функцию **push_back()** по очевидной причине: место вставки нового элемента определяет контейнер **set**, а не программист. Вместо этого следует использовать функцию **insert()**:

```
inventory.insert(Fruit{"quince",5});
inventory.insert(Fruit{"apple", 200, 0.37});
```

Одно из преимуществ контейнера **set** над контейнером **map** заключается в том, что мы можем непосредственно использовать значение, полученное посредством итератора. Поскольку в контейнере **set** нет пар (ключ, значение), как в контейнере **map** (см. раздел 21.6.3), оператор разыменования возвращает значение элемента:

```
for (auto p = inventory.begin(); p!=inventory.end(); ++p)
    cout << *p << '\n';
```

Разумеется, этот фрагмент работает, только если вы определили оператор << для класса `Fruit`.

21.7. Копирование

В разделе 21.2 мы назвали функцию `find()` "простейшим полезным алгоритмом". Естественно, эту точку зрения можно обосновать. Многие простые алгоритмы являются полезными, даже тривиальные. Зачем писать новый код, если можно использовать тот, который кто-то уже написал и отладил? С точки зрения простоты и полезности алгоритм `copy()` даст алгоритму `find()` фору. В библиотеке STL есть три варианта алгоритма `copy()`.

Операции копирования	
`copy(b,e,b2)`	Копирует последовательность [b,e) в последовательность [b2,b2+(e–b))
`unique_copy(b,e,b2)`	Копирует последовательность [b,e) в последовательность [b2,b2+(e–b)), отбрасывая смежные копии
`copy_if(b,e,b2,p)`	Копирует из последовательности [b,e) в последовательность [b2,b2+(e–b)) элементы, удовлетворяющие предикату p

21.7.1. Алгоритм copy

Базовая версия алгоритма `copy()` определена следующим образом:

```
template<typename In, typename Out>
// Требует Input_iterator<In>() и Output_iterator<Out>()
Out copy(In first, In last, Out res)
{
    while (first!=last) {
        *res = *first;      // Копирует элемент
        ++res;
        ++first;
     }
     return res;
}
```

Получив пару итераторов, алгоритм `copy()` копирует последовательность в другую последовательность, определенную итератором, указывающим на ее первый элемент:

```
void f(vector<double>& vd, list<int>& li)
    // Копирует элементы списка типа int
    //  в вектор чисел типа double
{
```

```
    if (vd.size() < li.size())
        error("Выходной контейнер слишком мал");
    copy(li.begin(), li.end(), vd.begin());
    // ...
}
```

Обратите внимание, что тип входной последовательности алгоритма может отличаться от типа выходной последовательности. Это обстоятельство повышает универсальность алгоритмов из библиотеки STL: они работают со всеми видами последовательностей, не делая лишних предположений об их реализации. Мы не забыли проверить, достаточно ли места в результирующей последовательности для записи вводимых элементов. Такая проверка входит в обязанности программиста. Алгоритмы из библиотеки STL программировались для достижения максимальной универсальности и оптимальной производительности; по умолчанию они не проверяют диапазоны и не выполняют другие потенциально снижающие производительность проверки, защищающие пользователей. Каждый раз, когда это требуется, пользователь должен сам выполнить необходимые проверки, как в примере выше это сделали мы.

21.7.2. Итераторы потоков

Вы будете часто слышать выражение "копировать в поток вывода" или "копировать из потока ввода". Это удобный и полезный способ описания некоторых видов ввода-вывода. Для выполнения этой операции действительно используется алгоритм **copy()**.

Напомним свойства последовательностей.

- Последовательность имеет начало и конец.
- Переход к следующему элементу последовательности осуществляется с помощью оператора **++**.
- Значение элемента последовательности можно получить с помощью оператора ***.

Потоки ввода и вывода можно легко описать точно так же. Рассмотрим пример.

```
ostream_iterator<string> oo{cout}; // Присваивание *oo выполняет
                                   // запись в поток cout
*oo = "Hello, ";                   // Означает cout << "Hello, "
++oo;                              // "Готов к выводу следующего
                                   // элемента"
*oo = "World!\n";                  // Означает cout << "World!\n"
```

Вы можете самостоятельно подумать, как это можно реализовать. В стандартной библиотеке есть тип **ostream_iterator<T>**, предназначенный для

работы с потоком вывода: это итератор, который можно использовать для записи значений типа `T`.

В стандартной библиотеке есть также тип `istream_iterator<T>` для чтения значений типа `T` из потока.

```
istream_iterator<string> ii{cin}; // Чтение *ii – это чтение
                                  // строки из cin
string s1 = *ii;                  // Означает cin >> s1
++ii;                             // "Готов к вводу следующего
                                  // элемента"
string s2 = *ii;                  // Означает cin >> s2
```

Используя итераторы `ostream_iterator` и `istream_iterator`, можно использовать алгоритм `copy()` для осуществления ввода-вывода. Например, вот так — быстро и небрежно — можно сделать простейший словарь:

```
int main()
{
    string from, to;
    cin >> from >> to;  // Вводим имена исходного и целевого файлов

    ifstream is{from};  // Открываем поток ввода
    ofstream os{to};    // Открываем поток вывода

    istream_iterator<string> ii{is}; // Создаем итератор ввода
    istream_iterator<string> eos;    // Ограничитель ввода
    ostream_iterator<string>         // Создаем итератор вывода
                        oo{os,"\n"};

    vector<string> b{ii,eos};      // Вектор инициализируется потоком ввода
    sort(b.begin() ,b.end());      // Сортировка буфера
    copy(b.begin() ,b.end() ,oo); // Копирование буфера в выходной поток
}
```

Итератор `eos` — это ограничитель, означающий "конец ввода". Когда поток `istream` достигает конца ввода (часто именуемый `eof`), его итератор `istream_iterator` становится равным итератору `istream_iterator` по умолчанию (который в нашем исходном тексте имеет имя `eos`).

Обратите внимание на то, что мы инициализируем объект класса `vector` парой итераторов. В качестве инициализатора пара итераторов `{a,b}`, инициализирующая контейнер, означает следующее: "Считать последовательность [a,b) в контейнер". Естественно, для этого мы использовали пару итераторов `{ii,eos}` — начало и конец ввода. Это позволяет нам не использовать явно оператор `>>` и функцию `push_back()`. Мы настоятельно не рекомендуем использовать альтернативный вариант.

```
vector<string> b(max_size); // Не пытайтесь угадать объем входных данных!!
copy(ii,eos,b.begin());
```

Люди, пытающиеся угадать максимальный размер ввода, обычно недооценивают его, переполняют буфер и создают серьезные проблемы как для себя, так и для пользователей. Переполнение буфера также часто приводит к уязвимости системы безопасности компьютера для взлома.

ПОПРОБУЙТЕ

Приведите программу в рабочее состояние и протестируйте ее на небольшом файле, скажем, содержащем несколько сотен слов. Затем испытайте "*настоятельно не рекомендованную версию*", в которой делается предположение о количестве входных данных, и посмотрите, что произойдет при переполнении буфера ввода `b`. Обратите внимание на то, что наихудшим сценарием является тот, в котором вы не замечаете ничего плохого и передаете программу пользователям.

В нашей маленькой программе мы считываем слова, а затем их сортируем. Пока все, что мы делаем, кажется очевидным, но почему мы записываем слова не на нужные места, так что потом вынуждены их сортировать? Что еще неприятнее, оказывается, что мы записываем слова и выводим их на печать столько раз, сколько раз они появляются в потоке ввода.

Последнюю проблему можно решить, используя алгоритм `unique_copy()` вместо алгоритма `copy()`. Функция `unique_copy()` просто не копирует повторяющиеся идентичные значения. Например, если программа, использующая обычный алгоритм `copy()`, получит на вход строку

```
the man bit the dog
```

то она выведет на экран слова

```
bit
dog
man
the
the
```

Если же используем в ней алгоритм `unique_copy()`, то программа выведет следующий список слов:

```
bit
dog
man
the
```

Откуда взялись переходы на новую строку? Вывод с разделителями настолько распространен, что конструктор класса `ostream_iterator` позволяет вам (при необходимости) указать строку, которая должна быть выведена после каждого значения:

```
ostream_iterator<string> oo{os,"\n"};
```

Очевидно, что переход на новую строку — это распространенный выбор для вывода, позволяющий людям легче разбираться в результатах, но, возможно, вы предпочитаете использовать пробелы? Тогда следует написать такой код:

```
ostream_iterator<string> oo{os," "};
```

При этот выводимые результаты выглядели следующим образом.

```
bit dog man the
```

21.7.3. Использование set для поддержки упорядоченности

Существует еще более простой способ получить приведенный в конце предыдущего раздела вывод: использовать контейнер **set** вместо контейнера **vector**.

```
int main()
{
    string from, to;
    cin >> from >> to;  // Вводим имена исходного
                        // и целевого файлов

    ifstream is{from};  // Открываем поток ввода
    ofstream os{to};    // Открываем поток вывода

    set<string> b {istream_iterator<string>{is},
                   istream_iterator<string>{} };
    // Копирование буфера в поток вывода:
    copy(b.begin() ,b.end() ,
         ostream_iterator<string>{os," "});
}
```

Когда мы вставляем значение в контейнер **set**, дубликаты игнорируются. Кроме того, элементы контейнера **set** хранятся в требуемом порядке, так что сортировка оказывается не нужной. При использовании правильных инструментов большинство задач решаются без труда.

21.7.4. Алгоритм copy_if

Алгоритм **copy()** выполняет копирование без каких-либо условий. Алгоритм **unique_copy()** отбрасывает повторяющиеся соседние элементы, имеющие одинаковые значения. Третий алгоритм копирует только те элементы, для которых заданный предикат является истинным.

```
template<typename In, typename Out, typename Pred>
    // Требует Input_iterator<In>(), Output_operator<Out>()
    // и Predicate<Pred, Value_type<In>>()
```

```
Out copy_if(In first, In last, Out res, Pred p)
    // Копирует элементы, удовлетворяющие предикату
{
    while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

Используя наш функциональный объект **Larger_than** из раздела 21.4, мы можем, например, найти все элементы последовательности, которые больше шести.

```
void f(const vector<int>& v)
    // Копируем все элементы, большие шести
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(), Larger_than(6));
    // ...
}
```

Из-за моей ошибки этот алгоритм отсутствовал в стандарте ISO 1998. В настоящее время эта ошибка исправлена, но до сих пор встречаются реализации языка C++, в которых нет алгоритма **copy_if**. В таком случае просто воспользуйтесь определением, данным в этом разделе.

21.8. Сортировка и поиск

Часто мы хотим упорядочить данные. Мы можем добиться этого, используя структуры, поддерживающие упорядочение, такие как **map** и **set**, или выполняя сортировку. Наиболее распространенной и полезной операцией сортировки в библиотеке STL является алгоритм **sort()**, который мы уже несколько раз использовали. По умолчанию алгоритм **sort()** в качестве критерия сортировки использует оператор **<**, но мы можем задавать и собственные критерии.

```
template<typename Ran>
// Требует Random_access_iterator<Ran>()
void sort(Ran first, Ran last);

template<typename Ran, typename Cmp>
// Требует Random_access_iterator<Ran>()
// и Less_than_comparable<Cmp,Value_type<Ran>>()
void sort(Ran first, Ran last, Cmp cmp);
```

В качестве примера сортировки с использованием пользовательского критерия покажем, как упорядочить строки без учета регистра символов.

```
struct No_case { // Выполняется ли lowercase(x) < lowercase(y)?
    bool operator()(const string& x, const string& y) const
    {
        for (int i = 0; i<x.length(); ++i) {
            if (i == y.length()) return false;     // y<x
            char xx = tolower(x[i]);
            char yy = tolower(y[i]);
            if (xx<yy) return true;                // x<y
            if (yy<xx) return false;               // y<x
        }
        if (x.length()==y.length()) return false; // x==y
        return true; // x<y (В x меньше символов)
    }
};

void sort_and_print(vector<string>& vc)
{
    sort(vc.begin(),vc.end(),No_case());

    for (const auto& s : vc)
        cout << s << '\n';
}
```

После того как последовательность отсортирована, нам больше не обязательно перебирать все элементы с самого начала контейнера с помощью функции `find()`; вместо этого можно использовать бинарный поиск, учитывающий упорядочение элементов. По сути, бинарный поиск работает следующим образом.

Предположим, что мы ищем значение *x*. Рассмотрим средний элемент последовательности.

- Если значение этого элемента равно *x*, мы нашли его!
- Если значение этого элемента меньше *x*, то, поскольку элемент со значением *x* должен находиться в правой половине последовательности, мы просматриваем правую половину (выполняя в ней бинарный поиск).
- Если значение этого элемента больше *x*, то поскольку элемент со значением x находится в левой половине последовательности, мы просматриваем левую половину (выполняя в ней бинарный поиск).
- Если мы достигли последнего элемента последовательности (перемещаясь влево или вправо) и не нашли значения *x*, значит, такого элемента в последовательности нет.

Для длинных последовательностей бинарный поиск выполняется намного быстрее, чем алгоритм `find()` (представляющий собой линейный поиск). Алгоритмами бинарного поиска в стандартной библиотеке являются `binary_search()` и `equal_range()`. Что следует пони-

мать под словом "длинные"? Это зависит от обстоятельств, но десяти элементов обычно уже достаточно, чтобы продемонстрировать преимущество алгоритма `binary_search()` над алгоритмом `find()`. На последовательности, состоящей из тысячи элементов, алгоритм `binary_search()` (благодаря времени работы $O(\log_2 N)$) выполняет поиск примерно в 200 раз быстрее, чем алгоритм `find()` (см. раздел 21.6.4).

Алгоритм `binary_search` имеет два варианта.

```
template<typename Ran, typename T>
bool binary_search(Ran first, Ran last, const T& val);

template<typename Ran, typename T, typename Cmp>
bool binary_search(Ran first, Ran last, const T& val, Cmp cmp);
```

Эти алгоритмы требуют, чтобы их входные последовательности были упорядочены. Если это условие не выполняется, то могут возникнуть такие "интересные вещи", как, например, зацикливание. Алгоритм `binary_search()` просто сообщает, содержит ли контейнер заданное значение.

```
void f(vector<string>& vs)      // Вектор vs упорядочен
{
    if (binary_search(vs.begin(),vs.end(),"starfruit")) {
        // В контейнере есть строка "starfruit"
    }
        // ...
}
```

Итак, алгоритм `binary_search()` — идеальное средство, если нас интересует, есть заданное значение в контейнере или нет. Если же нам нужно найти этот элемент, мы можем использовать функцию `lower_bound()`, `upper_bound()` или `equal_range()` (разделы 23.4 и Б.5.4). Как правило, это необходимо, когда элементы контейнера представляют собой объекты, содержащие больше информации, чем просто ключ; когда в контейнере содержатся несколько элементов с одинаковыми ключами; или когда нас интересует, какой именно элемент удовлетворяет критерию поиска.

21.9. Алгоритмы контейнеров

Итак, мы определяем алгоритмы стандартной библиотеки в терминах последовательностей элементов, определяемых итераторами. Входная последовательность определяется парой итераторов [`b`,`e`), где `b` указывает на первый элемент последовательности, а `e` — на элемент, следующий за последним (§20.3). Выходная последовательность определяется итератором, указывающим на первый элемент.

```
void test(vector<int> & v)
{
    sort(v.begin(),v.end()); // Сортировка элементов v
                             // от v.begin() до v.end()
}
```

Все хорошо и обобщенно. Например, можно отсортировать половину вектора:

```
void test(vector<int> & v)
{
    // Сортировка первой половины элементов v:
    sort(v.begin(),v.begin()+v.size()/2);
    // Сортировка второй половины элементов v:
    sort(v.begin()+v.size()/2,v.end());
}
```

Однако указание диапазона элементов слишком многословное, и в большинстве случаев сортируется не какая-то часть контейнера, а весь он целиком. То есть в большинстве случаев мы хотели бы писать кратко:

```
void test(vector<int> & v)
{
    sort(v); // Сортировка v
}
```

Такой вариант алгоритма `sort()` стандартная библиотека не предоставляет, но его нетрудно написать самостоятельно:

```
template<typename C> // Требует Container<C>()
void sort(C& c)
{
    std::sort(c.begin(),c.end());
}
```

Мы сочли эту функцию настолько полезной, что добавили ее в заголовочный файл `std_lib_facilities.h`.

Таким образом легко обработать входные последовательности, но для согласованности со стандартной библиотекой мы обычно возвращаем итераторы.

```
template<typename C, typename V> // Требует Container<C>()
Iterator<C> find(C& c, Val v)
{
    return std::find(c.begin(),c.end(),v);
}
```

Естественно, `Iterator<C>` представляет собой тип итератора класса `C`.

Задание

После выполнения каждой операции выводите содержимое вектора на экран.

1. Определите структуру `struct Item { string name; int iid; double value; /* ... */ };`, создайте контейнер `vector<Item>` с именем `vi` и заполните его десятью элементами, считанными из файла.
2. Отсортируйте контейнер `vi` по полю `name`.
3. Отсортируйте контейнер `vi` по полю `iid`.
4. Отсортируйте контейнер `vi` по полю `value`; выведите его содержимое на экран в порядке убывания значений (т.е. самое большое значение должно быть выведено первым).
5. Вставьте в контейнер элементы `Item("horse shoe",99,12.34)` и `Item("Canon S400",9988,499.95)`.
6. Удалите два элемента `Item` из контейнера `vi`, задавая значение поля `name` удаляемого элемента.
7. Удалите два элемента `Item` из контейнера `vi`, задавая значение поля `iid` удаляемого элемента.
8. Повторите упражнение с контейнером типа `list<Item>` вместо `vector<Item>`.

Теперь поработайте с контейнером `map`.

1. Определите контейнер `map<string,int>` с именем `msi`.
2. Вставьте в него десять пар (имя, значение), например `msi["lecture"]=21`.
3. Выведите пары (имя, значение) в поток `cout` в удобном для вас виде.
4. Удалите пары (имя, значение) из контейнера `msi`.
5. Напишите функцию, считывающую пары (имя, значение) из потока `cin` и помещающую их в контейнер `msi`.
6. Прочитайте десять пар (имя, значение) из потока ввода и поместите их в контейнер `msi`.
7. Выведите элементы контейнера `msi` в поток `cout`.
8. Выведите сумму (целочисленных) значений из контейнера `msi`.
9. Определите контейнер `map<int,string>` с именем `mis`.
10. Введите значения из контейнера `msi` в контейнер `mis`; иначе говоря, если в контейнере `msi` есть элемент (`"lecture",21`), то контейнер `mis` должен содержать элемент (`21,"lecture"`).

11. Выведите элементы контейнера `mis` в поток `cout`.

Несколько заданий, касающихся контейнера `vector`.

1. Прочитайте несколько чисел с плавающей точкой (не меньше 16 значений) из файла в контейнер `vector<double>` с именем `vd`.
2. Выведите элементы контейнера `vd` в поток `cout`.
3. Создайте вектор `vi` типа `vector<int>` с таким же количеством элементов, что и в контейнере `vd`; скопируйте элементы из контейнера `vd` в контейнер `vi`.
4. Выведите в поток `cout` пары (`vd[i],vi[i]`) по одной в строке.
5. Выведите на экран сумму элементов контейнера `vd`.
6. Выведите на экран разность сумм элементов контейнеров `vd` и `vi`.
7. Существует стандартный алгоритм `reverse`, получающий в качестве аргументов последовательность (пару итераторов); поменяйте порядок следования элементов `vd` на противоположный и выведите элементы `vd` в поток `cout`.
8. Вычислите среднее значение элементов в контейнере `vd` и выведите его на экран.
9. Создайте новый контейнер `vector<double>` с именем `vd2` и скопируйте в него элементы контейнера `vd`, которые меньше среднего значения.
10. Отсортируйте контейнер `vd` и выведите его элементы на экран.

Контрольные вопросы

1. Приведите примеры полезных алгоритмов из библиотеки STL.
2. Что делает алгоритм `find()`? Приведите по крайней мере пять примеров.
3. Что делает алгоритм `count_if()`?
4. Что алгоритм `sort(b,e)` использует в качестве критерия поиска?
5. Как алгоритмы STL получают контейнеры в качестве аргумента ввода?
6. Как алгоритмы STL получают контейнеры в качестве аргумента вывода?
7. Как алгоритмы STL обозначают ситуации "не найден" или "сбой"?
8. Что такое функциональный объект?
9. Чем функциональный объект отличается от функции?
10. Что такое предикат?
11. Что делает алгоритм `accumulate()`?
12. Что делает алгоритм `inner_product()`?
13. Что такое ассоциативный контейнер? Приведите не менее трех примеров.

14. Почему класс `list` не является ассоциативным контейнером?
15. Сформулируйте принцип организации бинарного дерева.
16. Что собой представляет сбалансированное дерево?
17. Сколько места занимает элемент в контейнере `map`?
18. Сколько места занимает элемент в контейнере `vector`?
19. Зачем нужен контейнер `unordered_map`, если есть (упорядоченный) контейнер `map`?
20. Чем контейнер `set` отличается от контейнера `map`?
21. Чем контейнер `multimap` отличается от контейнера `map`?
22. Зачем нужен алгоритм `copy()`, если мы вполне могли бы написать простой цикл?
23. Что такое бинарный поиск?

Термины

`accumulate()`	`map`	лямбда-выражение
`binary_search()`	`set`	обобщенность
`copy()`	`sort()`	поиск
`copy_if()`	`unique_copy()`	последовательность
`equal_range()`	`unordered_map`	предикат
`find()`	`upper_bound()`	сбалансированное дерево
`find_if()`	алгоритм	сортировка
`inner_product()`	ассоциативный контейнер	функциональный объект
`lower_bound()`	итератор потока	хеш-функция

Упражнения

1. Перечитайте главу и выполните все упражнения из врезок **Попробуйте**, если вы еще не сделали этого.
2. Найдите надежный источник документации по библиотеке STL и перечислите все стандартные алгоритмы.
3. Самостоятельно реализуйте алгоритм `count()`. Протестируйте его.
4. Самостоятельно реализуйте алгоритм `count_if()`. Протестируйте его.
5. Что нам следовало бы сделать, если бы мы не могли вернуть итератор `end()`, означающий, что элемент не найден? Заново спроектируйте и реализуйте алгоритмы `find()` и `count()` так, чтобы они получали итераторы, указывающие на первый и последний элементы. Сравните результаты со стандартными версиями.
6. В примере класса `Fruit` из раздела 21.6.5 мы копировали структуры `Fruit` в контейнер `set`. Что делать, если мы не хотим *копировать*

эти структуры? Мы могли бы вместо этого использовать контейнер `set<Fruit*>`; однако в этом случае мы были бы вынуждены определить оператор сравнения для этого контейнера. Выполните это упражнение еще раз, используя контейнер `set<Fruit*,Fruit_comparison>`. Обсудите разницу между этими двумя реализациями.

7. Напишите функцию бинарного поиска для класса `vector<int>` (без использования стандартного алгоритма). Выберите любой интерфейс, какой захотите. Протестируйте его. Насколько вы уверены, что ваша функция бинарного поиска работает правильно? Напишите функцию бинарного поиска для контейнера `list<string>`. Протестируйте ее. Насколько похожи эти две функции бинарного поиска? Как вы думаете, были бы они настолько похожими, если бы вам не было ничего известно о библиотеке STL?

8. Вернитесь к примеру, связанному с подсчетом частоты слов из раздела 21.6.1, и модифицируйте его так, чтобы слова выводились в порядке возрастания частот, а не в лексикографическом порядке. При этом на экран должна выводиться строка `3: C++`, а не `C++: 3`.

9. Определите класс `Order` (заказ), члены которого содержат имя клиента, его адрес, дату рождения и контейнер `vector<Purchase>`. Класс `Purchase` должен содержать поля `name`, `unit_price` и `count`, характеризующие товар. Определите механизм чтения из файла и записи в файл объектов класса `Order`. Определите механизм для вывода на экран объектов класса `Order`. Создайте файл, содержащий по крайней мере десять объектов класса `Order`, считайте его в контейнер `vector<Order>`, отсортируйте по имени клиента и запишите обратно в файл. Создайте другой файл, содержащий по крайней мере десять объектов класса `Order`, примерно треть из которых хранится в первом файле, считайте их в контейнер `list<Order>`, отсортируйте по адресам (клиента) и запишите обратно в файл. Объедините эти два файла в третий, используя функцию `std::merge()`.

10. Вычислите общую сумму заказов в двух файлах из предыдущего упражнения. Значение для отдельного объекта класса `Purchase` (разумеется) равно `unit_price*count`.

11. Разработайте графический пользовательский интерфейс для ввода заказов в файлы.

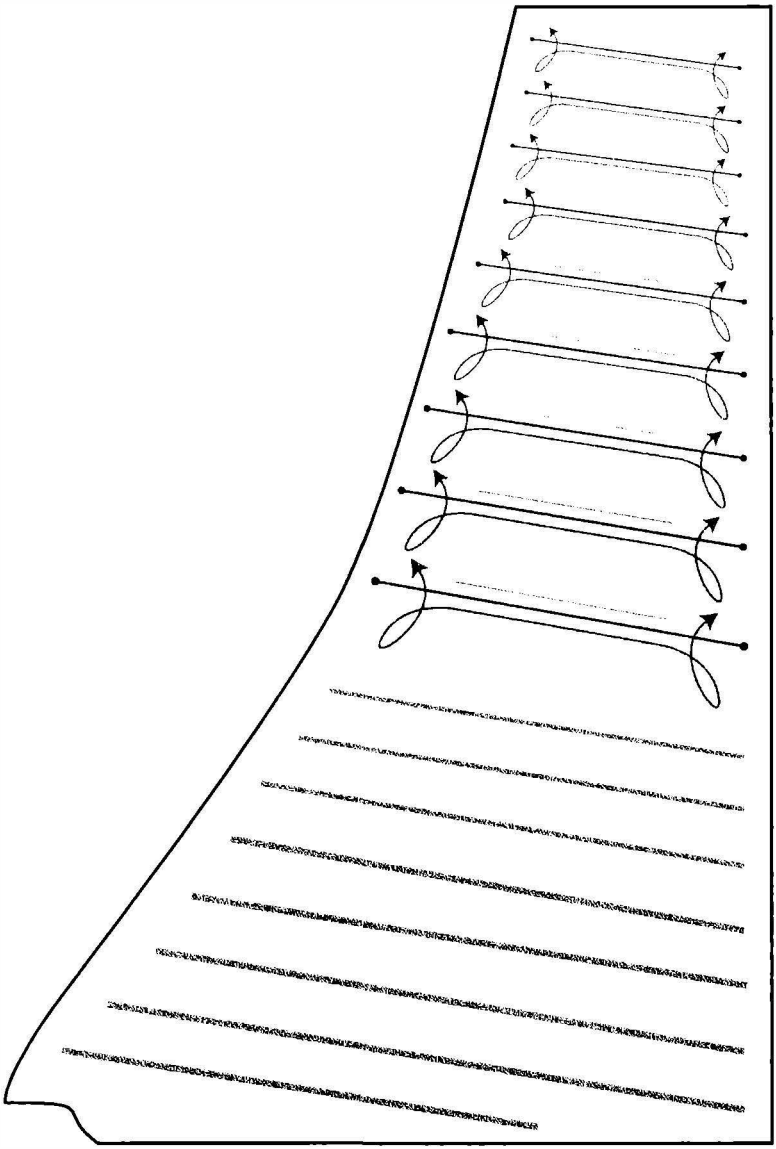
12. Разработайте графический пользовательский интерфейс для запросов к файлу заказов; например, "Найти все заказы от `Joe`," "Определить общую стоимость заказов в файле `Hardware`" или "Перечислить все заказы из файла `Clothing`". *Указание*: сначала разработайте обычный интерфейс и лишь потом на его основе начинайте разрабатывать графический.

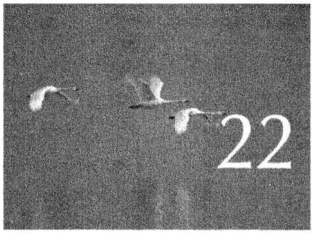
13. Напишите программу, "очищающую" текстовый файл для использования в программе, обрабатывающей запросы на поиск слов; иначе говоря, замените знаки пунктуации пробелами, переведите слова в нижний регистр, замените выражения *don't* словами *do not* (и т.д.) и замените существительные во множественном числе существительными в единственном числе (например, слово *ships* станет *ship*). Не перестарайтесь. Например, определить множественное число в принципе трудно, поэтому просто удаляйте букву *s*, если обнаружите как слово *ship*, так и слово *ships*. Примените эту программу к реальному текстовому файлу, содержащему не менее 5 000 слов (например, к научной статье).
14. Напишите программу (используя результат предыдущего упражнения), отвечающую на вопросы наподобие "Сколько раз встречается в файле слово *ship*?", "Какое слово встречается чаще всего?", "Какое слово в файле самое длинное?", "Какое слово в файле самое короткое?", "Перечислите все слова на букву *s*" или "Перечислите все слова, состоящие из четырех букв".
15. Разработайте графический пользовательский интерфейс для предыдущего упражнения.

Послесловие

Библиотека STL является частью стандартной библиотеки ISO C++, содержащей контейнеры и алгоритмы. Она предоставляет обобщенные, гибкие и полезные базовые инструменты. Эта библиотека позволяет сэкономить массу усилий: изобретать колесо заново может быть забавным, но вряд ли продуктивным занятием. Если у вас нет весомых причин избегать библиотеки STL, используйте ее контейнеры и основные алгоритмы. Что еще важнее, библиотека STL — это пример обобщенного программирования, демонстрирующий, как конкретные задачи и конкретные решения могут вырасти в коллекцию мощных и универсальных инструментов. Если вам необходимо работать с данными — а большинство программистов именно этим и занимаются, — библиотека STL предоставит примеры, множество идей и подходов, которые помогут вам в решении ваших задач.

Часть **IV**
Дополнительные темы





22

Идеалы и история

Когда кто-то говорит "Мне нужен такой язык программирования, в котором достаточно просто сказать, чего я хочу", дайте ему леденец.

— Алан Перлис (Alan Perlis)

В этой главе очень кратко и выборочно изложена история языков программирования и описаны идеалы, которым они призваны служить. Эти идеалы и выражающие их языки программирования образуют основу профессионализма. Поскольку в настоящей книге используется язык C++, мы сосредоточили свое внимание именно на нем, а также на языках, появившихся под его влиянием. Цель этой главы — изложить основы и перспективы развития идей, представленных в книге. Описывая каждый из языков, мы рассказываем о его создателе или создателях: язык — это не просто абстрактное творение; это конкретное решение, найденное людьми для стоявших перед ними в определенный момент времени задач.

22.1. История, идеалы и профессионализм

"История — это чушь", — безапелляционно заявил Генри Форд (Henry Ford). Противоположное мнение широко цитируется еще с античных времен: "Тот, кто не знает историю, обречен ее повторять". Проблема заключается в том, чтобы выбрать, какую историю следует знать, а какую отбросить: "95% всей информации — это чушь", как утверждает другое известное изречение (со своей стороны заметим, что 95%, вероятно, являются преуменьшенной оценкой). Наша точка зрения на связь истории с современностью состоит в том, что без понимания истории невозможно стать профессионалом. Люди, очень мало знающие предысторию своей области знаний, как правило, являются легковерными, поскольку история любой области деятельности замусорена правдоподобными, но не работоспособными идеями. "Плоть" истории состоит из идей, ценность которых доказывается практикой.

Мы бы с удовольствием поговорили о происхождении ключевых идей, лежащих в основе многих языков программирования и разных видов программного обеспечения, таких как операционные системы, базы данных, графические системы, сети, веб, сценарии и так далее, но эти важные и полезные приложения можно найти повсюду. Места, имеющегося в нашем распоряжении, едва хватает лишь для того, чтобы хотя бы поверхностно описать идеалы и историю языков программирования.

Конечная цель программирования заключается в создании полезных программных систем. В горячке споров о методах и языках программирования об этом легко забыть. Помните об этом! Если вам требуется напоминание, перечитайте еще раз главу 1.

22.1.1. Цели и философия языка программирования

Что такое язык программирования? Для чего он предназначен? Ниже приводятся распространенные варианты ответа на первый вопрос.

- Инструмент для инструктирования машин.
- Способ записи алгоритмов.
- Средство общения программистов.
- Инструмент для экспериментирования.
- Средство управления компьютеризированными устройствами.
- Способ выражения отношения между концепциями.
- Средство выражения проектных решений высокого уровня.

Наш ответ таков: "Все вместе и еще больше!" Очевидно, что здесь речь идет об универсальных языках программирования. Кроме них, существуют специализированные и предметно-ориентированные языки программирования, предназначенные для более узких и более точно сформулированных задач. Какие свойства языка программирования считаются желательными?

- Переносимость.
- Безопасность типов.
- Точная определенность.
- Высокая производительность.
- Способность точно выражать идеи.
- Легкая отладка.
- Легкое тестирование.
- Доступ ко всем системным ресурсам.
- Независимость от платформы.
- Возможность выполнения на всех платформах (например, в Linux, Windows, OS/2, в смартфонах, во встроенных системах).
- Устойчивость на протяжении десятилетий.
- Постоянное совершенствование в ответ на изменения, происходящие в прикладной области.
- Легкость обучения.
- Небольшой размер.

- Поддержка популярных стилей программирования (например, объектно-ориентированного и обобщенного программирования).
- Средства, облегчающие анализ программ.
- Широкие возможности.
- Поддержка со стороны большого сообщества.
- Поддержка новичков (студентов, учащихся).
- Исчерпывающие возможности для экспертов (например, конструкторов инфраструктуры).
- Доступность большого количества инструментов для разработки программ.
- Доступность большого количества компонентов программного обеспечения (например, библиотек).
- Поддержка со стороны сообщества разработчиков открытого кода.
- Поддержка со стороны поставщиков основных платформ (Microsoft, IBM и т.д.).

К сожалению, все эти возможности нельзя получить одновременно. Это досадно, поскольку каждое из перечисленных свойств объективно является положительным: каждое из них приносит пользу, а язык, не имеющий этих свойств, вынуждает программистов выполнять дополнительную работу и осложняет им жизнь. Причина, из-за которой невозможно получить все эти возможности одновременно, носит фундаментальный характер: некоторые из них являются взаимоисключающими. Например, язык не может полностью не зависеть от платформы и в то же время открывать доступ ко всем системным ресурсам; программа, обращающаяся к ресурсу, не существующему на конкретной платформе, не сможет на ней работать вообще. Аналогично мы, очевидно, хотели бы, чтобы язык (а также инструменты и библиотеки, необходимые для его использования) был небольшим и легким для изучения, но это противоречит требованию полной поддержки программирования на всех системах и в любых предметных областях.

Идеалы в программировании играют важную роль. Они служат ориентирами при выборе технических решений и компромиссов при разработке каждого языка, библиотеки и инструмента, который должен сделать проектировщик. Да, когда вы пишете программы, вы играете роль проектировщика и должны принимать проектные решения.

22.1.2. Идеалы программирования

Предисловие к книге *The C++ Programming Language* начинается со слов "Язык C++ — универсальный язык программирования, разработанный для того, чтобы серьезные программисты получали больше удовольствия

от работы". Что это значит? Разве программирование не предусматривает поставку готовой продукции? А почему ничего не сказано о правильности, качестве и сопровождении программ? А почему не упомянуто время от начального замысла новой программы до ее появления на рынке? А разве поддержка разработки программного обеспечения не важна? Все это, разумеется, тоже важно, но мы не должны забывать о программисте. Рассмотрим другой пример. Дональд Кнут (Donald Knuth) сказал: "Самое лучшее в компьютере Alto то, что он ночью не работает быстрее". Alto — это компьютер из центра Xerox Palo Alto Research Center (PARC), бывший одним из первых персональных компьютеров, в отличие от обычных компьютеров, предназначенных для совместного использования и провоцировавших острое соперничество между программистами за дневное время работы.

Наши инструменты и методы программирования предназначены для того, чтобы программист работал лучше и достигал более высоких результатов. Пожалуйста, не забывайте об этом. Какие принципы мы можем сформулировать, чтобы помочь программисту создавать наилучшее программное обеспечение с наименьшими затратами энергии? Мы уже выражали наше мнение по всей книге, поэтому этот раздел, по существу, представляет собой резюме.

Основная причина, побуждающая нас создавать хорошую структуру кода, — стремление вносить в него изменения без излишних усилий. Чем лучше структура, тем легче изменить код, найти и исправить ошибку, добавить новое свойство, настроиться на новую архитектуру, повысить быстродействие программы и т.д. Именно это мы имеем в виду, говоря "хорошо".

В оставшейся части раздела мы рассмотрим следующие вопросы.

- Чего мы хотим от кода.
- Два общих подхода к разработке программного обеспечения, сочетание которых обеспечивает лучший результат, чем каждый по отдельности.
- Ключевые аспекты структуры программ, выраженные в коде.
  - Непосредственное выражение идей.
  - Уровень абстракции.
  - Модульность.
  - Согласованность и минимализм.

Идеалы должны воплощаться в жизнь. Они являются основой для размышлений, а не просто забавными фразами, которыми перекидываются менеджеры и эксперты. Наши программы должны приближаться к идеалу. Когда мы заходим в тупик, то возвращаемся назад, чтобы увидеть, не является ли наша проблема следствием отступления от принципов (иногда это помогает). Когда мы оцениваем программу (жела-

тельно еще до ее поставки пользователям), мы ищем отступление от идеалов, которые в будущем могут вызвать проблемы. Применяйте идеалы как можно чаще, но помните, что практичные концепции (например, производительность и простота), а также слабости языка (ни один язык не является совершенным) часто позволяют лишь достаточно близко приблизиться к идеалу, но не достичь его.

Идеалы могут помочь нам принимать конкретные технические решения. Например, мы не можем принять решение о выборе интерфейса для библиотеки единолично и в полной изоляции (см. раздел 14.1). В результате может возникнуть путаница. Вместо этого мы должны вспомнить о нашем первом принципе и решить, что именно является важным для данной конкретной библиотеки, а затем создать логичный набор интерфейсов. В идеале следовало бы сформулировать наши принципы проектирования и принятия компромиссных решений для каждого проекта в его документации и прокомментировать их в коде.

Начиная проект, обдумайте принципы и посмотрите, как они связаны с задачами и ранее существовавшими идеями их решений. Это хороший способ выявления и уточнения идей. Когда позднее, на этапе проектирования и программирования, вы зайдете в тупик, вернитесь назад и найдите место, где ваш код отклонился от идеалов, — именно там, вероятнее всего, кроются ошибки и возникают проблемы, связанные с проектированием. Этот подход является альтернативой методу отладки, принятому по умолчанию, когда программист постоянно проверяет одно и то же место с помощью одних и тех же методов поиска ошибок. "Ошибка всегда кроется там, где вы ее не ожидаете, — или вы ее уже нашли".

22.1.2.1. Чего мы хотим

Как правило, мы хотим следующего.

- *Корректность*. Да, очень трудно определить, что мы подразумеваем под словом "корректный", но это важная часть работы. Часто это понятие в рамках конкретного проекта определяют для нас другие люди, но в этом случае мы должны интерпретировать то, что они говорят.
- *Легкость сопровождения*. Любая успешная программа со временем изменяется; она настраивается на новое аппаратное обеспечение и платформу, дополняется новыми возможностями и при этом из нее необходимо удалить новые ошибки. В следующих разделах мы покажем, как структура программы позволяет достичь этого.
- *Производительность*. Производительность (эффективность) — понятие относительное. Она должна быть адекватной цели программы. Часто программисты утверждают, что эффективный код с

необходимостью должен быть низкоуровневым, а высокоуровневая структура ухудшает эффективность программы. Мы, напротив, считаем, что следование рекомендуемым нами принципам часто позволяет обеспечивать более высокую эффективность кода. Примером такого кода является библиотека STL, которая одновременно является абстрактной и очень эффективной. Низкая производительность часто может быть следствием как чрезмерного увлечения низкоуровневыми деталями, так и пренебрежения ими.

- *Своевременное завершение.* Выход совершенной программы на год позже запланированного срока — не слишком хорошее событие. Очевидно, что люди хотят невозможного, но мы должны создать качественное программное обеспечение за разумный срок. Существует миф, утверждающий, что законченная в срок программа не может быть высококачественной. Мы же считаем, что акцент на хорошую структуру (например, управление ресурсами, инварианты и проект интерфейса) и ориентация на тестирование и использование подходящих библиотек (часто разработанных для конкретных приложений или предметных областей) позволяют полностью уложиться в сроки.

Все сказанное стимулирует наш интерес к структуре кода.

- Если в программе есть ошибка (а каждая большая программа содержит ошибки), то найти ее легче, если программа имеет четкую структуру.
- Если программу необходимо объяснить постороннему или как-то модифицировать, то четкую структуру понять намного легче, чем мешанину низкоуровневых деталей.
- Если программа испытывает проблемы с производительностью, то настроить высокоуровневую программу, как правило, намного легче (поскольку она точнее соответствует общим принципам и имеет хорошо определенную структуру), чем низкоуровневую. Для начинающих программистов высокоуровневая структура намного понятнее. Кроме того, высокоуровневый код намного легче тестировать и настраивать, чем низкоуровневый.

Программа обязательно должна быть понятной. Хорошим считается все, что помогает нам понимать программу и размышлять о ней. В общем случае порядок лучше беспорядка, если только порядок не является результатом чрезмерного упрощения.

22.1.2.2. Общие подходы

Существуют два подхода к созданию правильного программного обеспечения.

- *Снизу вверх.* Система компонуется только из составляющих частей, правильность которых уже доказана.
- *Сверху вниз.* Система компонуется из составляющих частей, предположительно содержащих ошибки, а уже затем все эти ошибки вылавливаются.

Интересно, что наиболее надежные системы созданы путем сочетания обоих подходов, хотя они очевидным образом противоречат друг другу. Причина проста: для крупных реальных систем ни один из этих подходов не гарантирует требуемой правильности, адаптируемости и удобства сопровождения.

- Мы не можем создать и проверить основные компоненты, заранее устранив все источники ошибок.
- Мы не можем полностью компенсировать недостатки основных компонентов (библиотек, подсистем, иерархий классов и т.д.), объединив их в законченную систему.

Однако сочетание этих двух подходов может дать больше, чем каждый из них по отдельности: мы можем создать (или заимствовать, или приобрести) компоненты, имеющие достаточно высокое качество, так что остальные проблемы можно устранить с помощью обработки ошибок и систематического тестирования. Кроме того, если мы продолжаем создавать компоненты все большего качества, то из них можно создавать все большие части системы, сокращая долю необходимого беспорядочного связующего их кода.

Тестирование является существенной частью разработки программного обеспечения. Более подробно оно обсуждается в главе 26. Тестирование — это систематический поиск ошибок. Тестируйте как можно раньше и как можно чаще. Мы пытаемся разрабатывать наши программы так, чтобы упростить тестирование и помешать ошибкам скрыться в запутанном коде.

22.1.2.3. Непосредственное выражение идей

Когда мы выражаем какую-то идею — высоко- или низкоуровневую, — желательно сделать это непосредственно в коде, а не искать обходные пути. Основной принцип выражения идей непосредственно в коде имеет несколько специфических вариантов.

- *Выражение идей непосредственно в коде.* Например, аргумент лучше представлять с помощью специального типа (например, `Month` или `Color`), а не общего (например, `int`).
- *Независимое представление в коде независимых идей.* Например, стандартная функция `sort()` (за некоторым исключением) может упорядочивать любой стандартный контейнер с любым типом элементов; концепции сортировки, критерии сортировки, контейнеры

и тип элемента являются независимыми понятиями. Если бы мы должны были создать "вектор объектов, расположенных в динамической памяти, элементы которого принадлежат классу, порожденному из класса `Object` с функцией-членом `before()`, определенной для применения в функции `vector::sort()`", то должны были бы создать существенно менее обобщенную версию функции `sort()`, поскольку сделали предположения о способе хранения, иерархии классов, доступных функциях-членах, упорядочении и т.д.

- *Представление взаимоотношений между идеями непосредственно в коде.* Наиболее распространенными отношениями, которые можно непосредственно выразить в коде, являются наследование (например, класс `Circle` является разновидностью класса `Shape`) и параметризация (например, класс `vector<T>` выражает нечто общее для всех векторов, независимо от типа их элементов).
- *Свободное сочетание идей, выраженных в коде — там и только там, где такая комбинация имеет смысл.* Например, функция `sort()` позволяет использовать разные типы элементов и виды контейнеров, но эти элементы должны поддерживать операцию `<` (если нет, то следует использовать функцию `sort()` с дополнительным аргументом, задающим критерий сравнения), а контейнеры, которые мы собираемся упорядочивать, должны поддерживать итераторы с произвольным доступом.
- *Простое выражение простых идей.* Следование принципам, сформулированным выше, может привести к созданию кода, носящего слишком общий характер. Например, мы можем столкнуться с иерархией классов с более сложной таксономией (структурой наследования), чем требуется, или с семью параметрами для каждого (кажущегося простым) класса. Для того чтобы избежать возможных осложнений, мы пытаемся создавать простые версии для наиболее распространенных или наиболее важных ситуаций. Например, кроме общей версии функции `sort(b,e,op)`, сортирующей элементы с помощью критерия сравнения `op`, существует вариант `sort(b,e)`, выполняющий неявную сортировку с помощью отношения "меньше". Если бы мы могли, то предусмотрели бы также версию `sort(c)` для сортировки стандартного контейнера с помощью отношения "меньше" и функцию `sort(c,op)` для сортировки стандартного контейнера с помощью критерия `op`.

22.1.2.4. Уровень абстракции

Мы предпочитаем *работать на максимально возможном уровне абстракции*, иначе говоря, стремимся выражать свои решения в как можно более общем виде.

Рассмотрим, например, как представлены записи в телефонной книге, которая может храниться в вашем мобильном телефоне. Мы могли бы представить множество пар (имя, значение) с помощью класса `vector<pair<string,Value_type>>`. Однако если мы почти всегда обращаемся к этому множеству для поиска имени, то более высокий уровень абстракции обеспечит нам класс `map<string,Value_type>`. Это позволит не писать (и, соответственно, не отлаживать) функции доступа к записям. С другой стороны, класс `vector<pair<string,Value_type>>` сам по себе находится на более высоком уровне абстракции, чем два массива, `string[max]` и `Value_type[max]`, в которых отношение между строкой и значением носит неявный характер. На самом низком уровне абстракции могло бы находиться сочетание типа `int` (количество элементов) и двух указателей `void*` (ссылающихся на какую-то форму представления, известную программисту, но не компилятору). В нашем примере каждое из предложенных решений можно отнести к низкому уровню абстракции, поскольку в каждом из них основное внимание сосредоточено на представлении пар значений, а не на их функциях. Для того чтобы приблизиться к реальному приложению, следует определить класс, который непосредственно отражает способ его использования. Например, мы могли бы написать код приложения, используя класс `Phonebook` с удобным интерфейсом. Класс `Phonebook` можно было бы реализовать с помощью одного из описанных выше представлений данных.

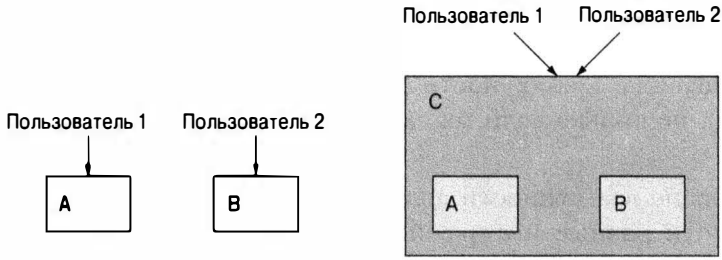
Причина, по которой мы предпочитаем оставаться на наивысшем уровне абстракции (если в нашем распоряжении есть соответствующий механизм абстракций и наш язык поддерживает его на приемлемом уровне эффективности), заключается в том, что такие формулировки ближе к нашим размышлениям о задаче и ее решениях, чем решения, выраженные на уровне аппаратного обеспечения компьютера.

Как правило, основной причиной перехода на низкий уровень абстракции называют эффективность. Однако это следует делать только в случае реальной необходимости (раздел 25.2.2). Использование низкоуровневых (более примитивных) языковых средств не всегда повышает производительность программы. Иногда оно исключает возможности оптимизации. Например, используя класс `Phonebook`, можем выбрать способ его реализации, например, в виде сочетания массивов `string[max]` и `Value_type[max]` или в виде класса `map<string,Value_type>`. Для одних приложений более эффективным оказывается первый вариант, а для других — второй. Естественно, производительность не является основным фактором, если вы пишете программу для хранения записей из своей телефонной книжки. Но она становится существенной, если необходимо хранить и обрабатывать миллионы записей. Что еще важнее, использование низкоуровневых средств сопряжено с затратами рабочего времени, которого программисту в результате не хватит на усовершенствование программы (повышение ее производительности или чего-то другого).

22.1.2.5. Модульность

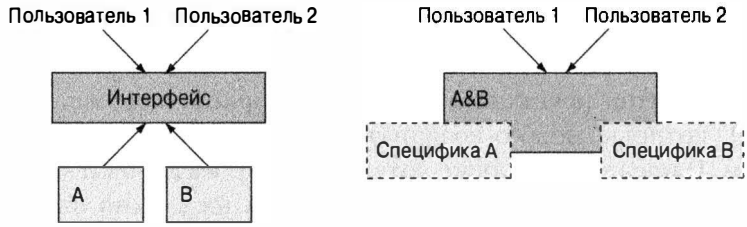
Модульность — это идеал. Мы хотим составлять наши системы из компонентов (функций, классов, иерархий классов, библиотек и т.д.), которые можно создавать, анализировать и тестировать по отдельности. В идеале нам также хотелось бы проектировать и реализовывать такие компоненты таким образом, чтобы их можно было использовать повторно, в нескольких программах. *Повторное использование* (reuse) — это создание систем из ранее протестированных компонентов, которые уже были где-то использованы, а также проектирование и применение таких компонентов. Мы уже касались этой темы, обсуждая классы, иерархии классов, проектирование интерфейсов и обобщенное программирование. Большинство из того, что мы говорим о стилях программирования в разделе 22.1.3, связано с проектированием, реализацией и использованием компонентов, допускающих повторное использование. Следует подчеркнуть, что не каждый компонент можно использовать в нескольких программах; некоторые программы являются слишком специализированными, и их нелегко приспособить для использования в других условиях.

Модульность кода должна отражать основные логические разделы приложения. Не следует повышать степень повторного использования, просто погружая два совершенно разных класса A и B в "повторно используемый компонент" C. Объединение интерфейсов классов A и B в новом модуле C усложняет код.



Здесь оба пользователя работают с модулем C. Пока вы не заглянете внутрь модуля C, вы можете подумать, что оба пользователя получают преимущества благодаря тому, что совместно используют общедоступный компонент. Выгоды от совместного использования (повторного использования) могут (в данном случае этого не происходит) включать в себя более легкое тестирование, меньший объем кода, расширения пользовательской базы и т.д. К сожалению, за исключением случая излишнего упрощения, это не столь уж редкая ситуация.

Чем же можно помочь? Может быть, следует создать общий интерфейс классов A и B?



Эти диаграммы подсказывают, что следует использовать наследование и параметризацию соответственно. Но в обоих случаях, для того чтобы эта работа имела смысл, интерфейс должен быть меньше, чем простое объединение интерфейсов классов A и B. Иначе говоря, для того чтобы пользователь получил выгоду от принятого решения, классы A и B должны иметь фундаментальную общность. Обратите внимание на то, что мы снова вернулись к интерфейсам (см. разделы 9.7 и 25.4.2) и как следствие — к инвариантам (см. раздел 9.4.3).

22.1.2.6. Согласованность и минимализм

Согласованность и минимализм — первичные идеалы для выражения идей. Таким образом, мы можем забыть о них как о вопросах, касающихся внешней формы. Однако запутанный проект очень трудно реализовать элегантно, поэтому требование согласованности и минимализма можно рассматривать как критерии проектирования, влияющие на большинство мельчайших деталей программы.

- Не добавляйте возможность, если сомневаетесь в ее необходимости.
- Похожие возможности должны иметь похожие интерфейсы (и имена), но только если их сходство носит фундаментальный характер.
- Непохожие возможности должны иметь непохожие имена (и по возможности разные интерфейсы), но только если их различие носит фундаментальный характер.

Согласованное именование, стиль интерфейса и стиль реализации облегчают эксплуатацию программы. Если код последователен, то программист не будет вынужден изучать новый набор соглашений для каждой части крупной системы. Примером является библиотека STL (см. главы 20 и 21, разделы Б.4–6). Если обеспечить согласованность не удается (например, из-за наличия старого кода или кода, написанного на другом языке программирования), то здравой идеей может быть создание интерфейса, который согласуется со стилем остальной части программы. В противном случае этот чужеродный ("странный", "плохой") стиль "заразит" каждую часть программы, вынужденную к нему обращаться.

Для того чтобы обеспечить минимализм и согласованность, следует тщательно (и последовательно) документировать каждый интерфейс. В этом случае легче будет заметить несогласованность и дублирование кода. Документирование предусловий, постусловий и инвариантов может оказаться особенно полезным, поскольку оно привлекает внимание к управлению ресурсами и сообщениям об ошибках. Последовательная обработка ошибок и согласованная стратегия управления ресурсами играют важную роль в обеспечении простоты программы (см. раздел 19.5).

Некоторые программисты придерживаются принципа проектирования KISS ("Keep It Simple, Stupid" — "Делай проще, тупица"). Нам даже доводилось слышать, что принцип KISS — единственный стоящий принцип проектирования. Однако мы предпочитаем менее вызывающие формулировки, например "Keep simple things simple" ("Не усложняй простые вещи") и "Keep it simple: as simple as possible, but no simpler" ("Все должно быть как можно более простым, но не проще"). Последнее высказывание принадлежит Альберту Эйнштейну (Albert Einstein). Оно подчеркивает опасность чрезмерного упрощения, выходящего за рамки здравого смысла и разрушающего проект. Возникает очевидный вопрос: "Просто для кого и по сравнению с чем?"

22.1.3. Стили и парадигмы

Проектируя и реализуя программы, мы должны придерживаться последовательного стиля. Язык C++ поддерживает четыре главных стиля, которые можно считать фундаментальными.

- Процедурное программирование.
- Абстракция данных.
- Объектно-ориентированное программирование.
- Обобщенное программирование.

Иногда их называют (несколько помпезно) парадигмами программирования. Существует еще много парадигм, например функциональное программирование (functional programming), логическое программирование (logic programming), программирование на основе правил (rule-based programming), программирование на основе ограничений (constraints-based programming) и аспектно-ориентированное программирование (aspect-oriented programming). Однако язык C++ не поддерживает эти парадигмы непосредственно, а мы не можем охватить их в одной книге, поэтому откладываем эти вопросы на будущее.

- *Процедурное программирование.* Основная идея этой парадигмы — составление программы из функций, применяемых к аргументам. Примерами являются библиотеки математических функций, таких

как `sqrt()` и `cos()`. Язык C++ поддерживает этот стиль посредством понятия функций (см. главу 8). Вероятно, самой ценной является возможность выбирать механизм передачи аргументов по значению, по ссылке и по константной ссылке. Часто данные организовываются в структуры с помощью конструкций `struct`. Явные механизмы абстракции (такие, как закрытые данные-члены и функции-члены класса) не используются. Отметим, что этот стиль программирования — и функции — является неотъемлемой частью любого другого стиля.

- *Абстракция данных.* Основная идея этой парадигмы — сначала предоставить набор типов для предметной области, а затем писать программы, их использующие. Классическим примером являются матрицы (разделы 24.3–24.6). Интенсивно используется явное сокрытие данных (например, использование закрытых членов класса). Популярными примерами абстракции данных являются стандартные классы `string` и `vector`, демонстрирующие сильную зависимость между абстракциями данных и параметризацией, используемой в обобщенном программировании. Слово "абстракция" используется в названии этой парадигмы потому, что взаимодействие с типом осуществляется посредством интерфейса, а не прямого доступа к его реализации.

- *Объектно-ориентированное программирование.* Основная идея этой парадигмы программирования — организация типов в иерархии, чтобы выразить их отношения непосредственно в коде. Классический пример — иерархия `Shape`, описанная в главе 14. Этот подход имеет очевидную ценность, когда типы действительно имеют иерархические взаимоотношения. Однако существует сильная тенденция к его избыточному применению; иначе говоря, люди создают иерархии типов, не имея на это фундаментальных причин. Если кто-то создает производные типы, задайте вопрос "Зачем?" Что выражает это порождение? Чем различие между базовым и производным классами может мне помочь в данном конкретном случае?

- *Обобщенное программирование.* Основная идея этой парадигмы программирования — взять конкретные алгоритмы и поднять их на более высокий уровень абстракции, добавив параметры, позволяющие варьировать типы без изменения сущности алгоритма. Простым примером такого повышения уровня абстракции является функция `high()`, описанная в главе 20. Алгоритмы `find()` и `sort()` из STL являются классическими алгоритмами поиска и сортировки, выраженными в очень общей форме с помощью обобщенного программирования. (См. также примеры в главах 20 и 21.)

Подведем итоги. Часто люди говорят о стилях программирования (парадигмах) так, будто они представляют собой противоречащие друг другу альтернативы: вы используете либо обобщенное программирование, либо объектно-ориентированное. Если хотите выразить решения задач наилучшим образом, используйте комбинацию этих стилей. Выражение "наилучшим образом" означает, что вашу программу легко читать, писать, легко эксплуатировать и при этом она достаточно эффективна. Рассмотрим классический пример **Shape** (происходящий из языка Simula; см. раздел 22.2.4), который обычно считается воплощением объектно-ориентированного программирования. Первое решение может выглядеть так:

```
void draw_all(vector<Shape*>& v)
{
    for(int i = 0; i<v.size(); ++i) v[i]->draw();
}
```

Этот фрагмент кода действительно выглядит "довольно объектно-ориентированным". Он существенно зависит от иерархии классов и вызова виртуальной функции, при котором автоматически находится правильная функция **draw()** для каждого конкретного объекта класса **Shape**; иначе говоря, для объекта класса **Circle** будет вызвана функция **Circle::draw()**, а для объекта класса **Open_polyline** — функция **Open_polyline::draw()**. Однако класс **vector<Shape*>** по существу является конструкцией обобщенного программирования: он использует параметр (тип элемента), который выясняется на этапе компиляции. Следует подчеркнуть, что итерацию по всем элементам можно выразить с помощью алгоритма стандартной библиотеки.

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),mem_fun(&Shape::draw));
}
```

Третьим аргументом функции **for_each()** является функция, которая должна вызываться для каждого элемента последовательности, заданной двумя первыми аргументами (раздел Б.5.1). Предполагается, что третья функция представляет собой обычную функцию (или функцию-объект), которая вызывается с помощью синтаксической конструкции **f(x)**, а не функцию-член, вызываемую с помощью синтаксической конструкции **p->f()**. Следовательно, для того чтобы указать, что на самом деле мы хотим вызвать функцию-член (виртуальную функцию **Shape::draw()**), необходимо использовать стандартную библиотечную функцию **mem_fun()** (раздел Б.6.2). Дело в том, что функции **for_each()** и **mem_fun()**, будучи шаблонными, на самом деле не очень хорошо соответствуют объектно-ориентированной парадигме; они полностью относятся к обобщенному программированию. Еще интереснее

то, что функция **mem_fun()** является автономной (шаблонной) функцией, возвращающей объект класса. Другими словами, ее следует отнести к простой абстракции данных (нет наследования) или даже к процедурному программированию (нет сокрытия данных). В результате мы можем констатировать, что всего лишь одна строка кода использует все четыре фундаментальных стиля программирования, поддерживаемых языком C++.

Зачем же мы написали вторую версию примера для рисования всех фигур? По существу она не отличается от первой, к тому же на несколько символов длиннее. В свое оправдание скажем, что выражение концепции цикла с помощью функции **for_each()** является "более очевидным и менее уязвимым для ошибок", чем цикл **for**, но для многих этот аргумент не является очень убедительным. Лучше сказать, что функция **for_each()** выражает то, что мы хотим сделать (пройти по последовательности), а не как мы это хотим сделать. Однако большинству людей достаточно просто сказать "Это полезно". Такая запись демонстрирует путь к обобщению (в лучших традициях обобщенного программирования), позволяющий решить множество задач. Почему все фигуры хранятся в векторе, а не в списке или в некоторой обобщенной последовательности? Давайте напишем третью (более общую) версию.

```
template<typename Iter> void draw_all(Iter b, Iter e)
{
    for_each(b,e,mem_fun(&Shape::draw));
}
```

Теперь код работает со всеми видами последовательностей фигур. В частности, мы можем даже вызвать его для элементов массива объектов класса **Shape**.

```
Point p{0,100};
Point p2{50,50};
Shape* a[] = {new Circle{p,50},new Triangle{p,p2,Point{25,25}}};
draw_all(a,a+2);
```

Мы можем также предоставить версию попроще, ограничив ее работой только с контейнерами:

```
template<typename Cont> void draw_all(Cont& c)
{
    for (auto& p : c) p->draw();
}
```

Или даже, воспользовавшись концепциями стандарта C++14 (см. раздел 19.3.3), можем написать

```
void draw_all(Container& c)
{
    for (auto& p : c) p->draw();
}
```

Главное в том, что этот код явно является объектно-ориентированным, обобщенным и очень похожим на обычный процедурный код. Он основан на абстракции данных в иерархии классов и реализации отдельных контейнеров. За неимением лучшего термина мы называем программирование, использующее комбинацию наиболее удобных стилей, *мультипарадигменным* (multi-paradigm programming). Однако я пришел к тому, что думаю о нем как просто о *программировании*: "парадигмы", в первую очередь, отражают ограниченный взгляд на то, как могут быть решены поставленные задачи, и слабые стороны языков программирования, использованных для выражения этих решений. Я предвижу светлое будущее программирования по мере развития техники, языков программирования и средств поддержки.

22.2. Обзор истории языков программирования

На заре человечества программисты вручную высекали нули и единицы на камнях! Ну хорошо, хорошо, это просто небольшое преувеличение. В этом разделе мы вернемся (почти) к началу и кратко опишем основные вехи истории языков программирования в аспекте их связи с языком C++.

Существует множество языков программирования. Они появляются со скоростью примерно 2000 языков за десять лет; впрочем, скорость их исчезновения примерно такая же. В этом разделе, кратко упомянув десяток языков, мы охватим последние почти шестьдесят лет. Более подробную информацию можно найти на веб-странице `http://research.ihost.com/hopl/HOPL.html`. Там же имеются ссылки на все статьи, опубликованные на трех конференциях ACM SIGPLAN HOPL (History of Programming Languages — история языков программирования). Эти статьи прошли строгое рецензирование, а следовательно, они более полны и достоверны, чем среднестатистические источники информации в сети веб. Все языки, которые мы обсудим, были представлены на конференциях HOPL. Заметим, что, набрав полное название статьи в поисковой системе в вебе, вы имеете немалые шансы найти ее. Кроме того, большинство специалистов по компьютерным наукам, упомянутых в этом разделе, имеют свои страницы в вебе, на которых можно найти больше информации об их работе.

Мы вынуждены приводить только очень краткое описание языков в этой главе, при том что каждый упомянутый язык (и сотни не упомянутых) заслуживает отдельной книги. В каждом языке мы выбрали только самое главное. Надеемся, что читатели воспримут это как приглашение к самостоятельному поиску, а не подумают "И это все, что можно сказать о языке X!" Напомним, что каждый упомянутый здесь язык был в свое время большим достижением и внес важный вклад в программирование. Просто из-за недостатка места мы не в состоянии отдать этим языкам должное, но не

упомянуть о них было бы совсем несправедливо. Мы хотели бы также привести несколько строк кода на каждом из этих языков, но, к сожалению, для этого просто не хватило места (см. упр. 5 и 6).

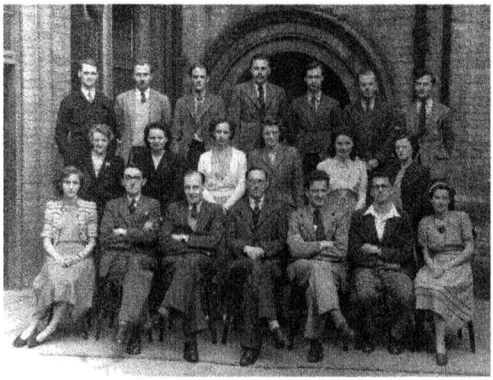
Слишком часто об искусственно созданных объектах (например, о языках программирования) говорят лишь, что они собой представляют, или как о результатах анонимного "процесса разработки". Это неправильное изложение истории: как правило, в особенности на первых этапах, на язык влияют идеи, место работы, личные вкусы и внешние ограничения одного человека или (чаще всего) нескольких людей. Таким образом, за каждым языком стоят конкретные люди. Ни компании IBM и Bell Labs, ни Cambridge University, ни другие организации не разрабатывают языки программирования. Их изобретают люди, работающие в этих организациях, обычно в сотрудничестве со своими друзьями и коллегами.

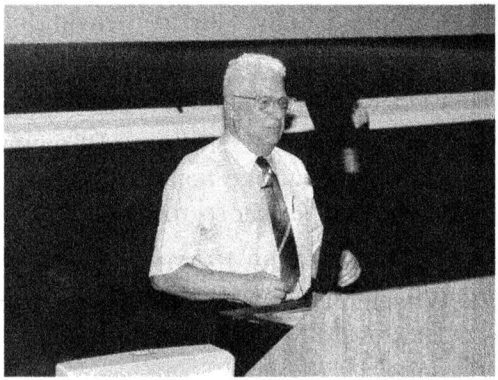
Стоит отметить курьезный феномен, который часто приводит к искаженному взгляду на историю. Фотографии знаменитых ученых и инженеров часто делались тогда, когда они уже были знаменитыми и маститыми членами национальных академий, Королевского общества, рыцарями Святого Джона, лауреатами премии Тьюринга и т.д. Иначе говоря, на фотографиях они на десятки лет старше, чем в те годы, когда они сделали свои выдающиеся изобретения. Почти все они продуктивно работали до самой глубокой старости. Однако, вглядываясь в далекие годы возникновения наших любимых языков и методов программирования, попытайтесь представить себе молодого человека (в науке и технике по-прежнему слишком мало женщин), пытающегося выяснить, хватит ли у него денег для того, чтобы пригласить свою девушку в приличный ресторан, или молодого отца семейства, решающего, как совместить презентацию важной работы на конференции с отпуском. Седые бороды, лысые головы и немодные костюмы появятся много позже.

22.2.1. Первые языки программирования

Когда — начиная с 1949 года — в мире появились первые электронные компьютеры с хранимыми программами, каждый из них имел собственный язык программирования. Существовало взаимно однозначное соответствие между выражением алгоритма (например, вычисления орбиты планеты) и командами конкретной машины. Очевидно, что ученый (пользователями чаще всего были ученые) писал математические формулы, но программа представляла собой список машинных инструкций. Первые примитивные списки состояли из десятичных или восьмеричных цифр, точно соответствовавших их представлению в машинной памяти. Позднее появились ассемблер и "автокоды"; иначе говоря, люди разработали языки, в которых машинные команды и машинное оборудование (например, регистры) имели символические имена. Так, программист мог написать

"LD R0 123", чтобы загрузить содержимое ячейки памяти, расположенной по адресу 123, в нулевой регистр. Однако каждая машина по-прежнему имела собственный набор команд и собственный язык программирования.





Ярким представителем разработчиков языков программирования того времени является, несомненно, Дэвид Уилер (David Wheeler) из компьютерной лаборатории Кембриджского университета (University of Cambridge Computer Laboratory). В 1948 году он написал первую реальную программу, которая когда-либо была выполнена на компьютере, хранившем программы в своей памяти (программа, вычислявшая таблицу квадратов; см. раздел 4.4.2.1). Он был одним из десяти людей, объявивших о создании первого компилятора (для машинно-зависимого автокода). Он изобрел вызов функции (да, даже такое очевидное и простое понятие было когда-то изобретено). В 1951 году он написал блестящую статью о разработке библиотек, которая на двадцать лет опередила свое время! В соавторстве с Морисом Уилксом (Maurice Wilkes) (см. фото выше) и Стенли Джиллом (Stanley Gill) он написал первую книгу о программировании. Он получил первую степень доктора философии в области компьютерных наук (в Кембриджском университете в 1951 году), а позднее внес большой вклад в развитие аппаратного обеспечения (кеш-архитектура и ранние локальные сети) и алгоритмов (например, алгоритм шифрования TEA (см. раздел 25.5.6) и преобразование Бэрроуза–Уилера (Burrows–Wheeler transform) — алгоритм сжатия, использованный в архиваторе bzip2). Дэвид Уилер стал научным руководителем докторской диссертации Бьярне Страуструпа (Bjarne Stroustrup). Как видите, компьютерные науки — молодая дисциплина. Дэвид Уилер выполнил большую часть своей выдающейся работы, еще будучи аспирантом. Впоследствии он стал профессором Кембриджского университета и членом Королевского общества (Fellow of the Royal Society).

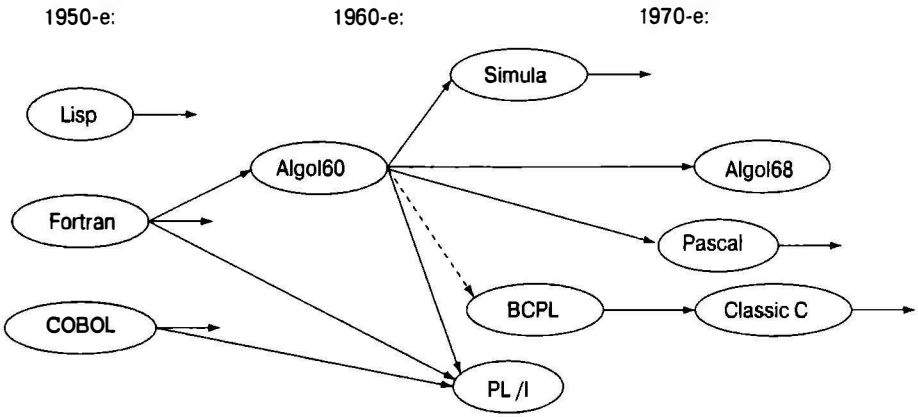
Ссылки

1. Burrows, M., and David Wheeler. "A Block Sorting Lossless Data Compression Algorithm." Technical Report 124, Digital Equipment Corporation, 1994.

2. Bzip2: `www.bzip.org`.
3. Cambridge Ring website: `http://koo.corpus.cam.ac.uk/projects/earlyatm/cr82`.
4. Campbell-Kelly, Martin. "David John Wheeler." *Biographical Memoirs of Fellows of the Royal Society*, Vol. 52, 2006. (Формальная биография.)
5. EDSAC: `http://en.wikipedia.org/wiki/EDSAC`.
6. Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley, 1968 (и более поздние издания). Поищите имя "David Wheeler" в предметном указателе каждого тома.
7. TEA: `http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm`.
8. Wheeler, D. J. "The Use of Sub-routines in Programmes." Proceedings of the 1952 ACM National Meeting. (Это библиотека технических отчетов начиная с 1951 года.)
9. Wilkes, M.V., D.Wheeler, and S.Gill. *Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley Press, 1951; 2nd edition, 1957. (Первая книга о программировании.)

22.2.2. Корни современных языков программирования

Ниже приведена диаграмма важнейших первых языков программирования.



Важность этих языков частично объясняется тем, что они широко использовались (а в некоторых случаях используются и ныне), а частично тем, что они стали предшественниками важных современных языков, причем часто наследники имели те же имена. Этот раздел посвящен трем ранним языкам — Fortran, COBOL и Lisp, — ставшим прародителями большинства современных языков программирования.

22.2.2.1. Язык программирования Fortran

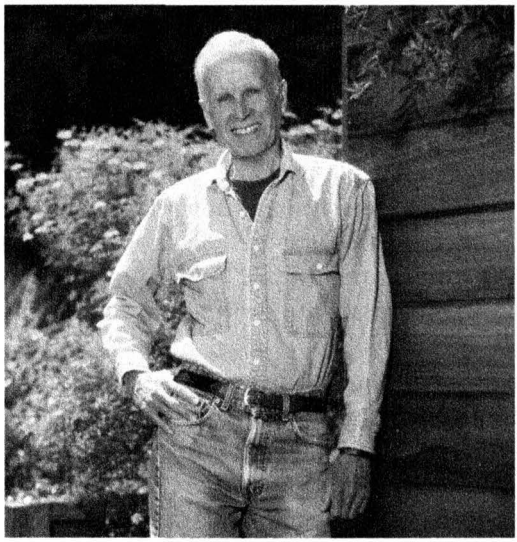
Появление языка Fortran в 1956 году, вероятно, является наиболее значительным событием в истории языков программирования. Fortran — это сокращение от "Formula Translation" (трансляция формул). Его основная идея заключалась в генерации эффективного машинного кода из текста, ориентированного на людей, а не на машины. Система обозначений, принятая в языке Fortran, напоминала систему, которой пользовались ученые и инженеры, решающие математические задачи, а не машинные команды (тогда лишь недавно появившихся) электронных компьютеров.

С современной точки зрения язык Fortran можно рассматривать как первую попытку непосредственного представления предметной области в коде. Он позволял программистам выполнять операции линейной алгебры почти так же, как они описаны в учебниках. В языке Fortran есть массивы, циклы и стандартные математические формулы (использующие стандартные математические обозначения, такие как `x+y` и `sin(x)`). Язык содержал стандартную библиотеку математических функций, механизмы ввода-вывода, причем пользователь мог самостоятельно определять дополнительные функции и библиотеки.

Система обозначений была достаточно машинно-независимой, так что код на языке Fortran часто можно было переносить из одного компьютера в другой с минимальными изменениями. Это было огромное достижение в то время. По этим причинам язык Fortran считается первым высокоуровневым языком программирования.

Считалось важным, чтобы машинный код, сгенерированный на основе исходного кода, написанного на языке Fortran, был как можно ближе к оптимальному с точки зрения эффективности: машины в те времена были огромными и чрезвычайно дорогими (во много раз дороже зарплаты коллектива программистов), удивительно (по современным меркам) медленными (около 100 тыс. операций в секунду) и имели абсурдно малую память (скажем, 8 Кбайт). Однако люди умудрялись втискивать в эти машины полезные программы, и это ограничивало применение улучшенной системы обозначений (ведущее к повышению производительности работы программиста и повышению переносимости программ).

Язык Fortran пользовался огромным успехом в области научных и инженерных вычислений, для которых он, собственно, и предназначался. С момента своего появления он постоянно эволюционировал. Основными версиями языка Fortran являются версии II, IV, 77, 90, 95 и 03, причем до сих пор продолжаются споры о том, какой из языков сегодня используется чаще: Fortran77 или Fortran90.



Первое определение и реализация языка Fortran были выполнены коллективом сотрудников компании IBM под руководством Джона Бэкуса (John Backus): "Мы не знали, чего хотели и как это сделать. Язык просто вырастал". Что они могли знать? До сих пор никто ничего подобного не делал, но постепенно они разработали или открыли основную структуру компилятора (лексический, синтаксический и семантический анализ), а также оптимизацию. И по сей день язык Fortran является лидером в области оптимизации математических вычислений. Среди открытий, сделанных после появления языка Fortran, была система обозначений для определения грамматики: форма Бэкуса–Наура (Backus–Naur Form — BNF). Впервые она была использована в языке Algol60 (см. раздел 22.2.3.1) и в настоящее время используется в большинстве современных языков. Мы использовали вариант формы BNF в нашей грамматике, описанной в главах 6 и 7.

Много позже Джон Бэкус стал основоположником новой области языков программирования (функционального программирования), опирающейся, на математический подход к программированию (в отличие от машинно-ориентированного подхода, основанного на чтении и записи содержимого ячеек памяти). Следует подчеркнуть, что в чистой математике нет понятия присваивания и даже оператора. Вместо этого вы "просто" указываете, что должно быть истинным при данном множестве условий. Некоторые корни функционального программирования уходят в язык Lisp (см. раздел 22.2.2.3), а ряд идей функционального программирования нашел отражение в библиотеке STL (глава 21).

Ссылки

1. Backus, John. "Can Programming Be Liberated from the von Neumann Style?" *Communications of the ACM*, 1977. (Лекция по случаю присуждения премии Тьюринга.)
2. Backus, John. "The History of FORTRAN I, II, and III." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
3. Hutton, Graham. *Programming in Haskell*. Cambridge University Press, 2007. ISBN 0521692695.
4. ISO/IEC 1539. *Programming Languages — Fortran*. (Стандарт Fortran95.)
5. Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0521390222.

22.2.2.2. Язык программирования COBOL

Для программистов, решающих задачи, связанные с бизнесом, язык COBOL (COmmon Business-Oriented Language — язык программирования для коммерческих и деловых задач) был (и кое-где остается до сих пор) тем, чем язык Fortran был (и кое-где остается до сих пор) для программистов, проводящих научные вычисления. Основной акцент в этом языке сделан на обработку данных.

- Копирование.
- Хранение и поиск (хранение записей).
- Вывод на печать (отчеты).

Вычисления в этом языке программирования рассматривались как второстепенные вопросы (что часто было вполне оправданно в тех областях приложений, для которых предназначался язык COBOL). Некоторые даже утверждали (или надеялись), что язык COBOL настолько близок к деловому английскому языку, что менеджеры смогут программировать самостоятельно и программисты скоро станут не нужны. Менеджеры многие годы лелеяли эту надежду, страстно желая сэкономить на программистах. Однако в реальности на это никогда не было даже намека.

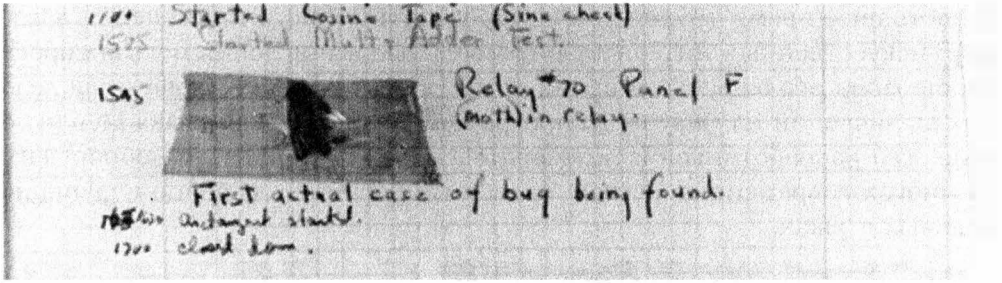
Изначально язык COBOL был разработан комитетом CODASYL в 1959–1960 годах по инициативе Министерства обороны США (U.S. Department of Defense) и группы основных производителей компьютеров для выполнения вычислений, связанных с деловыми и коммерческими задачами. Проект был основан на языке FLOW-MATIC, изобретенном Грейс Хоппер. Одним из ее вкладов в разработку языка было использование синтаксиса, близкого к синтаксису английского языка (в отличие от математических обозначений, принятых в языке Fortran и доминирующих до сих пор). Как

и язык Fortran, а также все успешные языки программирования, COBOL претерпевал непрерывные изменения. Основными версиями были 60, 61, 65, 68, 70, 80, 90 и 04.

Грейс Мюррей Хоппер (Grace Murray Hopper) имела степень доктора философии по математике, полученную в Йельском университете (Yale University). Во время второй мировой войны она работала на военно-морской флот США на самых первых компьютерах. Через несколько лет, проведенных в только что возникшей компьютерной промышленности, она вернулась на службу в военно-морской флот.

> "Контр-адмирал доктор Грейс Мюррей Хоппер (Военно-морской флот США) была замечательной женщиной, достигшей грандиозных результатов в программировании на первых компьютерах. На протяжении всей своей жизни она была лидером в области разработки концепций проектирования программного обеспечения и внесла большой вклад в переход от примитивных методов программирования к использованию сложных компиляторов. Она верила, что лозунг "мы всегда так делали" не всегда является достаточным основанием для того, чтобы ничего не менять".
>
> — Анита Борг (Anita Borg). Из выступления на конференции "Grace Hopper Celebration of Women in Computing", 1994

Грейс Мюррей Хоппер часто называют первой, кто назвал ошибку в компьютере "жучком" (bug). Безусловно, она была одной из первых, кто использовал этот термин и подтвердил это документально.



Жучок был реальным (молью) и повлиял на аппаратное обеспечение самым непосредственным образом. Большинство современных "жучков" гнездятся в программном обеспечении и внешне выглядят не так эффектно.

Ссылки

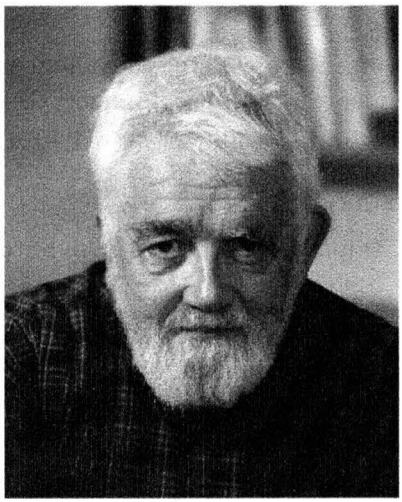
1. Биография Г.М. Хоппер: `http://tergestesoft.com/~eddysworld/hopper.htm`.
2. ISO/IEC 1989:2002. *Information Technology — Programming Languages — COBOL.*
3. Sammet, Jean E. "The Early History of COBOL." *ACM SIGPLAN Notices*, Vol. 13, No. 8, 1978. Special Issue: History of Programming Languages Conference.

22.2.2.3. Язык программирования Lisp

Язык Lisp был разработан в 1958 году Джоном Маккарти (John McCarthy) из Массачусетсского технологического института (MIT) для обработки связанных списков и символьной информации (этим объясняется его название: LISt Processing). Изначально язык Lisp интерпретировался, а не компилировался (во многих случаях это положение не изменилось и в настоящее время). Существуют десятки (а вероятнее всего, сотни) диалектов языка Lisp. Часто говорят, что язык Lisp подразумевает разнообразные реализации. В данный момент наиболее популярными диалектами являются языки Common Lisp и Scheme. Это семейство языков было и остается основой для исследований в области искусственного интеллекта (хотя поставляемые программные продукты часто написаны на языке C или C++). Одним из основных источников вдохновения для создателей языка Lisp было лямбда-исчисление (точнее, его математическое понятие).

Языки Fortran и COBOL были разработаны специально для устранения реальных проблем в соответствующих предметных областях. Разработчики и пользователи языка Lisp больше интересовались собственно программированием и элегантностью программ. Часто их усилия приводили к успеху. Язык Lisp был первым языком, не зависевшим от аппаратного обеспечения, причем его семантика имела математическую основу.

В настоящее время трудно точно определить область применения языка Lisp: искусственный интеллект и символьные вычисления нельзя спроектировать на реальные задачи так четко, как это можно сделать для деловых вычислений или научного программирования. Идеи языка Lisp (и сообщества разработчиков и пользователей языка Lisp) можно обнаружить во многих современных языках программирования, особенно в функциональных языках.



Джон Маккарти получил степень бакалавра по математике в Калифорнийском технологическом институте (California Institute of Technology), а степень доктора философии по математике — в Принстонском университете (Princeton University). Следует подчеркнуть, что среди разработчиков языков программирования много математиков. После периода плодотворной работы в MIT в 1962 году Маккарти переехал в Станфорд, чтобы участвовать в основании лаборатории по изучению искусственного интеллекта (Stanford AI lab). Ему приписывают изобретение термина *искусственный интеллект* (artificial intelligence), а также множество достижений в этой области.

Ссылки

1. Abelson, Harold, and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996. ISBN 0262011530.
2. ANSI INCITS 226-1994 (formerly ANSI X3.226:1994). *American National Standard for Programming Language — Common LISP*.
3. McCarthy, John. "History of LISP." *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
4. Steele, Guy L. Jr. *Common Lisp: The Language*. Digital Press, 1990. ISBN 1555580416.

5. Steele, Guy L. Jr., and Richard Gabriel. "The Evolution of Lisp". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

22.2.3. Семейство языков Algol

В конце 1950-х годов многие почувствовали, что программирование стало слишком сложным, специализированным и слишком ненаучным. Возникло убеждение, что языки программирования излишне разнообразны и что их следует объединить в один язык без больших потерь для общности на основе фундаментальных принципов. Эта идея носилась в воздухе, когда группа людей собралась вместе под эгидой IFIP (International Federation of Information Processing — Международная федерация по обработке информации) и всего за несколько лет создала новый язык, который совершил революцию в области программирования. Большинство современных языков, включая язык C++, обязаны своим существованием этому проекту.

22.2.3.1. Язык программирования Algol60

"Алгоритмический язык" ("ALGOrithmic Language" — Algol), ставший результатом работы группы IFIP 2.1, открыл новые концепции современных языков программирования.

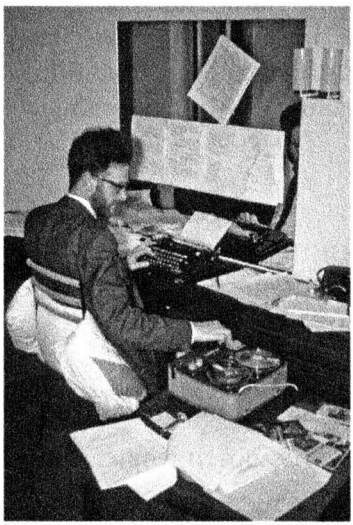
- Контекст лексического анализа.
- Использование грамматики для определения языка.
- Четкое разделение синтаксических и семантических правил.
- Четкое разделение определения языка и его реализации.
- Систематическое использование типов (статических, т.е. типов времени компиляции).
- Непосредственная поддержка структурного программирования.

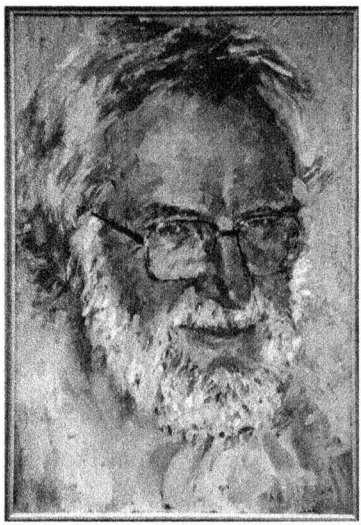
Само понятие "язык программирования общего назначения" пришло вместе с языком Algol. До того времени языки предназначались для научных вычислений (например, Fortran), деловых расчетов (например, COBOL), обработки списков (например, Lisp), моделирования и т.д. Из всех перечисленных язык Algol60 ближе всех к языку Fortran.

К сожалению, язык Algol60 так и не вышел за пределы академической среды. Многим он казался слишком странным. Программисты, предпочитавшие Fortran, утверждали, что программы на Algol60 работают слишком медленно, программисты, работавшие на языке Cobol, говорили, что Algol60 не поддерживает обработку деловой информации, программисты, работавшие на языке Lisp, говорили, что Algol60 недостаточно гибок,

большинство остальных людей (включая менеджеров, управляющих инвестициями в разработку программного обеспечения) считали его слишком академичным, и, наконец, многие американцы называли его "слишком европейским". Большинство критических замечаний было справедливым. Например, в отчете о языке Algol60 не был определен ни один механизм ввода-вывода! Однако эти замечания можно адресовать большинству языков программирования того времени, ведь именно язык Algol установил новые стандарты во многих областях программирования.

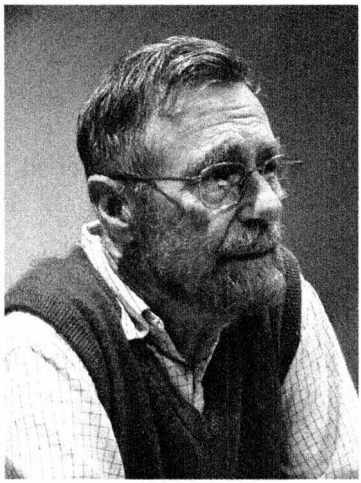
Одна из проблем, связанных с языком Algol60, заключалась в том, что никто не знал, как его реализовать. Эта проблема была решена группой программистов под руководством Питера Наура (Peter Naur), редактора отчета по языку Algol60, и Эдсгером Дейкстрой (Edsger Dijkstra).





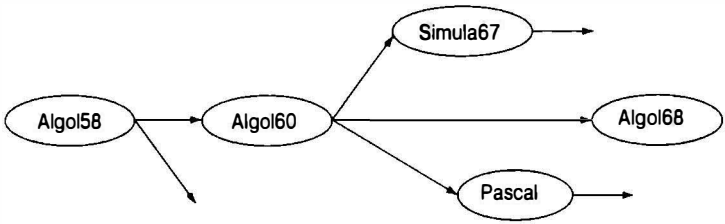
Питер Наур получил образование астронома в Копенгагенском университете (University of Copenhagen) и работал в Техническом университете Копенгагена (Technical University of Copenhagen — DTH), а также на датскую компанию Regnecentralen, производившую компьютеры. Программирование он изучал в 1950–1951 годах в компьютерной лаборатории в Кембридже (Computer Laboratory in Cambridge), поскольку в то время в Дании не было компьютеров, а позднее сделал блестящую академическую и производственную карьеру. Он был одним из авторов формы Бэкуса–Наура (BNF), использовавшейся для описания грамматик, а также одним из первых поборников формальных рассуждений о программах (Бьярне Страуструп впервые — приблизительно в 1971 году — узнал об использовании инвариантов из статей Питера Наура). Наур последовательно придерживался вдумчивого подхода к вычислениям, всегда учитывая человеческий фактор в программировании. Его поздние работы носили философский характер (хотя он считал традиционную академическую философию совер-

шенной чепухой). Он стал первым профессором даталогии в Копенгагенском университете (датский термин "даталогия" (datalogi) точнее всего переводится как "информатика"; Питер Наур ненавидел термин "компьютерные науки" (computer science), считая его абсолютно неправильным, так как вычисления — это не наука о компьютерах).



Эдсгер Дейкстра (Edsger Dijkstra) — еще один великий ученый в области компьютерных наук. Он изучал физику в Лейдене, но свои первые работы выполнил в Математическом центре (Mathematisch Centrum) в Амстердаме. Позднее он работал в нескольких местах, включая Эйндховенский технологический университет (Eindhoven University of Technology), компанию Burroughs Corporation и университет Техаса в Остине (University of Texas (Austin)). Он не только плодотворно работал над языком Algol, но и стал пионером и горячим сторонником использования математической логики в программировании и теории алгоритмов, а также одним из разработчиков и конструкторов операционной системы THE — одной из первых операционных систем, систематически использующих параллелизм. Название "THE" является аббревиатурой от "Technische Hogeschool Eindhoven" — университета, в котором Эдсгер Дейкстра работал в то время. Вероятно, самой известной стала его статья "Go-To Statement Considered Harmful", в которой он убедительно продемонстрировал опасность неструктурированных потоков управления.

Генеалогическое дерево языка Algol выглядит впечатляюще.



Обратите внимание на языки Simula67 и Pascal. Они являются предшественниками многих (вероятно, большинства) современных языков.

Ссылки

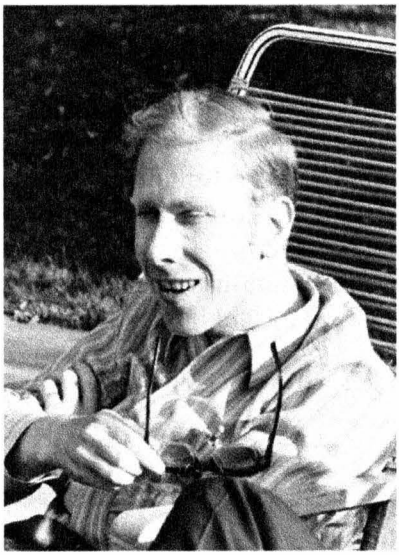
1. Dijkstra, EdsgerW. "Algol 60 Translation: An Algol 60 Translator for the x1 and Making a Translator for Algol 60". Report MR 35/61. Mathematisch Centrum (Amsterdam), 1961.
2. Dijkstra, Edsger. "Go-To Statement Considered Harmful". *Communications of the ACM*, Vol. 11 No. 3, 1968.
3. Lindsey, C. H. "The History of Algol-68". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
4. Naur, Peter, ed. "Revised Report on the Algorithmic Language Algol 60". A/S Regnecentralen (Copenhagen), 1964.
5. Naur, Peter. "Proof of Algorithms by General Snapshots". BIT, Vol. 6, 1966, p. 310–316. (Вероятно, первая статья о том, как доказать правильность программы.)
6. Naur, Peter. "The European Side of the Last Phase of the Development of ALGOL 60". *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
7. Perlis, Alan J. "The American Side of the Development of Algol". *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
8. van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, eds. *Revised Report on the Algorithmic Language Algol 68* (Sept. 1973). Springer-Verlag, 1976.

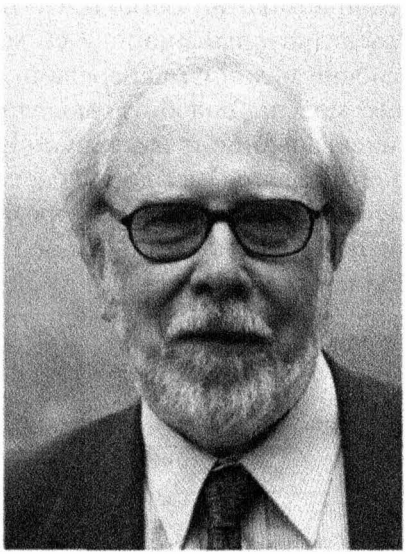
22.2.3.2. Язык программирования Pascal

Язык Algol68, указанный на генеалогическом дереве семейства языков Algol, был крупным и амбициозным проектом. Подобно языку Algol60, он был разработан комитетом по языку Algol (рабочей группой IFIP 2.1), но его реализация затянулась до бесконечности, и многие просто потеряли терпение и стали сомневаться, что из этого проекта получится что-нибудь полезное. Один из членов комитета по языку, Никлаус Вирт (Niklaus Wirth), решил разработать и реализовать собственный язык, являющийся наследником языка Algol. В противоположность языку Algol68, его язык, названный Pascal, был упрощенным вариантом языка Algol60.

Разработка языка Pascal была завершена в 1970 году, и в результате он действительно оказался простым и достаточно гибким. Часто утверждают, что он был предназначен только для преподавания, но в ранних статьях

его представляли как альтернативу языка Fortran, предназначенную для суперкомпьютеров того времени. Язык Pascal действительно несложно выучить, и после появления легко переносимых реализаций он стал популярным языком, который использовали для преподавания программирования, но составить конкуренции языку Fortran он не смог.





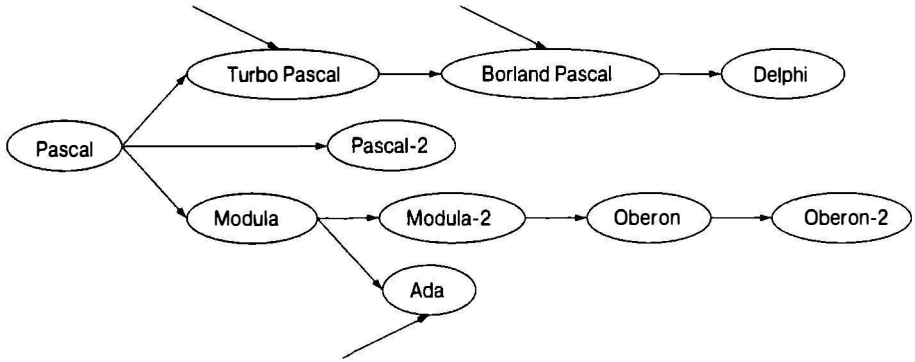
Язык Pascal создан профессором Никлаусом Виртом (Niklaus Wirth) из Технического университета Швейцарии в Цюрихе (Technical University of Switzerland in Zurich — ETH). Выше приведены его фотографии, сделанные в 1969 и 2004 годах. Он получил степень доктора философии (по электротехнике и компьютерным наукам) в Калифорнийском университете в Беркли (University of California at Berkeley) и на протяжении всей своей долгой жизни поддерживал связь с Калифорнией. Профессор Вирт был наиболее полным воплощением идеала профессионального разработчика языков программирования. На протяжении двадцати пяти лет от разработал и реализовал следующие языки программирования.

- Algol W.
- PL/360.
- Euler.
- Pascal.
- Modula.
- Modula-2.
- Oberon.
- Oberon-2.
- Lola (язык описания аппаратного обеспечения).

Никлаус Вирт описывал свою деятельность как бесконечный поиск простоты. Его работа оказала наибольшее влияние на программирование. Изучение этого ряда языков программирования представляет собой чрезвычайно интересное занятие. Профессор Вирт — единственный человек,

представивший на конференции HOPL (History of Programming Languages) два языка программирования.

В итоге оказалось, что язык Pascal слишком простой и негибкий, чтобы найти широкое промышленное применение. В 1980-х годах его спасла от забвения работа Андерса Хейльсберга (Anders Hejlsberg) — одного из трех основателей компании Borland. Он первым разработал и реализовал язык Turbo Pascal (который, наряду со многими другими возможностями, воплотил гибкие механизмы передачи аргументов), а позднее добавил в него объектную модель, подобную модели языка C++ (допускающую лишь одиночное наследование и имеющую прекрасный модульный механизм). Он получил образование в Техническом университете Копенгагена (Technical University in Copenhagen), в котором Питер Наур иногда читал лекции, — мир, как известно, тесен. Позднее Андерс Хейльсберг разработал язык Delphi для компании Borland и язык C# для компании Microsoft. Упрощенное генеалогическое дерево семейства языков Pascal показано ниже.



Ссылки

1. Borland/Turbo Pascal. `http://en.wikipedia.org/wiki/Turbo_Pascal`.
2. Hejlsberg, Anders, ScottWiltamuth, and Peter Golde. *The C# Programming Language, Second Edition*. Microsoft .NET Development Series. ISBN 0321334434.
3. Wirth, Niklaus. "The Programming Language Pascal". *Acta Informatics*, Vol. 1 Fasc 1, 1971.
4. Wirth, Niklaus. "Design and Implementation of Modula". *Software — Practice and Experience*, Vol. 7 No. 1, 1977.
5. Wirth, Niklaus. "Recollections about the Development of Pascal". Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

6. Wirth, Niklaus. *Modula-2 and Oberon*. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. `http://portal.acm.org/toc.cfm?id=1238844`.

22.2.3.3. Язык программирования Ada

Язык программирования Ada предназначался для программных нужд Министерства обороны США. В частности, он должен был стать языком, обеспечивающим создание удобочитаемого и легко сопровождаемого кода для встроенных систем программирования. Его наиболее очевидными предками являются языки Pascal и Simula (см. разделы 22.2.3.2 и 22.2.4). Лидером группы разработчиков языка Ada был Жан Ишби (Jean Ichbiah), который ранее был председателем группы Simula Users' Group. При разработке языка Ada основное внимание было уделено

- абстракции данных (но без наследования до 1995 года);
- строгой проверке статических типов;
- непосредственной языковой поддержке параллелизма.

Целью проекта Ada было воплощение принципов разработки программного обеспечения. В силу этого Министерство обороны разрабатывало не язык, а сам процесс проектирования языка. В этом процессе принимали участие огромное число людей и организаций, которые конкурировали друг с другом за создание наилучшей спецификации и наилучшего языка, воплощающего идеи победившей спецификации. Этим огромным двадцатилетним проектом (1975–1998) с 1980 года управлял отдел AJPO (Ada Joint Program Office).

В 1979 году язык получил название в честь леди Аугусты Ады Лавлейс (Augusta Ada Lovelace), дочери поэта лорда Байрона (Byron). Леди Лавлейс можно назвать первой программисткой современности (если немного расширить понятие современности), поскольку она сотрудничала с Чарльзом Бэббиджем (Charles Babbage), Лукасианским профессором[1] математики в Кембридже (занимавшим должность, которую ранее занимал Ньютон!) в процессе создания революционного механического компьютера в 1840-х годах. К сожалению, на практике машина Бэббиджа себя не оправдала.

---

[1] Эта должность была учреждена Генри Лукасом (Henry Lucas), членом совета Кембриджа. После кончины он завещал университету свое состояние и библиотеку (между прочим, с автографом Галилея "Диалоги", 1632), и на эти деньги была учреждена должность "Лукасианский профессор" с окладом 100 фунтов в год. С тех пор это одна из самых престижных должностей мира. Этого звания среди прочих были удостоены такие ученые, как Исаак Ньютон, Джордж Стокс, Поль Дирак и Стивен Хокинг. — *Примеч. пер.*

Благодаря продуманному процессу разработки язык Ada считается наилучшим языком, разработанным комитетом. Жан Ишби из французской компании, лидер победившего коллектива разработчиков, это решительно отрицал. Однако я подозреваю (после дискуссии с ним), что он мог бы разработать еще более хороший язык, если бы не был ограничен заданными условиями.

Министерство обороны США много лет предписывало использовать язык Ada в военных приложениях, что в итоге выразилось в афоризме "Язык Ada — это не просто хорошая идея, это — закон!" Сначала язык Ada просто "настоятельно рекомендовался" к использованию, но со временем многим проектировщикам было прямо запрещено использовать другие языки программирования (как правило, C++), а Конгресс США принял закон, требующий, чтобы в большинстве военных приложений использовался только язык Ada. Под влиянием рыночных и технических реалий этот закон был впоследствии отменен. Таким образом, Бьярне Страуструп был одним из очень немногих людей, чья работа была запрещена Конгрессом США.

Словом, язык Ada оказался намного лучше своей репутации. Мы подозреваем, что, если бы Министерство обороны США не было таким неуклюжим в его использовании и точно придерживалось принципов, положенных в его основу (стандарты процессов проектирования приложений, инструменты разработки программного обеспечения, документация и т.д.), его успех был бы более ярким. В настоящее время язык Ada играет важную роль в аэрокосмических приложениях и областях, связанных с разработкой аналогичных встроенных систем.

Язык Ada стал военным стандартом в 1980 году, стандарт ANSI был принят в 1983 году (первая реализация появилась в 1983 году — через три года *после* издания первого стандарта!), а стандарт ISO — в 1987 году.

Стандарт ISO был сильно пересмотрен (конечно, сравнительно) в издании 1995 года. Включение в стандарт значительных усовершенствований повысило гибкость механизмов параллелизма и поддержки наследования.

Ссылки

1. Barnes, John. *Programming in Ada 2005*. Addison-Wesley, 2006. ISBN 0321340787.
2. Consolidated Ada Reference Manual, consisting of the international standard (ISO/IEC 8652:1995). *Information Technology — Programming Languages — Ada, as updated by changes from Technical Corrigendum 1* (ISO/IEC 8652:1995:TC1:2000).
3. Информационная страница языка Ada: `www.adaic.org/`.
4. Whitaker, William A. ADA — *The Project: The DoD High Order Language Working Group*. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

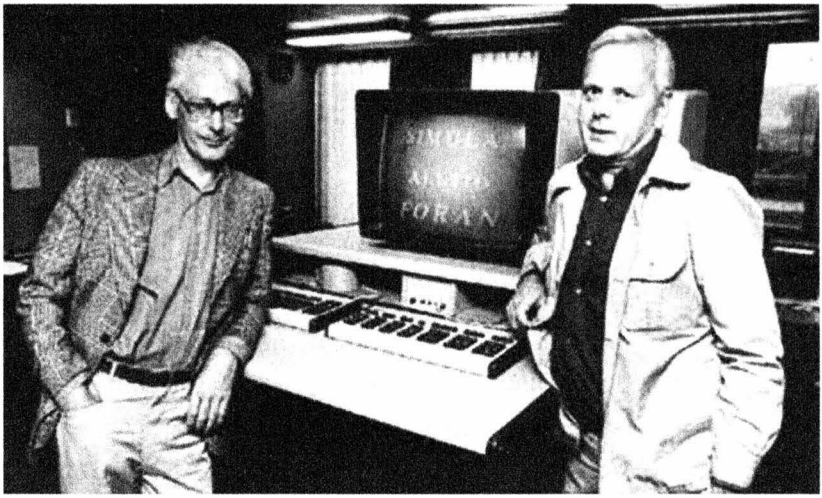
22.2.4. Язык программирования Simula

Язык Simula был разработан в первой половине 1960-х годов Кристеном Нюгордом (Kristen Nygaard) и Оле-Йоханом Далем (Ole-Johan Dahl) в Норвежском вычислительном центре (Norwegian Computing Center) и университете Осло (Oslo University). Язык Simula, несомненно, принадлежит семейству языков Algol. Фактически язык Simula является практически полным надмножеством языка Algol60. Однако мы уделили особое внимание языку Simula потому, что он является источником большинства фундаментальных идей, которые сегодня называют объектно-ориентированным программированием. Он был первым языком, в котором реализованы наследование и виртуальные функции. Слова *class* для пользовательского типа и *virtual* для функции, которую можно перекрыть и вызвать с помощью интерфейса базового класса, пришли в C++ из языка Simula.

Вклад языка Simula не ограничен языковыми возможностями. Он состоит в явно выраженном понятии объектно-ориентированного проектирования, основанного на идее моделирования реальных явлений в коде программы:

- представление идей в виде классов и объектов классов;
- представление иерархических отношений в виде иерархии классов (наследование).

Таким образом, программа становится множеством взаимодействующих объектов, а не монолитом.

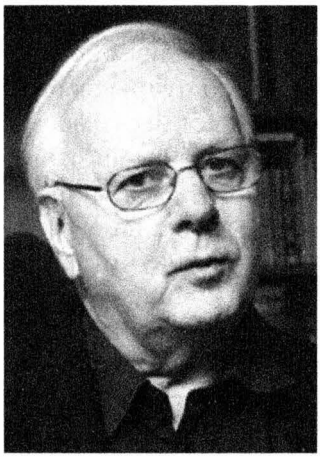


Кристен Нюгорд — один из создателей языка Simula67 (вместе с Оле-Йоханом Далем, на фото слева в очках) — был энергичным и щедрым гигантом (в том числе и по росту). Он посеял семена фундаментальных идей объектно-ориентированного программирования и проектирования, особенно наследования, и неотступно придерживался их на протяжении десятилетий. Его никогда не устраивали простые, краткие и близорукие ответы. Социальные вопросы также волновали его на протяжении десятков лет. Он искренне выступал против вступления Норвегии в Европейский Союз, видя в этом опасность излишней централизации, бюрократизации и пренебрежения интересами маленькой страны, находящейся на дальнем краю ЕС. В середине 1970-х годов Кристен Нюгорд посвящал значительное время работе на факультете компьютерных наук в Университете Аархуса (University of Aarhus) в Дании, где в это время Бьярне Страуструп проходил обучение по программе магистров.

Магистерскую степень по математике Кристен Нюгорд получил в Университете Осло (University of Oslo). Он умер в 2002 году, всего через месяц после того, как (вместе с другом всей своей жизни Оле-Йоханом Далем) получил премию Тьюринга — наивысший знак почета, которым Ассоциация по вычислительной технике (Association for Computing Machiner — ACM) отмечает выдающихся ученых в области компьютерных наук.

Оле-Йохан Даль был более традиционным академическим ученым. Его очень интересовали спецификации языков и формальные методы. В 1968 году он стал первым профессором по информатике (компьютерным наукам) в университете Осло.

В августе 2000 года король Норвегии объявил Даля и Нюгорда командорами ордена Святого Олафа (Commanders of the Order of Saint Olav). Все-таки есть пророки в своем отечестве!



Ссылки

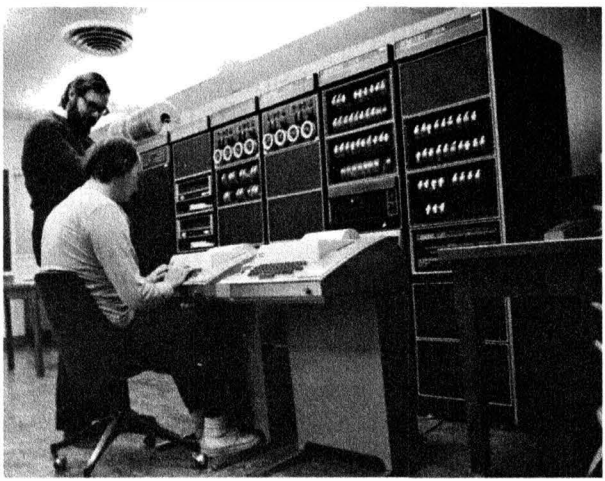
1. Birtwistle, G., O-J. Dahl, B. Myhrhaug, and K. Nygaard: *SIMULA Begin*. Studentlitteratur (Lund. Sweden), 1979. ISBN 9144062125.
2. Holmevik, J. R. "Compiling SIMULA: A Historical Study of Technological Genesis". *IEEE Annals of the History of Computing*, Vol. 16 No. 4, 1994, p. 25–37.
3. Krogdahl, S. "The Birth of Simula". Proceedings of the HiNC 1 Conference in Trondheim, June 2003 (IFIP WG 9.7, in cooperation with IFIP TC 3).
4. Nygaard, Kristen, and Ole-Johan Dahl. "The Development of the SIMULA Languages". *ACM SIGPLAN Notices*, Vol. 13 No. 8, 1978. Special Issue: History of Programming Languages Conference.
5. SIMULA Standard. *DATA processing — Programming languages — SIMULA*. Swedish Standard, Stockholm, Sweden (1987). ISBN 9171622349.

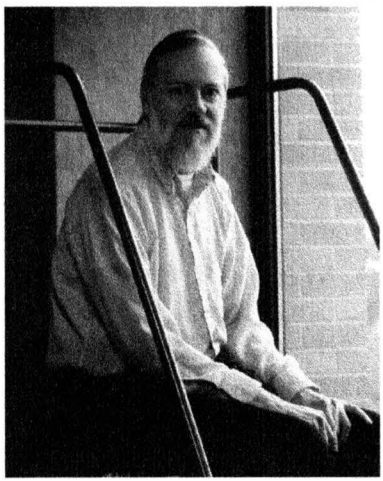
22.2.5. Язык программирования C

В 1970-м году считалось, что серьезное системное программирование — в частности, реализация операционной системы — должно выполняться в ассемблерном коде и не может быть переносимым. Это очень напоминало ситуацию, сложившуюся в научном программировании перед появлением языка Fortran. Несколько индивидуумов и групп бросили вызов этой ортодоксальной точке зрения. В долгосрочной перспективе язык программирования C (глава 27) оказался наилучшим результатом этих усилий.

Деннис Ритчи (Dennis Ritchie) разработал и реализовал язык программирования C в Исследовательском центре по компьютерным наукам (Computer Science Research Center) компании Bell Telephone Laboratories в Мюррей-Хилл, штат Нью-Джерси (Murray Hill, New Jersey). Прелесть языка C

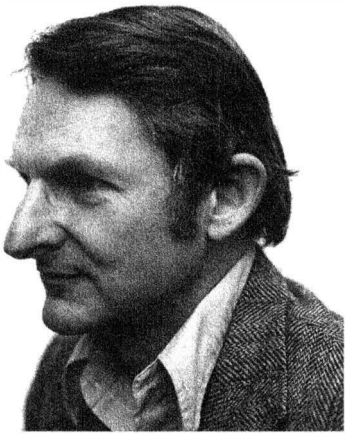
в том, что он был преднамеренно простым языком программирования, позволявшим близко подойти к фундаментальным аспектам аппаратного обеспечения. Большинство усложнений (которые в основном были заимствованы у языка C++ для обеспечения совместимости) было внесено позднее и часто вопреки желанию Денниса Ритчи. Частично успех языка C объясняется его широкой доступностью, но его реальная сила проявлялась в непосредственном отображении свойств языка на свойства аппаратного обеспечения (см. разделы 25.4–25.5). Деннис Ритчи лаконично описывал язык C как "строго типизированный язык со слабым механизмом проверки"; иначе говоря, язык C имел систему статических (распознаваемых на этапе компиляции) типов, а программа, использовавшая объект вопреки его определению, считалась неверной. Однако компилятор языка C не мог выполнять такие проверки. Это имело смысл, поскольку компилятор языка C должен был работать при наличии всего 48 Кбайт памяти. Вскоре язык C вошел в практику, и программисты написали программу *lint*, которая отдельно от компилятора проверяла соответствие кода системе типов.



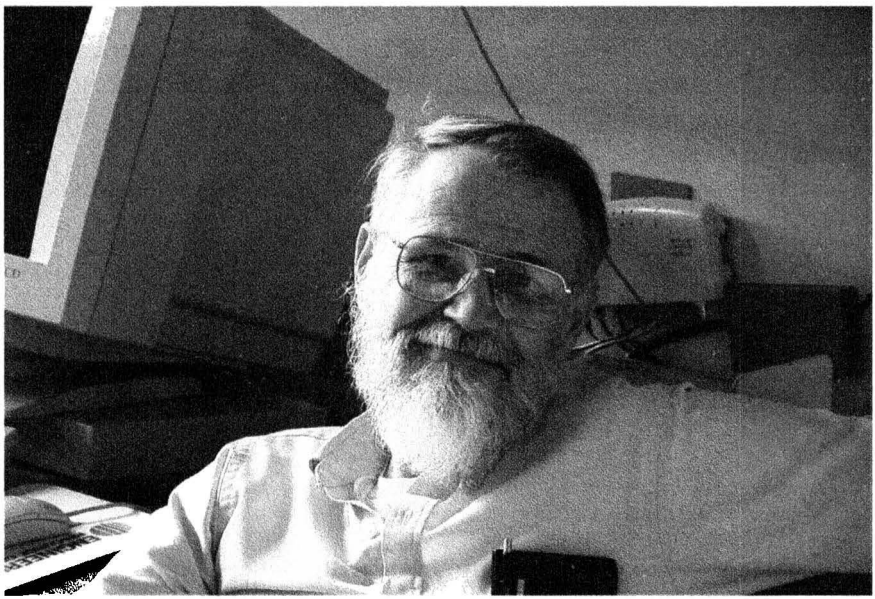


Кен Томпсон (Ken Thompson) и Деннис Ритчи стали авторами системы Unix, возможно, наиболее влиятельной операционной системы за все времена. Язык C ассоциировался и по-прежнему ассоциируется с операционной системой Unix, а через нее — с системой Linux и движением за открытый код.

Деннис Ритчи вышел на пенсию из компании Lucent Bell Labs. На протяжении сорока лет он работал в Исследовательском центре по компьютерным наукам компании Bell Telephone. Он закончил Гарвардский университет (Harvard University) по специальности "физика", степень доктора философии по прикладной математике им так и не была получена, поскольку он то ли забыл, то ли отказался внести небольшой (60 долларов) регистрационный взнос.



В 1974–1979 годах на развитие и адаптацию языка C++ оказали влияние многие люди из компании Bell Labs. В частности, Дуг Мак-Илрой (Doug McIlroy) был всеобщим любимцем, критиком, собеседником и генератором идей. Он оказал влияние не только на языки C и C++, но и на операционную систему Unix, а также на многое другое.



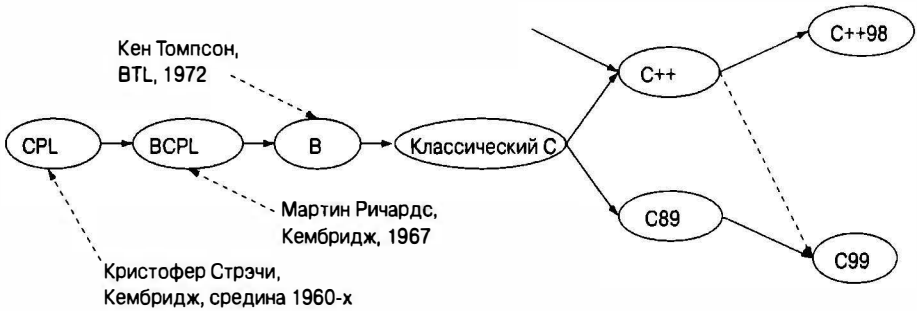
Брайан Керниган (Brian Kernighan) — выдающийся программист и писатель. Его программы и проза — образцы ясности. Стиль этой книги частично объясняется подражанием его шедевру — учебнику *The C Programming Language* (известному как K&R по первым буквам фамилий его авторов — Брайана Кернигана и Денниса Ритчи).

Мало выдвинуть хорошие идеи. Для того чтобы польза была ощутимой, их необходимо выразить в простейшей форме и сформулировать столь ясно, чтобы вас поняло много людей. Многословность — злейший враг ясности; кроме него, следует упомянуть запутанное изложение и излишнюю абстрактность. Пуристы часто насмехаются над результатами такой популяризации и предпочитают "оригинальные результаты", представленные в форме, доступной только экспертам. Мы к пуристам не относимся: новичкам трудно усвоить нетривиальные, но ценные идеи, хотя это необходимо для их профессионального роста и общества в целом.

В течение многих лет Брайан Керниган участвовал во многих важных программистских и издательских проектах. В качестве примера можно назвать язык AWK — один из первых языков сценариев, получивший название по инициалам своих авторов (Aho, Weinberger и Kernighan), а также AMPL — (A Mathematical Programming Language — язык математического программирования).

В настоящее время Брайан Керниган — профессор Принстонского университета (Princeton University); он превосходный преподаватель, ясно излагающий сложные темы. Более тридцати лет он работал в Исследовательском центре по компьютерным наукам компании Bell Telephone. Позднее компания Bell Labs стала называться AT&T Bell Labs, а затем разделилась на компании AT&T Labs и Lucent Bell Labs. Брайан Керниган закончил Университет Торонто (University of Toronto) по специальности "Физика"; степень доктора философии по электротехнике он получил в Принстонском университете.

Генеалогическое дерево семейства языка C представлено ниже.



Корни языка C уходят в так никогда и не завершившийся проект по разработке языка CPL в Англии, язык BCPL (Basic CPL), разработанный сотрудником Кембриджского университета (Cambridge University) Мартином Ричардсом (Martin Richards) во время его посещения Массачусетсского технологического института (MIT), а также в интерпретируемый язык B, созданный Кеном Томпсоном. Позднее язык C был стандартизован ANSI и ISO и подвергся сильному влиянию языка C++ (например, в нем появились проверка аргументов функций и ключевое слово `const`).

Разработка языка CPL была целью совместного проекта Кембриджского университета и Имперского колледжа Лондона (Imperial College). Изначально планировалось выполнить проект в Кембридже, поэтому буква "C" официально означает слово "Cambridge". Когда партнером в проекте стал Имперский колледж, официальным объяснением буквы "C" стало слово "Combined" ("совместный"). На самом деле (по крайней мере так нам рассказывали) его всегда связывали с именем Christopher в честь Кристофера Стрэчи (Christopher Strachey), основного разработчика языка CPL.

Ссылки

1. Начальные страницы Брайана Кернигана: `http://cm.bell-labs.com/cm/cs/who/bwk` и `www.cs.princeton.edu/~bwk/`.
2. Начальная страница Денниса Ритчи: `http://cm.bell-labs.com/cm/cs/who/dmr`.
3. ISO/IEIC 9899:1999. *Programming Languages — C*. (Стандарт языка программирования C.)
4. Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978. Second Edition, 1989. ISBN 0131103628.[2]
5. Список сотрудников Исследовательского центра по компьютерным наукам компании Bell Labs: `http://cm.bell-labs.com/cm/cs/alumni.html`.
6. Ritchards, Martin. BCPL — *The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
7. Ritchie, Dennis. The Development of the C Programming Language. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
8. Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

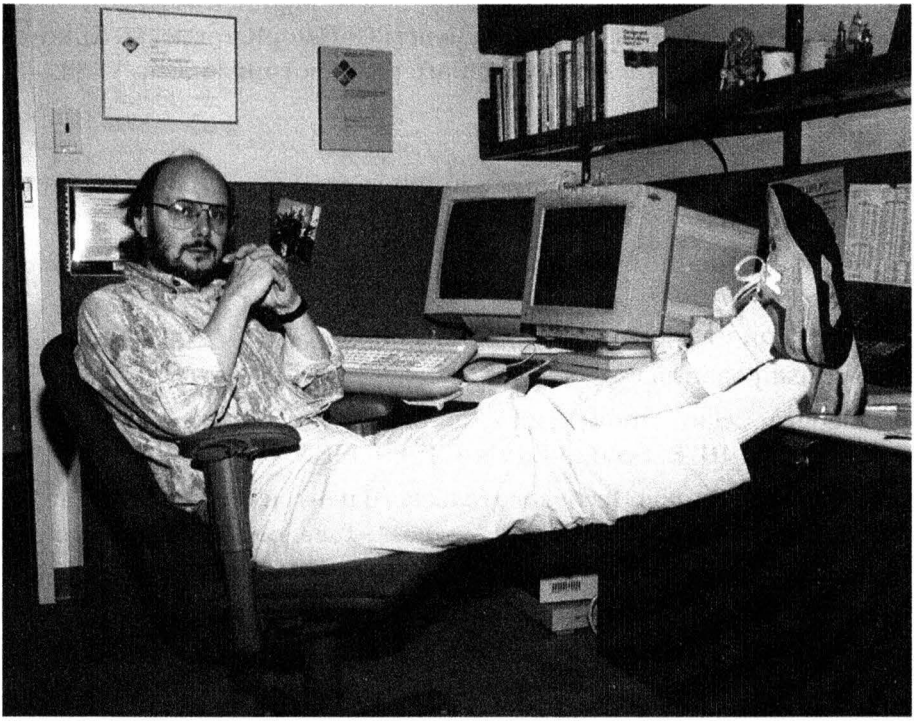
22.2.6. Язык программирования C++

Язык C++ — универсальный язык программирования с уклоном в системное программирование. Перечислим его основные свойства.

- Он лучше языка C.
- Он поддерживает абстракцию данных.
- Он поддерживает объектно-ориентированное программирование.
- Он поддерживает обобщенное программирование.

[2] Имеется русский перевод: Брайан У. Керниган, Деннис М. Ритчи. *Язык программирования C, 2-е издание* — М.: ООО "И.Д. Вильямс", 2006.

Язык C++ был разработан и реализован Бьярне Страуструпом из Исследовательского центра по компьютерным наукам компании Bell Telephone Laboratories в Мюррей-Хилл (Murray Hill), штат Нью-Джерси (New Jersey), где работали также Деннис Ритчи, Брайан Керниган, Кен Томпсон, Дуг Мак-Илрой и другие гиганты системы Unix.



Бьярне Страуструп получил степень магистра по математике и компьютерным наукам в своем родном городе Эрхусе (Erhus), Дания. Затем он переехал в Кембридж (Cambridge), где получил степень доктора философии по компьютерным наукам, работая с Дэвидом Уилером (David Wheeler). Цель создания языка C+ заключалась в следующем.

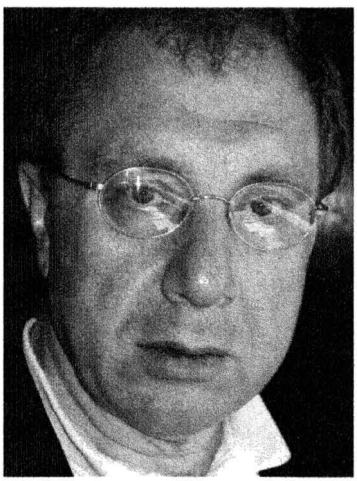
- Сделать методы абстрагирования доступными и управляемыми в рамках широко распространенных проектов.
- Внедрить объектно-ориентированное и обобщенное программирование в прикладные области, где основным критерием успеха является эффективность.

До появления языка C++ эти методы (часто необоснованно объединяемые под общим названием "объектно-ориентированное программирование") были практически неизвестны в индустрии. Как в научном программировании до появления языка Fortran, так и в системном программировании до появления языка C считалось, что эти технологии слишком дорогие для

использования в реальных приложениях и слишком сложные для обычных программистов.

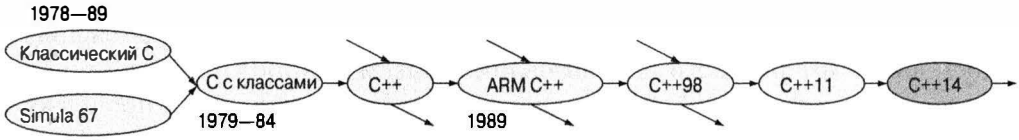
Работа над языком C++ началась в 1979 году, а в 1985 году он был выпущен для коммерческого использования. Затем Бьярне Страуструп и его друзья из компании Bell Labs и нескольких других организаций продолжали совершенствовать язык C++, и в 1990 году началась официальная процедура его стандартизации. С тех пор определение языка C++ было сначала разработано ANSI (Национальный институт стандартизации США), а с 1991 года — ISO (Международная организация по стандартизации). Бьярне Страуструп играл главную роль в этом процессе, занимая должность председателя ключевой подгруппы, ответственной за создание новых свойств языка. Первый международный стандарт (C++98) был ратифицирован в 1998 году, а второй в 2011 (C++11). Следующим стандартом ISO будет C++14, а после него очередным стандартом (то, что некоторые именуют C++1y) может стать C++17.

Наиболее значительным событием в истории языка C++ спустя десять лет после его появления стала стандартная библиотека контейнеров и алгоритмов — STL. Она стала результатом многолетней работы, в основном под руководством Александра Степанова (Alexander Stepanov), направленной на создание как можно более универсального и эффективного программного обеспечения и вдохновляемой красотой и полезностью математики.



Алекс Степанов — изобретатель библиотеки STL и пионер обобщенного программирования. Он закончил Московский государственный университет и работал в области робототехники и алгоритмов, используя разные языки программирования (включая Ada, Scheme и C++). С 1979 года он работал в академических организациях США, а также в промышленных компаниях, таких как GE Labs, AT&T Bell Labs, Hewlett-Packard, Silicon Graphics и Adobe.

Генеалогическое дерево языка C++ приведено ниже.



Язык "С с классами" был создан Бьярне Страуструпом как результат синтеза идей языков С и Simula. Этот язык умер сразу же после реализации его преемника — языка C++.

Обсуждение языков программирования часто сосредоточено на их элегантности и новых свойствах. Однако языки С и C++ стали самыми успешными языками программирования за всю историю компьютерных технологий не поэтому: их сила заключается в гибкости, производительности и устойчивости. Важные программные системы существуют несколько десятилетий, часто исчерпывая свои аппаратные ресурсы и подвергаясь совершенно неожиданным изменениям. Языки С и C++ смогли преуспеть в этой среде. Мы очень любим изречение Денниса Ритчи: "Одни языки люди разрабатывали, чтобы доказать свою правоту, а другие — чтобы решить задачу". Язык С относится ко второй категории языков. Бьярне Страуструп любит говорить "Даже я знаю, как разработать язык, более красивый чем C++". Цель языка C++, как и языка С, — не абстрактная красота (хотя мы очень ее ценим), а полезность.

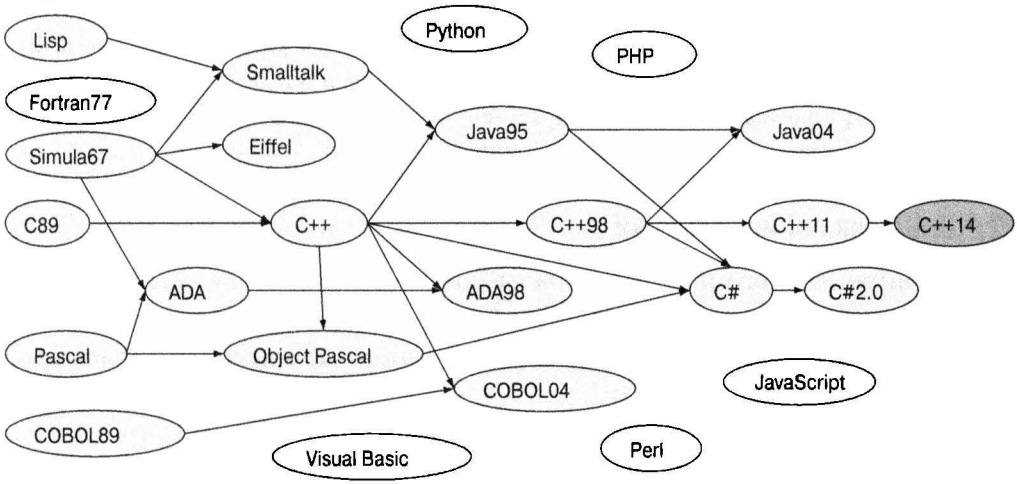
Ссылки

1. Публикации Александра Степанова: `www.stepanovpapers.com`.
2. Начальная страница Бьярне Страуструпа: `www.stroustrup.com`.
3. ISO/IEC 14882:2011. *Programming Languages — C++*. (Стандарт C++.)
4. Stroustrup, Bjarne. *A History of C++: 1979–1991*. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). ACM SIGPLAN Notices, Vol. 28 No. 3, 1993.
5. Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
6. Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 978-0321563842.
7. Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.
8. Stroustrup, Bjarne. *C and C++: Siblings*; *C and C++: A Case for Compatibility*; and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July, Aug., and Sept. 2002.
9. Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991– 2006". *Proceedings of the Third ACM SIGPLAN Conference on the*

History of Programming Languages (HOPL-III). San Diego, CA, 2007. `http://portal.acm.org/toc.cfm?id=1238844`.

22.2.7. Современное состояние дел

Как в настоящее время используются языки программирования и для чего? На этот вопрос действительно трудно ответить. Генеалогическое дерево современных языков, даже в сокращенном виде, слишком перегружено и запутано.



Фактически большинство статистических данных, найденных в вебе (или в других местах), ничуть не лучше обычных слухов, поскольку они пытаются оценить явления, слабо связанные с интенсивностью использования, например количество упоминаний в вебе какого-нибудь языка программирования, продаж компиляторов, академических статей, продаж книг и т.д. Эти показатели завышают популярность новых языков программирования по сравнению со старыми. Как бы то ни было, кто такой программист? Человек, использующий язык программирования каждый день? А может быть, студент, пишущий маленькие программы с целью изучения языка? А может быть, профессор, только рассуждающий о программировании? А может быть, физик, создающий программы почти каждый год? Является ли профессиональным программистом тот, кто — про определению — использует несколько языков программирования каждую неделю несколько раз или только один раз? Разные статистические показатели будут приводить к разным ответам.

Тем не менее мы обязаны ответить на этот вопрос, поскольку в 2014 году в мире было около десяти миллионов профессиональных программистов. Об этом свидетельствуют отчет компании IDC (специализирующейся на сборе данных), дискуссии с издателями и поставщиками компиляторов,

а также различные источники в вебе. Можете с нами спорить, но нам точно известно, что от одного до ста миллионов человек хотя бы наполовину подходят под разумное определение программиста. Какие языки они используют? Вероятно (просто вероятно), что более 90% их программ написано на языках Ada, C, C++, C#, COBOL, Fortran, Java, PERL, PHP, Python и Visual Basic.

Кроме упомянутых выше языков, мы могли бы перечислять десятки и даже сотни названий. Однако мы считаем необходимым упомянуть только интересные или важные языки. Если вам нужна дополнительная информация, можете найти ее самостоятельно. Профессионалы знают несколько языков и при необходимости могут изучить новый. Не существует единственного правильного языка для всех людей и для всех приложений. На самом деле все основные системы, которые нам известны, используют несколько языков.

22.2.8. Источники информации

Описание каждого языка содержит собственный список ссылок. Ниже приведены ссылки для нескольких языков.

Страницы и фотографии разработчиков языков программирования

`www.angelfire.com/tx4/cus/people/`.

Несколько примеров языков программирования

`http://dmoz.org/Computers/Programming/Languages/`.

Учебники

1. Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.
2. Sebesta, Robert W. *Concepts of Programming Languages*. Addison-Wesley, 2003. ISBN 0321193628.

Книги по истории языков программирования

1. Bergin, T. J., and R. G. Gibson, eds. *History of Programming Languages — II*. Addison-Wesley, 1996. ISBN 0202295021.
2. Hailpern, Brent, and Barbara G. Ryder, eds. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. `http://portal.acm.org/toc.cfm?id=1238844`.
3. Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts — The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 978-0465042265.
4. Sammet, Jean. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.

5. Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.

Контрольные вопросы

1. Зачем нужна история?
2. Зачем нужны языки программирования? Приведите примеры.
3. Перечислите некоторые фундаментальные принципы хороших языков программирования.
4. Что такое абстракция? Что такое высокий уровень абстракции?
5. Назовите высокоуровневые идеалы программирования.
6. Перечислите потенциальные преимущества высокоуровневого программирования.
7. Что такое повторное использование кода и в чем заключается его польза?
8. Что такое процедурное программирование? Приведите конкретный пример.
9. Что такое абстракция данных? Приведите конкретный пример.
10. Что такое объектно-ориентированное программирование? Приведите конкретный пример.
11. Что такое обобщенное программирование? Приведите конкретный пример.
12. Что такое мультипарадигменное программирование? Приведите конкретный пример.
13. Когда была выполнена первая программа на компьютере с хранимой программой?
14. Какую выдающуюся работу выполнил Дэвид Уилер?
15. Расскажите об основном вкладе первого языка программирования Джона Бэкуса.
16. Какой первый язык разработала Грейс Мюррей Хоппер?
17. В какой области компьютерных наук выполнил свою главную работу Джон Маккарти?
18. Какой вклад внес Питер Наур в создание языка Algol60?
19. Какую выдающуюся работу выполнил Эдсгер Дейкстра?
20. Какие языки спроектировал и реализовал Никлаус Вирт?
21. Какие языки разработал Андерс Хейльсберг?
22. Какова роль Жана Ишби в проекте Ada?

23. Какой стиль программирования связан с языком Simula?
24. Где (кроме Осло) преподавал Кристен Нюгорд?
25. Какую выдающуюся работу выполнил Оле-Йохан Даль?
26. Какая операционная система была разработана под руководством Кена Томпсона?
27. Какую выдающуюся работу выполнил Дуг Мак-Илрой?
28. Назовите наиболее известную книгу Брайана Кернигана.
29. Где работал Деннис Ритчи?
30. Какую выдающуюся работу выполнил Бьярне Страуструп?
31. Какие языки пытался использовать Алекс Степанов для проектирования библиотеки STL?
32. Назовите десять языков программирования, не описанных в разделе 22.2.
33. Диалектом какого языка программирования является язык Scheme?
34. Назовите два наиболее известных наследника языка C++.
35. Что означает "C" в названии "C++"?
36. Является ли слово "Fortran" аббревиатурой? Если да, то какие слова в нем использованы?
37. Является ли слово "COBOL" аббревиатурой? Если да, то какие слова в нем использованы?
38. Является ли слово "Lisp" аббревиатурой? Если да, то какие слова в нем использованы?
39. Является ли слово "Pascal" аббревиатурой? Если да, то какие слова в нем использованы?
40. Является ли слово "Ada" аббревиатурой? Если да, то какие слова в нем использованы?
41. Назовите самый лучший язык программирования.

Термины

В этой главе раздел "Термины" содержит названия языков, имена людей и названия организаций.

- Языки
  - Ada
  - Algol
  - BCPL
  - C
  - C++
  - COBOL
  - Fortran
  - Lisp
  - Pascal
  - Scheme
  - Simula

- Люди
  - Чарльз Бэббидж
  - Джон Бэкус
  - Никлаус Вирт
  - Оле-Йохан Даль
  - Эдсгер Дейкстра
  - Жан Ишби
  - Брайан Керниган
  - Дуг Мак-Илрой
  - Джон Маккарти
  - Питер Наур
  - Кристен Нюгорд
  - Деннис Ритчи
  - Алекс Степанов
  - Бьярне Страуструп
  - Кен Томпсон
  - Дэвид Уилер
  - Андерс Хейльсберг
  - Грейс Мюррей Хоппер
- Организации
  - Bell Laboratories
  - Borland
  - Cambridge University (Великобритания)
  - ETH (Швейцарский федеральный технический университет)
  - IBM
  - MIT
  - Norwegian Computer Center
  - Princeton University
  - Stanford University
  - Technical University of Copenhagen
  - U.S. Department of Defense
  - U.S. Navy

Упражнения

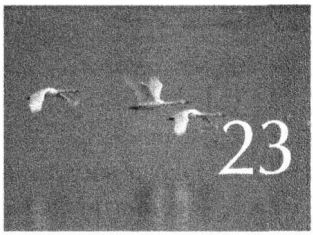
1. Дайте определение понятия *программирование*.
2. Дайте определение понятия *язык программирования*.
3. Пролистайте книгу и прочитайте эпиграфы к главам. Какие из них принадлежат специалистам по компьютерным наукам? Напишите один абзац, суммирующий их высказывания.
4. Пролистайте книгу и прочитайте эпиграфы к главам. Какие из них не принадлежат специалистам по компьютерным наукам? Назовите страну, где они родились, и область работы каждого из них.
5. Напишите программу "Hello, World!" на каждом из языков, упомянутых в этой главе.
6. Для каждого из упомянутых языков программирования найдите популярный учебник и первую законченную программу, написанную на нем. Напишите эту программу на всех остальных языках, упомянутых в главе. *Предупреждение*: скорее всего, вам придется написать около ста программ.
7. Очевидно, мы пропустили много важных языков. В частности, мы были вынуждены отказаться от описания всех языков, появившихся

после языка C++. Назовите пять современных языков, которые вы считаете достойными внимания, и напишите полторы страницы о трех из них.

8. Зачем нужен язык C++? Напишите 10–20-страничное сочинение.
9. Зачем нужен язык C? Напишите 10–20-страничное сочинение.
10. Выберите один язык программирования (не C и не C++) и напишите 10–20-страничное сочинение о его истории, целях и возможностях. Приведите много конкретных примеров. Кто использует этот язык и почему?
11. Кто в настоящее время занимает Лукасианскую кафедру в Кембридже?
12. Кто из разработчиков языков программирования, перечисленных в главе, имеет научную степень по математике, а кто нет?
13. Кто из разработчиков языков программирования, перечисленных в главе, имеет степень доктора философии, а кто нет? В какой области?
14. Кто из разработчиков языков программирования, перечисленных в главе, является лауреатом премии Тьюринга? За какие достижения? Найдите официальные объявления о присуждении премии Тьюринга лауреатам, упомянутым в главе.
15. Напишите программу, которая считывает файл, содержащий пары (имя, год), например (Algol,1960) и (C,1974), и выводит соответствующие имена на шкале времени.
16. Модифицируйте программу из предыдущего упражнения так, чтобы она считывала из файла кортежи (имя,год,(предшественники)), например (Fortran,1956,()), (Algol,1960,(Fortran)) или (C++,1985,(C,Simula)), и выводила граф со стрелками, направленными от предшественников к последователям. Используя эту программу, нарисуйте улучшенные варианты диаграмм из разделов 22.2.2 и 22.2.7.

Послесловие

Очевидно, что мы лишь вскользь затронули историю языков программирования и идеалов программного обеспечения. Мы считаем эти вопросы очень важными и потому глубоко огорчены тем, что не в состоянии изложить их в настоящей книге достаточно полно. Надеемся, что нам удалось передать свои чувства и некоторые идеи, относящиеся к нескончаемому поиску наилучшего программного обеспечения и методов программирования при проектировании и реализации языков программирования. Иначе говоря, помните, пожалуйста, что главное — это программирование, т.е. разработка качественного программного обеспечения, а язык программирования — просто инструмент для ее реализации.



23

Работа с текстом

Ничто не может быть очевидным настолько, чтобы быть действительно очевидным... Употребление слова "очевидно" свидетельствует об отсутствии логических аргументов.

— Эррол Моррис (Errol Morris)

В этой главе речь идет в основном об извлечении информации из текста. Мы храним знания в виде слов, зафиксированных в документах, таких как книги, сообщения электронной почты, или распечатанных таблиц, чтобы впоследствии извлечь их оттуда в форме, удобной для вычислений. Здесь мы опишем возможности стандартной библиотеки, которые интенсивнее остальных используются для обработки текстов: классы **string**, **iostream** и **map**. Затем введем регулярные выражения (класс **regex**) как способ выражения шаблонов текста. В заключение покажем, как с помощью регулярных выражений находить и извлекать из текста специфические элементы данных, такие как почтовые индексы, а также верифицировать форматы текстовых файлов.

23.1. Текст

По существу, мы постоянно работаем с текстом. Наши книги заполнены текстом, бо́льшая часть того, что мы видим на экране компьютера, — это текст, и исходный код наших программ является текстом. Наши каналы связи (всех видов) переполнены словами. Всю информацию, которой обмениваются два человека, можно было бы представить в виде текста, но не будем заходить так далеко. Изображения и звуки обычно лучше всего представлять в виде изображений и звуков (т.е. в виде совокупности битов), но все остальное можно обрабатывать с помощью программ анализа и преобразования текста.

Начиная с главы 3 мы использовали классы `iostream` и `string`, поэтому здесь мы лишь вкратце опишем соответствующие библиотеки. В особенности полезны при обработке текстов отображения (раздел 23.4), поэтому здесь мы приведем пример их использования для анализа электронной почты. После этого обзора в главе рассматриваются вопросы поиска шаблонных фрагментов в тексте с помощью регулярных выражений (разделы 23.5–23.10).

23.2. Строки

Класс `string` содержит последовательность символов и предоставляет ряд полезных операций, таких как добавление символа к строке, определение длины строки и конкатенация двух строк. На самом деле стандартный класс `string` обеспечивает довольно мало операций, но большинство из них оказываются полезными только при низкоуровневой обработке действительно сложных текстов. Здесь мы лишь упомянем о нескольких

наиболее полезных операциях. При необходимости их полное описание (как и исчерпывающий список операций из класса `string`) можно найти в справочнике или учебнике уровня экспертов. Все эти операции находятся в заголовочном файле `<string>` (обратите внимание — не в `<string.h>`).

Некоторые операции над строками

`s1 = s2`	Присвоение строки `s2` строке `s1`; `s2` может быть строкой `string` или С-строкой
`s += x`	Добавление `x` в конец строки; `x` может быть символом, строкой `string` или С-строкой
`s[i]`	Индексация
`s1+s2`	Конкатенация; символы в целевой строке будут копиями символов их строки `s1`, за которыми следуют копии символов из строки `s2`
`s1==s2`	Сравнение строк; либо `s1`, либо `s2` (но не оба одновременно) может быть С-строкой. То же самое относится к операции `!=`
`s1<s2`	Лексикографическое сравнение строк; либо `s1`, либо `s2` (но не оба одновременно) может быть С-строкой. То же самое относится к операциям `<=`, `>` и `>=`
`s.size()`	Количество символов в строке `s`
`s.length()`	Количество символов в строке `s`
`s.c_str()`	Символы строки `s`, представленные в виде С-строки
`s.begin()`	Итератор, указывающий на первый символ
`s.end()`	Итератор, указывающий на символ, следующий за концом строки `s`
`s.insert(pos,x)`	Вставка `x` перед символом `s[pos]`; `x` может быть строкой `string` или С-строкой. Строка `s` увеличивается, чтобы поместить символы из `x`
`s.append(x)`	Вставка `x` после последнего символа `s`; `x` может быть строкой `string` или С-строкой. Строка `s` увеличивается, чтобы поместить символы из объекта `x`
`s.erase(pos)`	Удаление конечных символов строки, начиная с `s[pos]`. Размер строки `s` становится равным `pos`
`s.erase(pos,n)`	Удаление `n` символов из строки `s`, начиная с `s[pos]`. Размер строки `s` становится равным `max(pos,size-n)`
`pos = s.find(x)`	Поиск `x` в строке `s`; `x` может быть символом, строкой `string` или С-строкой; переменная `pos` является индексом первого найденного символа или значением `string::npos` (позиция за концом строки `s`)
`in>>s`	Считывание слова, отделенного пробельными символами, из потока `in` в строку `s`
`getline(in,s)`	Считывание строки текста из потока `in` в строку `s`
`out<<s`	Запись строки `s` в поток `out`

Операции ввода-вывода описаны в главах 10 и 11 и подытожены в разделе 23.3. Обратите внимание на то, что операции ввода в объект класса

string при необходимости увеличивают его размер, поэтому переполнение никогда не происходит.

Операции **insert()** и **append()** перемещают символы, чтобы освободить место для новых. Операция **erase()** сдвигает символы влево, чтобы заполнить пробел, оставшийся после удаления символа.

На самом деле стандартная строка представляет собой шаблон **basic_string**, поддерживающий разные наборы символов, такие как Unicode, в котором предусмотрены тысячи символов (таких, как 🏴‍☠️, Ω, ∞, ☺, 🤖 или ♫ в дополнение к "обычным символам"). Скажем, если у вас есть тип, хранящий символы из набора Unicode, например **Unicode**, можно написать следующий код:

```
basic_string<Unicode> a_unicode_string;
```

Стандартный класс **string**, который мы используем, является просто классом **basic_string**, инстанцированным обычным **char**.

```
using string = basic_string<char>;
```

Мы не будем описывать символы или строки кода Unicode, но при необходимости вы можете работать с ними точно так же, как и с обычными символами и строками (с помощью обычных возможностей языка, класса **string**, потоков **iostream** и регулярных выражений). Если вам нужны символы Unicode, то лучше всего попросить совета у опытных пользователей; для того чтобы ваша программа была полезной, вы должны следовать не только правилам языка, но и некоторым системным соглашениям.

В контексте обработки текста важно помнить, что практически все можно представить в виде строки символов. Например, на этой странице число **12.333** представлено в виде строки, состоящей из шести символов и окруженной пробелами. Если вы считали это число, то должны сначала превратить эти символы в число с плавающей точкой и лишь затем применять к нему арифметические операции. Это приводит к необходимости преобразования чисел в строки **string** и строк — в числа. В разделе 11.4 мы видели, как превратить целое число в строку с помощью класса **ostringstream**. Этот прием можно обобщить для любого типа, имеющего оператор **<<**.

```
template<typename T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

Например:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

Значение строки **s1** равно **"12.333"**, а значение строки **s2** — **"17"**. Фактически функцию **to_string()** можно применять не только к числовым значениям, но и к любому классу **T** с оператором **<<**.

Обратное преобразование, из строки **string** в число, столь же простое, сколь и полезное.

```
struct bad_from_string : std::bad_cast
    // Класс для сообщений об ошибках при преобразовании строки
{
    const char* what() const // Перекрывает what() из bad_cast
    {
        return "некорректное преобразование из string";
    }
};

template<typename T> T from_string(const string& s)
{
    istringstream is(s);
    T t;
    if (!(is >> t)) throw bad_from_string();
    return t;
}
```

Например:

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
    int i = from_string<int>(s);
    // ...
}
catch (bad_from_string e) {
    error("неверная строка ввода",s);
}
```

Дополнительная сложность функции **from_string()** по сравнению с функцией **to_string()** объясняется тем, что класс **string** может представлять значения многих типов. Это значит, что каждый раз мы должны указывать, какой именно тип значений хотим извлечь из строки **string**. Кроме того, это значит, что рассматриваемая строка может не хранить значение типа, который мы ожидаем. Рассмотрим пример.

```
int d = from_string<int>("Mary had a little lamb"); // Ой!
```

Таким образом, возможна ошибка, которую мы представили исключением **bad_from_string**. В разделе 23.9 мы покажем, что функция **from_string()** (или эквивалентная ей) играет важную роль в серьезных текстовых приложениях, поскольку нам необходимо извлекать числовые значения из текстовых полей. В разделе 16.4.3 было показано, как эквивалентная

функция `get_int()` применяется в коде графического пользовательского интерфейса.

Обратите внимание, что функции `to_string()` и `from_string()` очень похожи. Фактически они являются обратными друг другу; иначе говоря (игнорируя детали, связанные с пробелами, округлением и т.д.), для каждого "разумного типа `T`" имеем

```
s==to_string(from_string<T>(s)) // Для всех s
```

и

```
t==from_string<T>(to_string(t)) // Для всех t
```

Здесь слово "разумный" означает, что для типа `T` должны быть определены конструктор по умолчанию, оператор `>>` и соответствующий оператор `<<`.

Заметим, что реализации функций `to_string()` и `from_string()` используют класс `stringstream` для выполнения всей тяжелой работы. Это наблюдение было использовано для определения универсальной операции преобразования двух произвольных типов с согласованными операциями `<<` и `>>`.

```
template<typename Target, typename Source>
Target to(Source arg)
{
    stringstream interpreter;
    Target result;
    if (!(interpreter << arg)        // Запись arg в поток
        || !(interpreter >> result) // Чтение result из потока
                                    // В потоке что-то осталось?
        || !(interpreter >> std::ws).eof())
            throw runtime_error{"сбой в to<>()"};
    return result;
}
```

Довольно забавное и остроумное решение: инструкция `!(interpreter>> std::ws).eof()` считывает любой пробельный символ, который может остаться в потоке `stringstream` после извлечения результата. Пробелы допускаются, но, кроме них, в потоке ввода не должно оставаться никаких иных символов, и мы можем проверить выполнение этого условия как обнаружение конца файла. Таким образом, если мы попытаемся считать целое число `int` из строки `string`, то и `to<int>("123")`, и `to<int>("123 ")` успешно завершатся, в то время как выражение `to<int>("123.5")` — не завершится из-за наличия символов `.5` в конце.

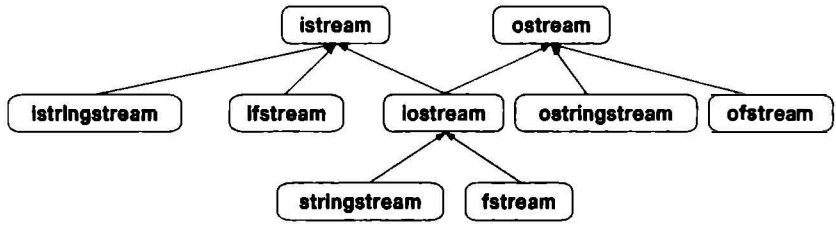
23.3. Потоки ввода-вывода

Рассматривая связь между строками и другими типами, мы приходим к потокам ввода-вывода. Библиотека ввода-вывода не просто выполняет ввод и вывод; она также осуществляет преобразования

между форматами строк и типами в памяти. Стандартные потоки ввода-вывода обеспечивают возможности чтения, записи и форматирования строк символов. Библиотека **iostream** описана в главах 10 и 11, поэтому просто подведем итог.

Потоковый ввод-вывод	
`in >> x`	Считывает данные из потока `in` в объект `x` в соответствии с типом объекта `x`
`out << x`	Записывает объект `x` в поток `out` в соответствии с типом объекта `x`
`in.get(c)`	Считывает символ из потока `in` в `c`
`getline(in,s)`	Считывает строку из потока `in` в строку `s`

Стандартные потоки организованы в виде иерархии классов (см. раздел 14.3).



В совокупности эти классы обеспечивают возможность выполнять ввод-вывод, используя файлы и строки (а также все, что выглядит как файлы и строки, как, например клавиатура и экран; см. главу 10). Как указано в главах 10 и 11, потоки **iostream** предоставляют широкие возможности для форматирования. Стрелки на рисунке обозначают наследование (см. раздел 14.3), поэтому, например, класс **stringstream** можно использовать вместо классов **iostream**, **istream** и **ostream**.

Как и строки, потоки ввода-вывода можно использовать с широкими символами данных, такими как Unicode, так же, как и с обычными символам. Еще раз заметим, что, если вам необходимо работать с вводом-выводом символов Unicode, лучше всего спросить совета у экспертов; для того чтобы стать полезной, ваша программа должна не просто соответствовать правилам языка, но и соблюдать определенные системные соглашения.

23.4. Ассоциативные контейнеры

Ассоциативные контейнеры (ассоциативные массивы и хеш-таблицы) играют ключевую роль в обработке текста. Причина проста — обрабатывая текст, мы накапливаем информацию, а она часто связана с текстовыми строками, такими как имена, адреса, почтовые индексы, номера карточек

социального страхования, место работы и т.д. Даже если некоторые из этих текстовых строк можно преобразовать в числовые значения, часто более удобно и проще обрабатывать их именно как текст и использовать этот текст для идентификации. В этом отношении ярким примером является подсчет слов (см. раздел 21.6). Если вы не чувствуете себя комфортно при работе с классом `map`, пожалуйста, еще раз прочитайте раздел 21.6.

Рассмотрим сообщение электронной почты. Мы часто выполняем анализ сообщений электронной почты и журнальных записей и поиск в них с помощью некоторой программы (например, Thunderbird, SeaMonkey или Outlook). Чаще всего эти программы скрывают от нас полное представление сообщения, но вся информация о том, кто его отослал, кто получил, через какие узлы оно прошло и многое другое поступает в программы в виде текста, содержащегося в заголовке письма. Так выглядит полное сообщение. Существуют тысячи инструментов для анализа заголовков. Большинство из них для извлечения информации использует регулярные выражения (описанные в разделах 23.5–23.9) и некоторые разновидности ассоциативных массивов. Например, мы часто выполняем поиск в файле электронной почты для сбора всех писем, поступивших от одного и того же отправителя, имеющих одну и ту же тему или содержащих информацию по некоторой теме.

Приведем упрощенный вид файла электронной почты для демонстрации некоторых методов извлечения данных из текстовых файлов. Показанные заголовки представляют собой реальные заголовки RFC2822 (см. страницу `www.faqs.org/rfcs/rfc2822.html`). Рассмотрим пример.

```
xxx
xxx
----
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>
This is a message just to say hello.
So, "Hello".
----
From: Joe Q. Public <john.q.public@example.com>
To: Mary Smith <@machine.tld:mary@example.net>, , jdoe@test .example
Date: Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>
Hi everyone.
----
To: "Mary Smith: Personal Account" <smith@home.example>
From: John Doe <jdoe@machine.example>
Subject: Re: Saying Hello
Date: Fri, 21 Nov 1997 11:00:00 -0600
Message-ID: <abcd.1234@local.machine.tld>
In-Reply-To: <3456@example.net>
```

```
References: <1234@local.machine.example> <3456@example.net>
This is a reply to your reply.
----
----
```

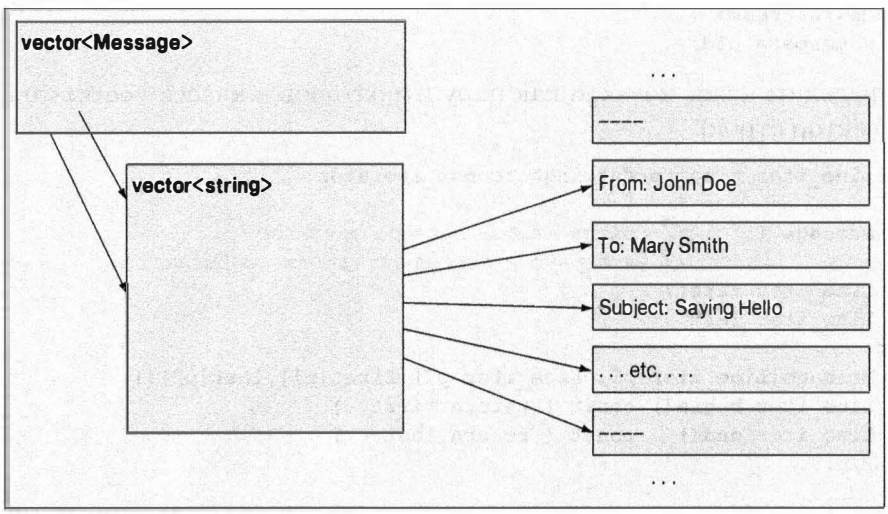
Вообще-то, мы сократили файл, отбросив большую часть информации и облегчив анализ, завершив каждое сообщение строкой, содержащей символы ---- (четыре дефиса). Мы собираемся написать "игрушечное приложение", которое будет искать все сообщения, отосланные отправителем John Doe, и выводить на экран их темы. Если мы сможем это сделать, то научимся делать много интересных вещей.

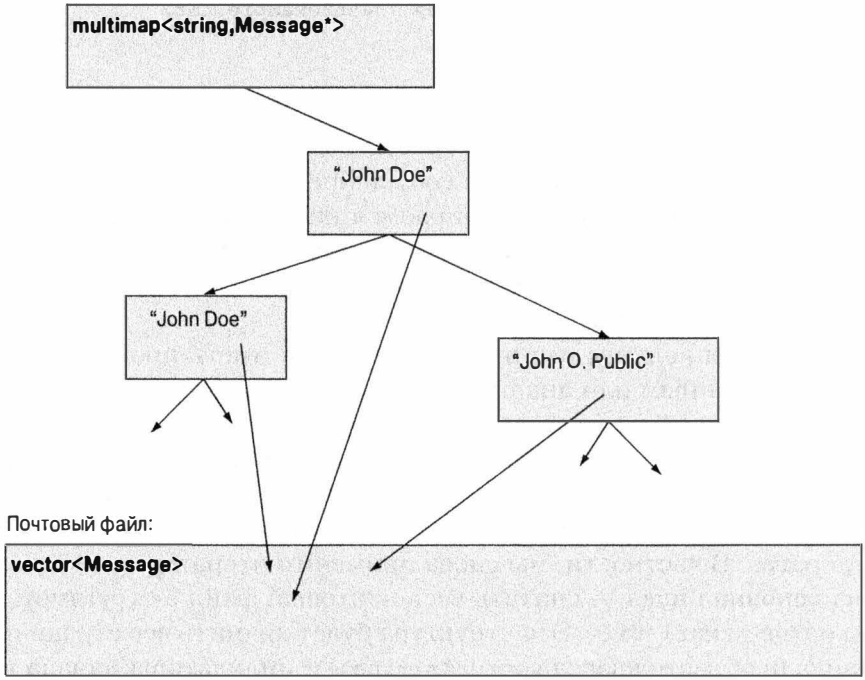
Во-первых, мы должны решить, хотим ли мы иметь произвольный доступ к данным или анализировать их как входные потоки. Мы выбрали первый вариант, поскольку в реальной программе нас, вероятно, интересовали бы несколько отправителей или несколько фрагментов информации, поступившей от конкретного отправителя. Кроме того, эту задачу решить труднее, поэтому нам придется проявить больше мастерства. В частности, мы снова применим итераторы.

Наша основная идея — считать весь почтовый файл в структуру, которую мы назовем `Mail_file`. Эта структура будет хранить все строки почтового файла (в объекте класса `vector<string>`) и индикаторы начала и конца каждого отдельного сообщения (в объекте класса `vector<Message>`).

Для этого мы добавим итераторы, а также функции `begin()` и `end()`, чтобы иметь возможность перемещаться по строкам и сообщениям как обычно. Эта схема обеспечит нам удобный доступ к сообщениям. Имея такой инструмент, мы напишем наше "игрушечное приложение", позволяющее собирать вместе все сообщения, поступившие от одного и того же адресата, чтобы их было легче найти.

Почтовый файл:





В заключение выведем на экран темы всех сообщений, поступивших от John Doe, чтобы проиллюстрировать созданный нами механизм доступа к структурам.

Мы используем для этого базовые средства стандартной библиотеки.

```
#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
```

Определим класс **Message** как пару итераторов в классе **vector<string>** (наш вектор строк).

```
using Line_iter = vector<string>::const_iterator;

class Message {      // Объект класса Message указывает
                     // на первую и последнюю строки сообщения
      Line_iter first;
      Line_iter last;
public:
      Message(Line_iter p1, Line_iter p2):first{p1},last{p2}{}
      Line_iter begin() const { return first; }
      Line_iter end()   const { return last;  }
      // ...
};
```

Определим класс **Mail_file** как структуру, содержащую строки текста и сообщения.

```
typedef vector<Message>::const_iterator Mess_iter;

struct Mail_file {    // Mail_file хранит все строки из файла
                      // и упрощает доступ к сообщениям
    string name;                  // Имя файла
    vector<string> lines;         // Строки по порядку
    vector<Message> m;            // Сообщения по порядку
    Mail_file(const string& n); // Чтение файла n в строки
    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const   { return m.end();   }
};
```

Заметим, что мы добавили в структуры данных итераторы, чтобы иметь возможность систематически перемещаться по ним. На самом деле мы не собираемся использовать здесь стандартные библиотечные алгоритмы, но если бы мы этого захотели, то итераторы позволили бы нам сделать это.

Для того чтобы найти и извлечь информацию, содержащуюся в сообщении, нужны две вспомогательные функции.

```
// Ищет отправителя в объекте класса Message; возвращает значение true,
// если он найден; если найден, помещает отправителя в строку s:
bool find_from_addr(const Message* m, string& s);
// Возвращает тему сообщения; если ее нет, возвращает "":
string find_subject(const Message* m);
```

Итак, мы можем написать следующий код для извлечения информации из файла.

```
int main()
{
    Mail_file mfile{"my-mail-file.txt"}; // Инициализируем mfile
                                         // данными из файла

    // Сначала собираем сообщения, поступившие от каждого
    // отправителя, в объекте класса multimap:
    multimap<string, const Message*> sender;

    for(const auto& m : mfile) {
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

    // Теперь обходим мультиотображение и извлекаем темы
    // сообщений, поступивших от John Doe:
    auto pp = sender.equal_range(
            "John Doe <jdoe@machine.example>");
    for(auto p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
}
```

Рассмотрим подробнее использование отображений. Мы использовали класс **multimap** (разделы 20.10 и Б.4), поскольку хотели собрать в одном месте много сообщений, поступивших из одного адреса. Класс **multimap** из стандартной библиотеки делает именно это (облегчая доступ к элементам с помощью одного и того же ключа). Очевидно (и типично), что наша задача распадается на две подзадачи:

- создать отображение;
- использовать отображение.

Мы создаем объект класса **multimap** путем обхода всех сообщений и их вставки в мультиотображение с помощью функции **insert()**:

```
for (const auto& m : mfile) {
    string s;
    if (find_from_addr(&m,s))
        sender.insert(make_pair(s,&m));
}
```

В ассоциативный массив включаются пары (ключ, значение), созданные с помощью функции **make_pair()**. Для того чтобы найти имя отправителя, используем "кустарную" функцию **find_from_addr()**.

Почему мы сначала поместили объекты класса **Message** в вектор, а затем создали объект класса **multimap**? Почему бы не включать объекты **Message** непосредственно в отображение? Причина носит простой и фундаментальный характер.

- Сначала мы создаем универсальную структуру, которую можно использовать для многих вещей.
- Затем используем ее в конкретном приложении.

Таким образом, мы создаем коллекцию в той или иной степени повторно используемых компонентов. Если бы мы сразу создали отображение в **Mail_file**, то вынуждены были бы переопределять его каждый раз, когда требовалось бы решать другую задачу. В частности, наше мультиотображение (многозначительно названное **sender**) упорядочено по полю адреса. Множество других приложений могут использовать другой критерий сортировки: по полям получателей, темы, временным меткам и т.п.

Такое поэтапное создание приложений (или по *слоям* (layers), как их иногда называют) может значительно упростить проектирование, реализацию, документацию и эксплуатацию программ. Дело в том, что каждая часть приложения решает отдельную задачу и делает это вполне очевидным образом. С другой стороны, для того чтобы сделать все сразу, нужен большой талант. Очевидно, что извлечение информации из заголовков сообщений электронной почты — это детский пример приложения. Значение разделения задач, выделения модулей и поступательного наращивания приложения по мере увеличения масштаба приложения проявляется все более ярко.

Для того чтобы извлечь информацию, мы просто ищем все упоминания ключа "John Doe" с помощью функции equal_range() (раздел Б.4.10). Затем мы проходим по всем элементам последовательности [first,second), возвращаемой функцией equal_range(), извлекая темы сообщений с помощью функции find_subject().

```
auto pp = sender.equal_range("John Doe <jdoe@machine.example>");

for (auto p = pp.first; p!=pp.second; ++p)
    cout << find_subject(p->second) << '\n';
```

Обходя элементы отображения, мы получаем последовательность пар (ключ, значение), в которых, как в любом другом объекте класса pair, первый элемент (в данном случае — ключ типа string) называется first, а второй (в данном случае — значение типа Message) — second (см. раздел 21.6).

23.4.1. Детали реализации

Очевидно, что мы должны реализовать используемые нами функции. Соблазнительно, конечно, сэкономить бумагу и спасти дерево, предоставив читателям самостоятельно решить эту задачу, но мы решили, что этот пример должен быть полным. Конструктор класса Mail_file открывает файл и создает векторы lines и m.

```
Mail_file::Mail_file(const string& n)
    // Открывает файл с именем n
    // Считывает строки из файла n в lines
    // Находит в lines сообщения и помещает их в m;
    // для простоты полагаем, что каждое сообщение
    // заканчивается строкой "----"
{
    ifstream in(n);    // Открываем файл
    if (!in) {
        cerr << "Нет файла" << n << '\n';
        exit(1);       // Прекращаем выполнение программы
    }

    for(string s; getline(in,s); ) // Создаем вектор строк
        lines.push_back(s);

    auto first = lines.begin();    // Создаем вектор сообщений
    for(auto p = lines.begin(); p!=lines.end(); ++p) {
        if (*p == "----") {        // Конец сообщения
            m.push_back(Message(first,p));
            first = p+1;           // Строка ---- не является
                                   // частью сообщения
        }
    }
}
```

Обработка ошибок здесь носит рудиментарный характер. Если бы мы писали эту программу для своих друзей, то постарались бы сделать ее получше.

ПОПРОБУЙТЕ

Выполните этот пример и убедитесь, что вы понимаете, что в нем происходит. Что означает "более хорошая обработка ошибок"? Измените конструктор `Mail_file` так, чтобы он обрабатывал ошибки форматирования, связанные с использованием строки "`----`".

Функции `find_from_addr()` и `find_subject()` пока что не имеют конкретного наполнения — пока мы не выясним, как идентифицировать информацию в файле с использованием регулярных выражений из разделов 23.6–23.10.

```
int is_prefix(const string& s, const string& p)
    // Является ли строка p первой частью строки s?
{
    int n = p.size();
    if (string(s,0,n)==p) return n;
    return 0;
}

bool find_from_addr(const Message* m, string& s)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "From: ")) (
            s = string(x,n);
            return true;
        }
    return false;
}
string find_subject(const Message* m)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "Subject: ")) return string(x,n);
    return "";
}
```

Обратите внимание на то, как мы используем подстроки: конструктор `string(s,n)` создает строку, состоящую из окончания строки `s`, начиная с элемента `s[n]` (т.е. `s[n]..s[s.size()-1]`), в то время как конструктор `string(s,0,n)` создает строку, состоящую из символов `s[0]..s[n-1]`. Поскольку эти операторы на самом деле создают новые строки и копируют символы, они должны использоваться очень осторожно, чтобы не снизить производительность программы.

Почему функции `find_from_addr()` и `find_subject()` так отличаются друг от друга? Например, одна из них возвращает переменную типа `bool`, а другая — объект класса `string`. Потому что мы хотели подчеркнуть следующие моменты.

- Функция `find_from_addr()` различает ситуации найденной пустой строки адреса (`""`) и отсутствующей строки адреса. В первом случае функция `find_from_addr()` возвращает значение `true` (поскольку адрес найден) и присваивает строке `s` значение `""` (потому что адресная строка просто оказалась пустой). Во втором случае она возвращает значение `false` (поскольку в файле вообще не оказалось адресной строки).
- Функция `find_subject()` возвращает строку `""` и когда строка темы сообщения оказалась пустой, и когда ее вообще нет.

Насколько полезным является такое различие, которое проводит функция `find_from_addr()`? Необходимо ли это? Мы считаем, что это полезно и необходимо. При поиске информации в файле данных это различие проявляется снова и снова: нашли ли мы искомую строку и содержит ли она то, что нам нужно? В реальной программе обе функции, и `find_from_addr()`, и `find_subject()`, следовало бы написать в стиле функции `find_from_addr()`, чтобы дать пользователям возможность выявлять такое различие.

Эта программа не является оптимальной с точки зрения производительности, но мы надеемся, что в типичных ситуациях она сработает достаточно быстро. В частности, она считывает входной файл только один раз и не хранит несколько копий текста из этого файла. Для крупных файлов было бы целесообразно заменить класс `multimap` классом `unordered_multimap`, но без реальных измерений невозможно сказать заранее, насколько это повысит эффективность программы.

Введение в стандартные ассоциативные контейнеры (`map`, `multimap`, `set`, `unordered_map` и `unordered_multimap`) находится в разделе 21.6.

23.5. Проблема

Потоки ввода-вывода и класс `string` помогают нам считывать и записывать последовательности символов, хранить их и выполнять над ними основные операции. Однако при работе с текстом во многих случаях необходимо анализировать контекст строки или рассматривать много похожих строк. Рассмотрим тривиальный пример. Возьмем сообщение электронной почты (последовательность слов) и посмотрим, содержит ли оно аббревиатуру штата США и почтовый код (две буквы, за которыми следуют пять цифр).

```
for (string s; cin>>s; ) {
    if (s.size()==7
    && isalpha(s[0]) && isalpha(s[1])
```

```
    && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
    && isdigit(s[5]) && isdigit(s[6]))
        cout << "найдено " << s << '\n';
}
```

Здесь значение `isalpha(x)` равно `true`, если `x` представляет собой букву, а значение `isdigit(x)` равно `true`, если `x` является цифрой (см. раздел 11.6). В этом (слишком) простом решении кроется несколько проблем.

- Оно громоздко (четыре строки, восемь вызовов функций).
- Мы пропускаем (преднамеренно?) почтовые индексы, не отделенные от своего контекста пробелом (например, `"TX77845"`, `TX77845-1234` и `ATX77845`).
- Мы пропускаем (преднамеренно?) почтовые индексы с пробелом между буквами и цифрами (например, `TX 77845`).
- Мы принимаем (преднамеренно?) почтовые индексы, в которых буквы находятся в нижнем регистре (например, `tx77845`).
- Если вы решите проанализировать почтовые индексы, имеющие другой формат (например, `CB3 0FD`), то будете вынуждены полностью переписать весь код.

Должен быть лучший способ! Перед тем как его описать, рассмотрим проблемы, с которыми мы встретимся на "старом добром пути" написания кода для обработки разных случаев.

- Если мы захотим обработать еще один формат, нам придется добавить инструкцию `if` или `switch`.
- Если мы захотим учитывать верхний и нижний регистры, нам придется явно преобразовывать строки (обычно в нижний регистр) или добавлять дополнительную инструкцию `if`.
- Мы должны как-то (как?) описать контекст, в котором выполняется поиск. Это значит, что мы должны работать с отдельными символами, а не со строками, т.е. потерять многие преимущества, предоставляемые потоками `iostream` (раздел 7.8.2).

Если хотите, попробуйте написать код в этом стиле, но нам очевидно, что в этом случае вы запутаетесь в нагромождении инструкций `if`, предназначенных для обработки особых ситуаций. Даже в этом простом примере мы стоим перед выбором (например, учитывать ли пяти- и девятизначные почтовые индексы). Во многих других примерах нам необходимо работать с повторениями (например, любым количеством цифр, за которыми следует знак восклицания, такими как `123!` и `123456!`). В конце концов, нельзя забывать и о префиксах и суффиксах. Как мы уже указывали (см. разделы 11.1 и 11.2), предпочтения пользователей по отношению к разным выходным форматам не ограничиваются стремлением программистов к

систематичности и простоте. Просто подумайте о разнообразных способах записи одной только даты.

```
2007-06-05
June 5, 2007
jun 5, 2007
5 June 2007
6/5/2007
5/6/07
05.06.2007
...
```

В этот момент, если не раньше, опытный программист воскликнет "Должен быть лучший способ!" (чем нагромождение обычного кода) и станет его искать. Простейшим и наиболее широко распространенным решением таких задач является использование так называемых *регулярных выражений* (regular expression).

Регулярные выражения являются основой большинства методов обработки текстов и команды `grep` в системе Unix (см. упр. 8), а также важной частью языков, интенсивно работающих со строками (такими, как AWK, PERL и PHP).

Регулярные выражения, которые мы будем использовать, являются частью стандартной библиотеки C++. Они совместимы с регулярными выражениями из языка PERL. Этой теме посвящено много книг, учебников и справочников, например рабочий отчет Комитета по стандартизации языка C++ (в сети веб он известен под названием WG21), документация Джона Мэддока (John Maddock) к `boost::regex` и учебники по языку PERL. Здесь мы изложим фундаментальные понятия, а также основные и наиболее полезные способы использования регулярных выражений.

ПОПРОБУЙТЕ

В последних двух абзацах "неосторожно" упомянуты несколько имен и аббревиатур без каких-либо объяснений. Поищите в вебе информацию о них.

23.6. Идея регулярных выражений

Основная идея регулярного выражения заключается в том, что оно определяет *шаблон* (pattern), который мы ищем в тексте. Посмотрим, как мы могли бы точно описать шаблон простого почтового кода, такого как `TX77845`. Результат первой попытки выглядит следующим образом:

```
wwddddd
```

Здесь символ `w` означает любую букву, а символ `d` — любую цифру. Мы используем символ `w` (от слова "word"), поскольку символ `l` (от слова "letter") слишком легко перепутать с цифрой 1. Эти обозначения вполне подходят для нашего простого примера, но давайте попробуем применить их для описания формата почтового кода, состоящего из девяти цифр (например, `TX77845-5629`). Что вы скажете о таком решении?

```
wwddddd-dddd
```

Выглядит неплохо, но как понять, что символ `d` означает "любая цифра", а знак - означает "обычный" дефис? Нам необходимо как-то указать, что символы `w` и `d` являются специальными: они представляют классы символов, а не самих себя (символ `w` означает "`a` или `b` или `c` или ...", а символ `d` означает "`1` или `2` или `3` или ..."). Все это слишком сложно. Добавим к букве, обозначающей имя класса символов, обратную косую черту, как это сделано в языке C++ (например, символ `\n` означает переход на новую строку). Таким путем мы придем к строке

```
\w\w\d\d\d\d\d-\d\d\d\d
```

Выглядит довольно некрасиво, но по крайней мере мы устранили неоднозначность, а обратные косые черты ясно обозначают то, что за ними следует "нечто необычное". Здесь повторяющиеся символы просто перечислены один за другим. Это не только утомительно, но и провоцирует ошибки. Вы можете быстро сосчитать, что перед обратной косой чертой до дефиса действительно стоят пять цифр, а после него — четыре? Мы смогли, но мы ведь нигде не *сказали* вам, что их действительно `5` и `4`, а потому вам приходится их пересчитывать. Для краткости после каждого символа можно было бы добавить счетчик, указывающий количество его повторений:

```
\w2\d5-\d4
```

И нам вновь нужна какая-то синтаксическая конструкция, чтобы показать, что числа `2`, `5` и `4` в этом шаблоне являются значениями счетчиков, а не просто цифрами `2`, `5` и `4`. Выделим значения счетчиков фигурными скобками.

```
\w{2}\d{5}-\d{4}
```

Теперь символ `{` является таким же специальным символом, как и обратная косая черта, `\`, но этого избежать невозможно, и мы должны просто учитывать этот факт.

Все бы ничего... но мы забыли о двух обстоятельствах: последние четыре цифры в почтовом коде являются необязательными. Иногда допустимыми являются оба варианта: `TX77845` и `TX77845-5629`. Этот факт можно выразить двумя основными способами:

`\w{2}\d{5}` *или* `\w{2}\d{5}-\d{4}`

или

`\w{2}\d{5}` *и необязательно* `-\d{4}`

Точнее говоря, сначала мы должны выразить идею группирования (или подшаблона), чтобы говорить о том, что строки `\w{2}\d{5}` и `-\d{4}` являются частями строки `\w{2}\d{5}-\d{4}`. Обычно группирование выражается с помощью круглых скобок.

```
(\w{2}\d{5})(-\d{4})
```

Теперь, разделив шаблон на два *подшаблона*, мы должны указать, что именно мы хотим с ними делать. Как обычно, введение новой возможности достигается за счет использования нового специального символа: теперь символ `(` является специальным, как и символы `\` и `{`. Обычно для обозначения операции "или" (альтернативы) используется символ `|`, а символ `?` — для обозначения чего-то условного (необязательного). Таким образом, мы можем написать следующие выражения:

```
(\w{2}\d{5})|(\w{2}\d{5}-\d{4})
```

и

```
(\w{2}\d{5})(-\d{4})?
```

Как и фигурные скобки при обозначении счетчиков (например, `\w{2}`), знак вопроса (`?`) используется как суффикс. Например, `(-\d{4})?` означает "необязательное `-\d{4}`"; т.е. мы интерпретируем четыре цифры, перед которыми стоит дефис, как суффикс. Так как мы ни для чего не используем круглые скобки, выделяющие пятизначный почтовый код (`\w{2}\d{5}`), их можно опустить:

```
\w{2}\d{5}(-\d{4})?
```

Чтобы полностью завершить решение задачи, поставленной в разделе 23.5, можно добавить необязательный пробел после двух букв.

```
\w{2} ?\d{5}(-\d{4})?
```

Запись `" ?"` выглядит довольно странно, но знак вопроса после пробела указывает на то, что пробел является необязательным. Если такая запись кажется вам странной и вы хотите, чтобы пробел не выглядел опечаткой, можете заключить его в скобки:

```
\w{2}( )?\d{5}((-\d{4})?
```

Если бы и это кому-то не нравилось, нам пришлось бы придумать обозначение для пробела, например `\s` (`s` — от слова "space"). В этом случае запись выглядела бы так:

```
\w{2}\s?\d{5}(-\d{4})?
```

Но что если кто-то поставит после букв два пробела? В соответствии с определенным выше шаблоном это означало бы, что мы принимаем коды `TX77845` и `TX 77845`, но не `TX  77845`. Это неправильно.

Нам нужно средство, чтобы сказать "ни одного, один или несколько пробелов", поэтому мы вводим суффикс `*`.

```
\w{2}\s*\d{5}(-\d{4})?
```

Имеет смысл выполнять каждый этап в строгой логической последовательности. Эта система обозначений логична и очень лаконична. Кроме того, принятые нами решения взяты не с потолка: выбранная нами система обозначений очень широко распространена. При решении большинства задач, связанных с обработкой текста, нам необходимо читать и записывать такие обозначения. Да, они похожи на результат прогулки кошки по клавиатуре, и ошибка в единственном месте (даже лишний или пропущенный пробел) полностью изменяет их смысл, но с этим приходится мириться. Мы не можем предложить ничего радикально лучшего, и этот стиль обозначений за тридцать лет распространился очень широко. Впервые он был использован в программе `grep` в операционной системе Unix, но даже тогда его нельзя было назвать совершенно новым.

23.6.1. Необработанные строковые литералы

Обратите внимание на все эти обратные косые черты в шаблонах регулярных выражений. Чтобы получить обратную косую черту в строковом литерале C++, ее надо предварить другой обратной косой чертой. Рассмотрим наш шаблон почтового индекса:

```
\w{2}\s*\d{5}(-\d{4})?
```

Чтобы представить его в виде строкового литерала, надо записать

```
"\\w{2}\\s*\\d{5}(-\\d{4})?"
```

Забегая немного наперед, мы понимаем, что многие необходимые нам шаблоны должны содержать двойные кавычки ("). Чтобы вставить эти кавычки в строковый литерал, их надо предварить обратной косой чертой. Этот процесс быстро становится неуправляемым. Фактически эта "проблема специальных символов" становится настолько раздражающей, что как C++, так и некоторые другие языки программирования ввели понятие необработанного строкового литерала (raw string literal), чтобы работать с реалистичными шаблонами регулярных выражений. В таком необработанном строковом литерале символ обратной косой черты представляет собой обычный символ обратной косой черты (а не служебный символ), а его удвоение — две обратные косые черты подряд, как и двойная кавычка обозначает просто двойную кавычку, а не окончание строки. При этом наш шаблон почтового индекса превращается в

```
R"(\w{2}\s*\d{5}(-\d{4})?)"
```

Символы `R"(` указывают начало строки, а `)"` — ее завершение, так что 22 символами самой строки являются

```
\w{2}\s*\d{5}(-\d{4})?
```

(без учета нулевого завершающего символа).

23.7. Поиск с помощью регулярных выражений

Теперь применим шаблон почтовых кодов из предыдущего раздела для поиска почтовых кодов в файле. Программа определяет шаблон, а затем ищет его, считывая файл строка за строкой. Когда программа находит шаблон в какой-то строке, она выводит номер строки и то, что именно она нашла.

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream in{"file.txt"};                    // Входной файл
    if (!in) cerr << "нет файла\n";

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; // Шаблон индекса
    cout << "маблон: " << pat << '\n';

    int lineno = 0;
    for (string line; getline(in,line); ) {  // Чтение строки
        ++lineno;
        smatch matches;                         // Совпавшая строка
        if (regex_search(line, matches, pat))
            cout << lineno << ": " << matches[0] << '\n';
    }
}
```

Эта программа требует объяснений. Регулярные выражения стандартной библиотеки можно найти в заголовочном файле **<regex>**. С их помощью мы можем определить шаблон **pat**:

```
regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; // Шаблон индекса
```

Шаблон **pat** является разновидностью строки, так что мы инициализируем его строкой. Здесь мы использовали необработанный строковый литерал. Однако **regex** — это *не только* строка, но и некоторый сложный механизм сопоставления строки шаблону, который скрыто создается при инициализации объекта типа **regex** нашим шабло-

ном для почтового индекса (или при присваивании ему этого шаблона) и который выходит за рамки нашей книги. Однако после инициализации объекта **regex** шаблоном его можно применять к каждой строке нашего файла.

```
smatch matches;
if (regex_search(line, matches, pat))
    cout << lineno << ": " << matches[0] << '\n';
```

Функция **regex_search(line,matches,pat)** ищет в строке **line** все, что соответствует регулярному выражению, хранящемуся в объекте **pat**, и если она находит какое-либо соответствие, то сохраняет его в объекте **matches**. Естественно, если соответствие не обнаружено, функция **regex_search(line,matches,pat)** возвращает значение **false**.

Переменная **matches** имеет тип **smatch**. Буква **s** в нем означает "sub" (или "string" — что вам больше нравится). По сути тип **smatch** представляет собой вектор совпадений с подшаблонами. Первый элемент, **matches[0]**, представляет собой полное совпадение. Мы можем интерпретировать элемент **matches[i]** как строку, если **i < matches.size()**. Итак, если для данного регулярного выражения максимальное количество подшаблонов равно **N**, выполняется условие **matches.size()==N+1**.

Что же такое подшаблон (sub-pattern)? Можно просто сказать: "Все, что заключено в скобки внутри шаблона". Глядя на шаблон **\w{2}\s*\d{5}(-\d{4})?**, мы видим скобки вокруг четырехзначного расширения индекса. Таким образом, мы видим только один подшаблон, так что **matches.size()==2**. Можно предположить, что у нас есть простой доступ к этим четырем последним цифрам. Рассмотрим пример.

```
for (string line; getline(in,line); ) {
    smatch matches;
    if (regex_search(line, matches, pat)) {
        // Полное соответствие
        cout << lineno << ": " << matches[0] << '\n';
        if (1<matches.size() && matches[1].matched)
            // Подшаблон
            cout << "\t: " << matches[1] << '\n';
    }
}
```

Строго говоря, мы не обязаны проверять выражение **1<matches.size()**, поскольку уже рассмотрели шаблон, но к этому нас подталкивает легкая паранойя (поскольку мы экспериментируем с разными шаблонами, хранящимися в объекте **pat**, и не все они содержат только один подшаблон). Мы можем проверить, обнаружен ли подшаблон, просматривая его член **matched**, в данном случае — **matches[1].matched**. В случае, если вам это интересно, когда значение **matches[i].matched** равно **false**, то подшаблоны **matches[i]**, у которых нет соответствия, выводятся как пустые строки. Аналогично, если подшаблон не существует, например **matches[17]** для

приведенного выше шаблона, то он рассматривается как подшаблон, соответствие которому не найдено.

Применим нашу программу к файлу, содержащему следующие строки:

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
zzz TX23456-3456sss ggg TX33456-1234
cvzcv TX77845-1234 sdsas
xxxTx77845xxx
TX12345-123456
```

Результат работы программы имеет следующий вид:

```
pattern: "\w{2}\s*\d{5}(-\d{4})?"
1: TX77845
2: tx 77843
5: TX23456-3456
    : -3456
6: TX77845-1234
    : -1234
7: Tx77845
8: TX12345-1234
    : -1234
```

Следует подчеркнуть несколько важных моментов.

- Мы не дали себя запутать неверно отформатированным индексом в строке, начинающейся символами **ggg** (кстати, что в нем неправильно?).
- В строке, содержащей символы **zzz**, мы нашли только первый индекс (мы ищем только один индекс в строке).
- В строках 5 и 6 мы нашли правильные суффиксы.
- В строке 7 мы нашли код , скрытый среди символов **xxx**.
- Мы нашли (к сожалению?) код , скрытый в строке **TX12345-123456**.

23.8. Синтаксис регулярных выражений

Мы рассмотрели довольно элементарный пример сопоставления регулярных выражений. Настало время рассмотреть регулярные выражения (в форме, использованной в библиотеке **regex**) более полно и систематично.

Регулярные выражения (regular expressions, regexps или regexs), по сути, образуют небольшой язык для выражения шаблонов символов. Этот мощный, выразительный и лаконичный язык иногда выглядит довольно таинственным. За десятилетия использования регулярных выражений в этом языке появилось много тонких свойств и несколько диалектов. Здесь мы опишем (большое и полезное) подмножество регуляр-

ных выражений, которое, возможно, в настоящее время является наиболее распространенным диалектом (язык PERL). Если читателям понадобится более подробная информация о регулярных выражениях или возникнет необходимость кому-то их объяснить, они могут найти все, что нужно, в вебе. Существует огромное количество учебников (очень разного качества) и спецификаций.

Кроме описанной, стандартная библиотека поддерживает системы обозначений языков ECMAScript, POSIX и awk, а также grep и egrep. К тому же она содержит массу возможностей для поиска. Это может оказаться чрезвычайно полезным, особенно если вам необходимо выполнить поиск по шаблону, определенному другим языком. Если вам понадобятся языковые средства, которые выходят за рамки тем, которые мы описываем, поищите их самостоятельно. Однако помните, что использование "как можно большего числа возможностей" — это не самоцель качественного программирования. При любой возможности постарайтесь сжалиться над бедным программистом, который будет эксплуатировать вашу программу (возможно, им окажетесь вы сами через несколько месяцев), читать ее и пытаться разобраться в вашем коде: пишите код так, чтобы он не был заумным без особой причины и не содержал малопонятных мест.

23.8.1. Символы и специальные символы

Регулярное выражение определяет шаблон, который можно использовать для сопоставления символов из строки. По умолчанию символ в шаблоне соответствует в строке самому себе. Например, регулярное выражение (шаблон) `"abc"` соответствует подстроке `abc` строки `Is there an abc here?`

Реальная мощь регулярных выражений заключается в "специальных символах" и сочетаниях символов, имеющих особый смысл в шаблоне.

Специальные символы	
.	Любой отдельный символ
[	Класс символов
{	Счетчик
(	Начало группы
)	Конец группы
\	Следующий символ имеет особый смысл
*	Ни одного, один или больше
+	Один или больше
?	Необязательный (ни одного или один)
|	Альтернатива (или)
^	Начало строки; отрицание
$	Конец строки

Например, выражение

```
x.y
```

соответствует любой строке, состоящей из трех символов, начинающейся с буквы `x` и заканчивающейся буквой `y`, например `xxy`, `x3y` и `xay`, но не `yxy`, `3xy` или `xy`.

Обратите внимание на то, что символы `{...}`, `*`, `+` и `?` являются постфиксными операторами. Например, выражение `\d+` означает "одна или несколько десятичных цифр".

Если хотите использовать в шаблоне один из специальных символов, сделайте его управляющим, поставив перед ним обратную косую черту; например, символ `+` в шаблоне является оператором "один или несколько", а символ `\+` — это знак "плюс".

23.8.2. Классы символов

Самые распространенные сочетания символов в сжатом виде представлены как специальные символы.

Специальные символы для классов символов		
`\d`	Десятичная цифра	`[[:digit:]]`
`\l`	Символ в нижнем регистре	`[[:lower:]]`
`\s`	Пробельный символ (пробел, знак табуляции и т.д.)	`[[:space:]]`
`\u`	Символ в верхнем регистре	`[[:upper:]]`
`\w`	Буквы (a–z или A–Z) или цифра (0–9), или знак подчеркивания (_)	`[[:alnum:]]`
`\D`	Не `\d`	`[^[:digit:]]`
`\L`	Не `\l`	`[^[:lower:]]`
`\S`	Не `\s`	`[^[:space:]]`
`\U`	Не `\u`	`[^[:upper:]]`
`\W`	Не `\w`	`[^[:alnum:]]`

Специальные символы в верхнем регистре означают "не вариант специального символа в нижнем регистре". В частности, символ `\W` означает "не буква", а не "буква в верхнем регистре".

Элементы третьего столбца (например, `[[:digit:]]`) представляют собой альтернативные синтаксические конструкции, использующие более длинные имена.

Как и библиотеки `string` и `iostream`, библиотека `regex` может обрабатывать большие наборы символов, такие как Unicode. Как и в случае библиотек `string` и `iostream`, мы просто упоминаем об этом, чтобы при необходимости читатели могли самостоятельно найти информацию. Обсуждение манипуляций текстами в кодировке Unicode выходит за рамки рассмотрения нашей книги.

23.8.3. Повторения

Повторяющиеся шаблоны задаются суффиксными операторами.

Повторение	
`{n}`	Ровно *n* раз
`{n,}`	*n* или больше раз
`{n,m}`	Не меньше *n* раз и не больше *m* раз
`*`	Ни одного, один или несколько, т.е. `{0,}`
`+`	Один или больше, т.е. `{1,}`
`?`	Необязательный (ни одного или один), т.е. `{0,1}`

Например, выражение

```
Ax*
```

соответствует символу `A`, за котором не следует ни одного символа или следует несколько символов `x`:

```
A
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Если мы требуем, чтобы символ `x` встречался хотя бы один раз, то следует использовать оператор `+`, а не `*`. Например, выражение

```
Ax+
```

соответствует символу `A`, за которым следует один или несколько символов `x`:

```
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

но не

```
A
```

Распространенный случай необязательного символа (ни одного или несколько) указывается с помощью знака вопроса. Например, выражение

```
\d-?\d
```

соответствует двум цифрам с необязательным дефисом между ними:

```
1-2
12
```

но не

```
1--2
```

Для задания конкретного количества вхождений или конкретного диапазона вхождений используются фигурные скобки. Например, выражение

```
\w{2}-\d{4,5}
```

соответствует только строкам, содержащим две буквы и дефис, за которым следуют четыре или пять цифр:

```
Ab-1234
XX-54321
22-54321
```

но не

```
Ab-123
?b-1234
```

Да, именно так — цифры также соответствуют специальному символу `\w`.

23.8.4. Группировка

Для того чтобы указать, что некое регулярное выражение является *подшаблоном* (sub-pattern), его следует заключить в круглые скобки. Рассмотрим пример.

```
(\d*:)
```

Данное выражение определяет подшаблон, не содержащий ни одной или содержащий несколько цифр, за которыми следует двоеточие. Группу можно использовать как часть более сложного шаблона. Рассмотрим пример.

```
(\d*:)?(\d+)
```

Это выражение задает необязательную и, возможно, пустую последовательность цифр, за которыми следуют двоеточие и последовательность из одной или нескольких цифр. И этот лаконичный и точный способ выражения шаблонов изобрели обычные люди!

23.8.5. Альтернативы

Символ "или" (`|`) задает альтернативу. Рассмотрим пример.

```
Subject: (FW:|Re:)?(.*)
```

Это выражение распознает тему сообщения электронной почты с необязательными символами `FW:` или `Re:`, за которыми может не стоять ни одного символа или может стоять несколько символов. Рассмотрим примеры.

```
Subject: FW: Hello, world!
Subject: Re:
Subject: Norwegian Blue
```

но не

```
SUBJECT: Re: Parrots
Subject FW: No subject!
```

Пустая альтернатива не допускается.

```
(|def) // Ошибка
```

Разрешается указать сразу несколько альтернатив.

```
(bs|Bs|bS|BS)
```

23.8.6. Наборы символов и диапазоны

Специальные символы представляют собой обозначение наиболее распространенных классов символов: цифр (`\d`); букв, цифр и знака подчеркивания (`\w`) и др. (см. раздел 23.7.2). Однако часто полезно определить собственный специальный символ. Сделать это очень легко. Рассмотрим примеры.

`[\w @]`	Символ слова, пробел или `@`
`[a-z]`	Символы в нижнем регистре от `a` до `z`
`[a-zA-Z]`	Символы в верхнем или нижнем регистре от `a` до `z`
`[Pp]`	Символ `P` в верхнем или нижнем регистре
`[\w\-]`	Символ слова или дефис (просто "-" задает диапазон)
`[asdfghjkl;']`	Символы среднего ряда клавиатуры QWERTY
`[.]`	Точка
`[.[{( \\*+?^$]`	Символ со специальным значением в регулярном выражении

В спецификации класса символов дефис (`-`) используется для указания диапазона, например `[1-3]` (`1`, `2` или `3`) или `[w-z]` (`w`, `x`, `y` или `z`). Пожалуйста, будьте аккуратны при использовании таких диапазонов: не все языки содержат одни и те же буквы, и порядки их следования в алфавитах разных языков могут различаться. Если вам необходим диапазон, не являющийся поддиапазоном букв и цифр, принятых в английском языке, обратитесь к документации.

Следует подчеркнуть, что мы используем специальные символы, такие как `\w` (означающий "любой словообразующий символ"), в спецификации класса символов. Тогда как нам вставить обратную косую черту (\) в класс символов? Как обычно, превратив ее в управляющий символ с помощью второй обратной косой черты: `\\`.

Если первым символом в спецификации класса символов идет символ `^`, это означает отрицание, например:

`[^aeiouy]`	Не английская гласная буква
`[^\d]`	Не цифра
`[ ^aeiouy]`	Пробел, символ `^` или английская гласная буква

В последнем регулярном выражении символ ^ стоит не на первом месте после открывающей квадратной скобки ([), поэтому это простой символ, а не оператор отрицания. Регулярные выражения имеют свои тонкости.

Реализация библиотеки **regex** содержит также набор именованных классов символов, используемых в сопоставлениях. Например, если требуется определить буквенно-цифровые символы (т.е. буквы или цифры: **a–z** или **A–Z**, или **0–9**), то это можно сделать с помощью регулярного выражения `[[:alnum:]]`. Здесь слово **alnum** представляет собой имя совокупности символов (набор буквенно-цифровых символов). Шаблон для непустой строки буквенно-цифровых символов, заключенной в двойные кавычки, может выглядеть так: `"[[:alnum:]]+"`. Чтобы поместить это регулярное выражение в строковый литерал, к кавычкам надо добавить обратные косые черты, сделав их управляющими символами:

```
string s = "\"[[:alnum:]]+\"";
```

Более того, чтобы поместить этот строковый литерал в объект класса **regex**, следует сделать управляющими символами не только кавычки, но и саму обратную косую черту:

```
regex s{"\\\"[[:alnum:]]+\\\""};
```

Применение необработанных строковых литералов существенно упрощает жизнь программиста:

```
regex s2{R"("[[:alnum:]]+")"};
```

Использовать необработанные строковые литералы предпочтительно тогда, когда в них содержатся обратные косые черты или двойные кавычки. Впрочем, как оказывается, их содержит большинство шаблонов в большинстве приложений.

Использование регулярных выражений приводит ко множеству соглашений об обозначениях. Вот список стандартных классов символов.

Классы символов	
alnum	Любой буквенно-цифровой символ
alpha	Любой буквенный символ
blank	Любой пробельный символ, не являющийся разделителем строк
cntrl	Любой управляющий символ
d	Любая десятичная цифра
digit	Любая десятичная цифра
graph	Любой графический символ
lower	Любой символ в нижнем регистре
print	Любой печатаемый символ
punct	Любой знак пунктуации

Классы символов	
s	Любой пробельный символ
space	Любой пробельный символ
upper	Любой символ в верхнем регистре
w	Любой словообразующий символ (буквенно-цифровой символ или знак подчеркивания)
xdigit	Любой шестнадцатеричный цифровой символ

Реализация библиотеки **regex** может содержать и другие классы символов, но если вы решили использовать именованный класс, не указанный в этом списке, убедитесь, что переносимость программы окажется достаточной для предполагаемых условий ее применения.

23.8.7. Ошибки в регулярных выражениях

Что произойдет, если мы зададим неправильное регулярное выражение? Рассмотрим пример.

```
regex pat1("(|ghi)"); // Отсутствует альтернатива
regex pat2("[c-a]");  // Неверный диапазон
```

Когда мы присваиваем шаблон объекту класса **regex**, он подвергается проверке. Если механизм сопоставления регулярных выражений не может работать из-за того, что регулярное выражение неправильное или слишком сложное, генерируется исключение **bad_expression**.

Рассмотрим небольшую программу, позволяющую прочувствовать работу механизма сопоставления регулярных выражений.

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
#include<sstream>
using namespace std;

// Получает из входного потока шаблон и набор строк;
// проверяет шаблон и ищет строки, содержащие его

int main()
{
    regex pattern;

    string pat;
    cout << "Введите шаблон: ";
    getline(cin,pat);            // Считываем шаблон

    try {
        pattern = pat;          // Проверка шаблона
```

```
        cout << "шаблон: " << pat << '\n';
    }
    catch (bad_expression) {
        cout << "некорректное выражение " << pat << "\n";
        exit(1);
    }

    cout << "введите строки:\n";
    int lineno = 0;

    for (string line; getline(cin,line); ) {
        ++lineno;
        smatch matches;
        if (regex_search(line, matches, pattern)) {
            cout << "строка " << lineno << ": " << line << '\n';
            for (int i = 0; i<matches.size(); ++i)
                cout << "\tmatches[" << i << "]: "
                     << matches[i] << '\n';
        }
        else
            cout << "не соответствует\n";
    }
}
```

ПОПРОБУЙТЕ

Запустите эту программу и попробуйте применить ее для проверки нескольких шаблонов, например `abc`, `x.*x`, `(.*)`, `\([^)]*\)` или `\w+ \w+( Jr\.)?`.

23.9. Сопоставление регулярных выражений

Регулярные выражения в основном используются в двух ситуациях.

- *Поиск* строки, соответствующей регулярному выражению в (произвольно длинном) потоке данных, — функция `regex_search()` ищет этот шаблон как подстроку в потоке.
- *Сопоставление* регулярного выражения строке (известного размера) — функция `regex_match()` ищет полное соответствие шаблона и строки.

Одним из примеров является поиск почтовых индексов в разделе 23.6. Рассмотрим извлечение данных из следующей таблицы.

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27
6A	10	9	19
6B	9	10	19
7A	7	19	26
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0MO	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	234	202	386

Эта таблица (количество учеников в 2007 году в средней школе, в которой учился Бьярне Страуструп) извлечена из контекста (с веб-страницы) и представляет собой типичную и не очень сложную разновидность анализируемых данных.

- Содержит числовые поля.
- Содержит символьные поля в строках, понятных только людям, знающим контекст, из которого извлечена таблица. (В данном случае ее могут понять только люди, знающие датский язык.)
- Символьные строки содержат пробелы.
- "Поля" отделены друг от друга разделителем, роль которого в данном случае играет символ табуляции.

Мы назвали эту таблицу типичной и не очень сложной, но следует иметь в виду, что одна тонкость в ней все же скрывается: на самом деле визуально мы не можем различить пробелы и знаки табуляции; эту задачу будет решать наш код.

Проиллюстрируем использование регулярных выражений для решения следующих задач.

- Убедимся, что таблица сформирована правильно (т.е. каждая строка имеет правильное количество полей).
- Убедимся, что суммы подсчитаны правильно (в последней строке содержатся суммы чисел по столбцам).

Если мы сможем это сделать, то сможем сделать почти все! Например, мы могли бы создать новую таблицу, в которой объединены строки, имеющие одинаковые первые цифры (указывающие год обучения), или проверить, увеличивается или уменьшается количество студентов с годами (см. упр. 10 и 11).

Для того чтобы проанализировать эту таблицу, нам нужны два шаблона: один — для заголовка и второй — для остальных строк.

```
regex header {R"(^[\w ]+(    [\w ]+)*$)"};
regex row {R"(^[\w ]+(       \d+)(    \d+)(    \d+)$)"};
```

Помните, мы хвалили синтаксис регулярных выражений за лаконичность и полезность, а не за легкость освоения новичками? На самом деле регулярные выражения имеют заслуженную репутацию *языка только для письма* (write-only language). Начнем с заголовка. Поскольку он не содержит никаких числовых данных, мы могли бы просто отбросить первую строку, но — исключительно для приобретения опыта — попробуем провести ее структурный анализ. Она содержит четыре словарных поля (буквенно-цифровых поля", разделенных знаками табуляции). Эти поля могут содержать пробелы, поэтому мы не можем просто использовать управляющий символ `\w`, чтобы задать эти символы. Вместо этого мы используем выражение `[\w ]`, т.е. словообразующий символ (букву, цифру или знак подчеркивания) или пробел. Один (или более) такой символ задается выражением `[\w ]+`. Мы хотим найти тот из них, который стоит в начале строки, поэтому пишем выражение `^[\w ]+`. "Шапочка" (`^`) означает "начало строки". Каждое из оставшихся полей можно выразить как знак табуляции, за которым следуют некие слова: `(  [\w ]+)`. До конца строки их может быть сколько угодно: `(  [\w ]+)*$`. Знак доллара (`$`) означает "конец строки".

Мы не можем увидеть, что знак табуляции действительно является таковым, но в данном случае он раскрывается в ходе набора текста и распознается сам.

Приступим к более интересной части упражнения: к шаблону для строк, из которых мы хотим извлекать числовые данные. Первое поле вновь имеет такой же шаблон, как и в предыдущем случае: `^[\w ]+`. За ним следуют ровно три числовых поля, перед каждым из которых стоит знак табуляции: `(  \d+)`, так что мы получаем следующий шаблон:

```
^[\w ]+(  \d+)(  \d+)(  \d+)$
```

После вставки в необработанный строковый литерал он превращается в

```
R"(^[\w ]+(  \d+)(  \d+)(  \d+)$)"
```

Теперь мы сделали все, что требовалось. Сначала проверим, правильно ли сформирована таблица.

```
int main()
{
    ifstream in {"table.txt"};      // Входной файл
    if (!in) error("Нет входного файла\n");
    string line;                    // Входной буфер
    int lineno = 0;

    regex header {R"(^[\w ]+(  [\w ]+)*$)"};
    regex row {R"(^[\w ]+(  \d+)(  \d+)(  \d+)$)"};

    if (getline(in,line)) {         // Проверка заголовка
        smatch matches;
        if (!regex_match(line, matches, header))
            error("нет заголовка");
    }

    while (getline(in,line)) {      // Проверка данных
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            error("Неверная строка",to_string(lineno));
    }
}
```

Для краткости мы не привели здесь директивы `#include`. Мы проверяем все символы в каждой строке, поэтому вызываем функцию `regex_match()`, а не `regex_search()`. Разница между ними заключается только в том, что функция `regex_match()` должна сопоставлять с шаблоном каждый символ из потока ввода, а функция `regex_search()` проверяет поток ввода, пытаясь найти соответствующую подстроку. Ошибочное использование функции `regex_match()`, когда подразумевалось использование функции `regex_search()` (и наоборот), может оказаться самой трудно обнаруживаемой ошибкой. Однако обе эти функции используют свои аргументы совершенно одинаково.

Теперь можем перейти к верификации данных в таблице. Мы подсчитаем количество мальчиков ("drenge") и девочек ("piger"), учащихся в школе. Для каждой строки мы проверяем, действительно ли в последнем поле ("ELEVER IALT") записана сумма первых двух полей. Последняя строка ("Alle klasser") содержит суммы по столбцам. Для проверки этого факта модифицируем выражение `row` так, чтобы текстовое поле содержало совпадение с подшаблоном и можно было распознать строку "Alle klasser".

```
int main()
{
    ifstream in {"table.txt"};  // Входной файл
    if (!in) error("Нет входного файла");

    string line;                // Входной буфер
    int lineno = 0;

    regex header {R"(^[\w ]+(   [\w ]+)*$)"};
    regex row {R"(^[\w ]+(   \d+)(   \d+)(   \d+)$)"};

    if (getline(in,line)) {     // Проверка заголовка
        smatch matches;
        if (regex_match(line, matches, header)) {
            error("Нет заголовка");
        }
    }

    // Итог по столбцам:
    int boys = 0;
    int girls = 0;

    while (getline(in,line)) {
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            cerr << "Неверная строка: " << lineno << '\n';

        if (in.eof()) cout << "Конец файла\n";

        // Проверка строки:
        int curr_boy   = from_string<int>(matches[2]);
        int curr_girl  = from_string<int>(matches[3]);
        int curr_total = from_string<int>(matches[4]);
        if (curr_boy+curr_girl != curr_total)
            error("Неверная сумма в строке\n");

        if (matches[1]=="Alle klasser") { // Последняя строка
            if (curr_boy != boys)
                error("Количество мальчиков не сходится\n");
            if (curr_girl != girls)
                error("Количество девочек не сходится\n");
```

```
            if (!(in>>ws).eof())
                error("символы после последней строки");
            return 0;
        }
        // Обновление итоговых значений:
        boys  += curr_boy;
        girls += curr_girl;
    }
    error("Нет итоговой строки");
}
```

Последняя строка по смыслу отличается от остальных: в ней содержатся итоговые суммы. Мы распознаем ее по метке ("Alle klasser"). Мы решили, что после последнего символа не должны стоять символы, не являющиеся пробельными (для распознавания этого факта используется методика из функции `to<>()`; см. раздел 23.2), и выдаем сообщение об ошибке в случае их обнаружения.

Для того чтобы извлечь числа из полей данных, мы использовали функцию `from_string()` из раздела 23.2. Мы уже убедились, что эти поля содержат только цифры, поэтому проверять правильность преобразования объекта класса `string` в переменную типа `int` необязательно.

23.10. Ссылки

Регулярные выражения — популярный и полезный инструмент, доступный во многих языках программирования и во многих форматах. Они поддерживаются элегантной теорией, основанной на формальных языках, и эффективной технологией реализации, основанной на конечных автоматах. Описание регулярных выражений, их теории, реализации и использования конечных автоматов выходит за рамки рассмотрения настоящей книги. Однако, поскольку это стандартная тема в компьютерных науках, а регулярные выражения очень популярны, найти дополнительную информацию при необходимости не составит труда.

Перечислим некоторые из этих источников.

1. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (обычно именуемая "Книга дракона"). Addison-Wesley, 2007. ISBN 0321547985.[1]
2. Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, ...)". `http://swtch.com/~rsc/regexp/regexp1.html`.

[1] Имеется русский перевод: Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. *Компиляторы: принципы, технологии и инструментарий.* — М: Издательский дом "Вильямс", 2008. ISBN 978-5-8459-1349-4.

3. Maddoc, J. boost::regex documentation. `www.boost.org/libs/regex/doc/index.html`.
4. Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning PERL, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.

Задание

1. Выясните, является ли библиотека `regex` частью вашей стандартной библиотеки. *Указание*: ищите `std::regex` и `tr1::regex`.
2. Запустите небольшую программу из раздела 23.7; для этого может понадобиться разобраться в настройках проекта и/или ключах командной строки для установления связи с библиотекой `regex` и использования соответствующего заголовочного файла.
3. Используйте программу из задания 2 для проверки шаблонов из раздела 23.7.

Контрольные вопросы

1. Где мы можем найти "текст"?
2. Какие возможности стандартной библиотеки чаще всего используются для анализа текста?
3. Куда вставляет элемент функция `insert()` — перед или после указанной позиции (или итератора)?
4. Что такое Unicode?
5. Как конвертировать тип в класс `string` и обратно?
6. В чем заключается разница между инструкцией `cin>>s` и вызовом функции `getline(cin,s)`, если `s` — это объект класса `string`?
7. Перечислите стандартные потоки.
8. Что собой представляет ключ отображения `map`? Приведите примеры полезных типов ключей.
9. Как перемещаться по элементам контейнера `map`?
10. В чем заключается разница между классами `map` и `multimap`? Какой полезной операции, существующей в классе `map`, нет в классе `multimap` и почему?
11. Какие операции требуются для однонаправленного итератора?
12. В чем заключается разница между пустым и отсутствующим полем? Приведите два примера.
13. Зачем нужен символ управляющей последовательности при формировании регулярных выражений?

14. Как превратить регулярное выражение в переменную типа `regex`?
15. Какие строки соответствуют шаблону `\w+\s\d{4}`? Приведите три примера. Какой строковый литерал нужно использовать для инициализации переменной типа `regex` заданным шаблоном?
16. Как (в программе) выяснить, является ли строка корректным регулярным выражением?
17. Что делает функция `regex_search()`?
18. Что делает функция `regex_match()`?
19. Как представить символ точки (.) в регулярном выражении?
20. Как выразить понятие "не меньше трех" в регулярном выражении?
21. Относится ли символ `7` к группе `\w`? А символ _ (подчеркивания)?
22. Какое обозначение используется для символов в верхнем регистре?
23. Как задать собственный набор символов?
24. Как извлечь значение из целочисленного поля?
25. Как представить число с плавающей точкой с помощью регулярного выражения?
26. Как извлечь число с плавающей точкой из строки, соответствующей шаблону?
27. Что такое совпадение с подшаблоном? Как к нему обратиться?

Термины

`multimap`	`smatch`	регулярное выражение
`regex_match()`	подшаблон	соответствие
`regex_search()`	поиск	шаблон

Упражнения

1. Запустите программу, работающую с файлом сообщений электронной почты; протестируйте ее, используя собственный, более длинный файл. Обеспечьте наличие в этом файле сообщений, вызывающих сообщения об ошибках, например сообщения с двумя строками адреса, несколько сообщений с одинаковыми адресами и/или темами и пустые сообщения. Кроме того, протестируйте программу на примере, который вообще не является сообщением и не соответствует спецификации программы, например на файле, не содержащем строк ----.
2. Добавьте класс `multimap` и используйте его для хранения тем сообщений. Пусть программа получает входные строки с клавиатуры и выводит каждое сообщение, тема которого совпадает с заданной строкой.

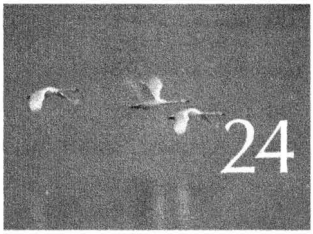
3. Модифицируйте пример из раздела 23.4 путем применения регулярных выражений для обнаружения темы и отправителя сообщения электронной почты.
4. Найдите реальный файл с сообщениями электронной почты (т.е. файл, содержащий реальные сообщения) и модифицируйте программу так, чтобы она могла извлекать темы по имени отправителя, задаваемого пользователем с клавиатуры.
5. Найдите большой файл с сообщениями электронной почты (тысячи сообщений) и замерьте время работы с ним программы с использованием сначала `multimap`, а затем — `unordered_multimap`. Обратите внимание на то, что в нашем приложении никак не используется преимущество упорядоченности объекта класса `multimap`.
6. Напишите программу, обнаруживающую даты в текстовом файле. Выведите на печать каждую строку, содержащую хотя бы одну дату, в формате **номер строки: текст**. Начните с регулярного выражения для простого формата, например 20.02.2004, и протестируйте на нем программу. Затем добавьте новые форматы.
7. Напишите программу (аналогичную предыдущей), которая находит в файле номера кредитных карточек. Разберитесь в том, какие форматы используются для записи номеров кредитных карточек, и реализуйте их проверку в вашей программе.
8. Модифицируйте программу из раздела 23.8.7 так, чтобы на ее вход поступали шаблон и имя файла. Результатом работы программы должны быть пронумерованные строки (**номер строки: текст**), соответствующие шаблону. Если соответствия не выявлены, ничего выводить не надо.
9. Используя функцию `eof()` (раздел Б.7.2), можно определить, какая строка в таблице является последней. Используйте эту функцию для упрощения программы, анализирующей таблицу (см. раздел 23.9). Проверьте вашу программу на файлах, содержащих пустые строки после таблицы, а также на файлах, которые не заканчиваются переходом на новую строку.
10. Модифицируйте программу проверки таблицы из раздела 23.9 так, чтобы она выводила новую таблицу, в которой объединялись бы строки, имеющие одинаковые первые цифры.
11. Модифицируйте программу для проверки таблицы из раздела 23.9 так, чтобы выяснить, возрастает или убывает количество учеников с годами.
12. Напишите программу на основе программы, выявляющей строки с датами (упр. 6), найдите все даты и переведите их в формат ISO

`yyyy-mm-dd`. Эта программа должна получать на вход файл и выводить результаты в выходной файл, идентичный входному, за одним исключением: даты в нем записаны в другом формате.

13. Соответствует ли точка (`.`) шаблону `'\n'`? Напишите программу, которая отвечает на этот вопрос.
14. Напишите программу, которую, подобно программе из раздела 23.8.7, можно использовать для экспериментов с сопоставлением шаблонов с помощью их ввода извне. Однако теперь программа должна считывать данные из файла в память (представляя разделение на строки с помощью символа перехода на новую строку `'\n'`), чтобы можно было экспериментировать с шаблонами, содержащими разрывы строк. Протестируйте программу на десятке таких шаблонов.
15. Опишите шаблон, который нельзя представить с помощью регулярного выражения.
16. Только для экспертов: докажите, что шаблон из предыдущего упражнения действительно не является регулярным выражением.

Послесловие

Легко впасть в заблуждение, считая, что компьютеры и вычисления имеют дело только с числами, что вычисления являются разновидностью математики. Очевидно, это не так. Просто посмотрите на экран компьютера; он заполнен текстом и пикселями. Может быть, ваш компьютер еще и проигрывает музыку. Для каждого приложения важно выбрать правильный инструмент. В контексте языка C++ это значит правильно выбрать подходящую библиотеку. Для работы с текстом ключевым инструментом зачастую является библиотека регулярных выражений. Кроме того, не следует забывать и об ассоциативных контейнерах и стандартных алгоритмах.



24

Числа

Любая сложная задача имеет решение — ясное, простое и неправильное.

— Г.Л. Менкен (H.L. Mencken)

В этой главе представлен обзор основных инструментов для численных расчетов, предоставляемых языком и его библиотекой. Мы рассмотрим фундаментальные проблемы с размером, точностью и округлением чисел. В центре внимания этой главы — многомерные массивы как в стиле языка C, так и из библиотеки N-мерных матриц. Здесь также описана генерация случайных чисел, которая часто необходима для тестирования, моделирования и игр. В заключение будут упомянуты стандартные математические функции и кратко изложены основные функциональные возможности комплексных чисел из стандартной библиотеки.

24.1. Введение

Для некоторых категорий людей (например, ученых, инженеров или статистиков) основным занятием являются серьезные числовые расчеты. В работе многих людей такие расчеты играют значительную роль. К этой категории относятся, например, специалисты по компьютерным наукам, иногда работающие с физиками. У большинства людей необходимость в числовых расчетах, выходящая за рамки простых арифметических действий над целыми числами и числами с плавающей точкой, возникает достаточно редко. Цель данной главы — описать языковые возможности C++, необходимые для решения простых вычислительных задач. Мы не пытаемся учить читателей численному анализу или тонкостям операций над числами с плавающей точкой; эти темы выходят за рамки рассмотрения нашей книги и тесно связаны с конкретными приложениями. Здесь мы собираемся рассмотреть следующие темы.

- Вопросы, связанные с встроенными типами фиксированного размера (например, такие как точность и переполнение).
- Массивы, как в стиле языка C, так и библиотека `Matrix`, которая лучше подходит для числовых расчетов.
- Введение в случайные числа.
- Математические функции из стандартной библиотеки.
- Комплексные числа.

Основное внимание уделено многомерным массивам в стиле языка C и библиотеке `Matrix`, которая позволяет упростить работу с матрицами (многомерными массивами).

24.2. Размер, точность и переполнение

Когда вы используете встроенные типы и обычные методы вычислений, числа хранятся в областях памяти фиксированного размера; иначе говоря, целочисленные типы (`int`, `long` и др.) представляют собой лишь приближение математического понятия целых чисел, а числа с плавающей точкой (`float`, `double` и др.) являются лишь приближением математического понятия действительных чисел. Отсюда следует, что с математической точки зрения некоторые вычисления оказываются неточными или неправильными. Рассмотрим пример.

```
float x = 1.0/333;
float sum = 0;
for (int i=0; i<333; ++i) sum+=x;
cout << setprecision(15) << sum << "\n";
```

Выполнив эту программу, мы получим не единицу, а

```
0.999999463558197
```

Мы ожидали чего-то подобного. Число с плавающей точкой состоит только из фиксированного количества битов, поэтому мы всегда можем "испортить" его, выполнив вычисление, результат которого состоит из большего количества битов, чем допускает аппаратное обеспечение. Например, рациональное число 1/3 невозможно представить точно как десятичное число (однако можно использовать много цифр его десятичного представления). Точно так же невозможно точно представить число 1/333, поэтому, когда мы складываем 333 копии числа `x` (наилучшее машинное приближение числа 1/333 с помощью типа `float`), то получаем число, немного отличающееся от единицы. При интенсивном использовании чисел с плавающей точкой возникает ошибка округления; остается лишь оценить, насколько сильно она влияет на результат.

Всегда проверяйте, насколько точными являются результаты. При вычислениях вы должны представлять себе, каким должен быть результат, иначе легко будете обмануты ошибкой вычислений или какой-то глупой ошибкой. Помните об ошибках округления и, если сомневаетесь, обратитесь за советом к эксперту или почитайте учебники по численным методам.

ПОПРОБУЙТЕ

Замените в примере число 333 числом 10 и снова выполните программу. Какой результат следовало ожидать? Какой результат вы получили? А ведь мы предупреждали!

Влияние фиксированного размера целых чисел может проявиться более впечатляюще. Дело в том, что числа с плавающей точкой по определению

являются приближениями действительных чисел, поэтому они могут терять точность (т.е. терять самые младшие значащие биты). Целые же числа часто переполняются (т.е. теряют самые старшие значащие биты). В итоге ошибки, связанные с числами с плавающей точкой, имеют более сложный характер (которые новички часто не замечают), а ошибки, связанные с целыми числами, бросаются в глаза (их трудно не заметить даже новичку). Мы предпочитаем, чтобы ошибки проявлялись как можно раньше, тогда их легче исправить.

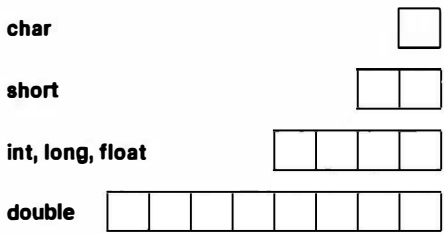
Рассмотрим целочисленную задачу.

```
short int y = 40000;
int i = 1000000;
cout << y << " " << i*i << "\n";
```

Выполнив эту программу, получим следующий результат:

```
-25536 -727379968
```

Этого и следовало ожидать. Здесь мы видим эффект переполнения. Целочисленные типы позволяют представить лишь относительно небольшие целые числа. Нам просто не хватит битов, чтобы точно представить каждое необходимое нам целое число способом, позволяющим выполнять эффективные вычисления. В данном случае двухбайтовое целое типа **short** не может представить значение 40 000, а четырехбайтовое целое типа **int** не может представить число 1 000 000 000 000. Точные размеры встроенных типов в языке C++ (раздел A.8) зависят от аппаратного обеспечения и компилятора; размер переменной **x** или типа **x** в байтах можно определить с помощью оператора **sizeof(x)**. По определению **sizeof(char)==1**. Это можно проиллюстрировать следующим образом.



Это размеры для операционной системы Windows и компилятора компании Microsoft. В языке C++ есть много способов представить целые числа и числа с плавающей точкой, используя разные размеры, но если у вас нет весомых причин поступать иначе, лучше придерживаться типов **char**, **int** и **double**. В большинстве программ (но, разумеется, не во всех) остальные целочисленные типы и типы с плавающей точкой вызывают больше проблем, чем дают преимуществ.

Целое число можно присвоить переменной, имеющей тип с плавающей точкой. Если целое число окажется больше, чем может представить

данный тип числа с плавающей точкой, произойдет потеря точности. Рассмотрим пример.

```
cout << "Размеры: " << sizeof(int)
            << '  ' << sizeof(float) << '\n';
int x = 2100000009;     // Большое целое число
float f = x;
cout << x << '  ' << f << endl;
cout << setprecision(15) << x << '  ' << f << '\n';
```

На нашем компьютере мы получили следующий результат:

```
Размеры: 4  4
2100000009  2.1e+009
2100000009  2100000000
```

Типы `float` и `int` занимают одинаковое количество памяти (4 байта). Тип `float` состоит из мантиссы (как правило, значения между нулем и единицей) и показателя степени (т.е. число имеет вид *мантисса*·$10^{\text{показатель степени}}$), поэтому он не может точно выразить самое большое число `int`. (Если бы мы попытались сделать это, то откуда бы мы взяли достаточное количество памяти для мантиссы после размещения показателя степени?) Как и следовало ожидать, переменная `f` представляет число `2100000009` настолько точно, насколько возможно. Однако последняя цифра `9` слишком велика для точного представления — именно поэтому мы и выбрали для иллюстрации это число.

С другой стороны, когда мы присваиваем число с плавающей точкой переменной целочисленного типа, происходит усечение; иначе говоря, дробная часть — цифры после десятичной точки — просто отбрасываются. Рассмотрим пример.

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

Значение переменной `x` оказывается равным `2`. Оно не будет равно `3`, как вы могли подумать, если применили "правило округления 4/5". В языке C++ преобразование типа `float` в тип `int` выполняется с помощью усечения, а не округления.

При вычислениях следует опасаться возможного переполнения и усечения. Язык C++ не решит эту проблему вместо вас. Рассмотрим пример.

```
void f(int i, double fpd)
{
    char c = i;       // Тип char действительно представляет
                      // очень маленькие целые числа
    short s = i;      // Опасно: тип int может не поместиться
                      // в памяти для переменной типа short
    i = i+1;          // Что будет, если число i до увеличения
```

```
                    // было максимально возможным?
    long lg = i*i;  // Опасно: long может быть не больше int
    float fps = fpd; // Опасно: значение типа double может
                    // не поместиться в типе float
    i = fpd;        // Усечение; например, 5.7 -> 5
    fps = i;        // Возможна потеря точности (при очень
                    // больших целочисленных значениях)
}

void g()
{
    char ch = 0;
    for (int i = 0; i<500; ++i)
        cout << int(ch++) << '\t';
}
```

Если сомневаетесь, поэкспериментируйте! Не следует отчаиваться и не следует ограничиваться чтением документации. Без экспериментов вы можете не понять содержание весьма сложной документации, касающейся числовых типов.

ПОПРОБУЙТЕ

Выполните функцию `g()`. Модифицируйте функцию `f()` так, чтобы она выводила на печать переменные `c`, `s`, `i` и т.д. Протестируйте программу при разных значениях.

Представление целых чисел и их преобразование еще будет рассматриваться в разделе 25.5.3. Где это возможно, мы стараемся ограничиваться немногими типами данных. Это позволяет минимизировать вероятность ошибок. Например, используя только тип `double` и избегая типа `float`, мы минимизируем вероятность возникновения проблем, связанных с преобразованием `double` во `float`. Фактически мы предпочитаем использовать для вычислений только типы `int`, `double` и `complex` (см. раздел 24.9), для символов — тип `char` и для логических значений — тип `bool`. Остальные арифметические типы мы используем только при крайней необходимости.

24.2.1. Пределы числовых диапазонов

Каждая реализация языка C++ определяет свойства встроенных типов в заголовочных файлах `<limits>`, `<climits>`, `<limits.h>` и `<float.h>`, чтобы программисты могли проверить пределы диапазонов, установить ограничители и т.д. Эти значения перечислены в разделе Б.9.1. Они критично важны при создании низкоуровневых инструментов. Если вас интересуют эти значения, значит, вы работаете непосред-

ственно с аппаратным обеспечением, хотя существуют и другие применения этой информации. Например, довольно часто возникают вопросы о тонкостях реализации языка, например "Насколько большим является тип `int`?" или "Имеет ли знак тип `char`?" Найти определенные и правильные ответы в системной документации бывает трудно, а в стандарте указаны только минимальные требования. Однако можно легко написать программу, находящую ответы на эти вопросы.

```
cout << "Количество байтов в типе int: "
     << sizeof(int) << '\n';
cout << "Наибольшее число типа int: "
     << INT_MAX << endl;
cout << "Наименьшее число типа int: "
     << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
    cout << "Тип char имеет знак\n";
else
    cout << "Тип char не имеет знака\n";

// Наименьшее положительное значение:
char ch = numeric_limits<char>::min();
cout << "Наименьшее положительное значение char: "
     << ch << '\n';
cout << "Значение int для char с наименьшим "
     << "положительным значением: " << int(ch) << '\n';
```

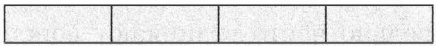
Если вы пишете программу, которая должна работать на разном аппаратном обеспечении, иногда возникает необходимость сделать эту информацию доступной для вашей программы. Иначе вам придется "прошить" ответы в программе, усложнив ее сопровождение.

Эти пределы могут быть также полезными для выявления переполнения.

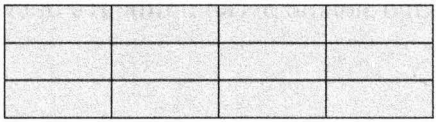
24.3. Массивы

Массив (array) — это последовательность элементов, в которой доступ к каждому элементу осуществляется с помощью его индекса (позиции). Синонимом этого понятия является *вектор* (vector). В этом разделе мы уделим внимание многомерным массивам, элементами которых также являются массивы. Обычно многомерный массив называют *матрицей* (matrix). Разнообразие синонимов свидетельствует о популярности и полезности этого общего понятия. Стандартные классы `vector` (раздел Б.4), `array` (раздел 20.9), а также встроенный массив (раздел А.8.2) являются одномерными. А что если нам нужен двумерный массив (например, матрица)? А если нам нужны семь измерений?

Проиллюстрировать одно- и двумерные массивы можно так.



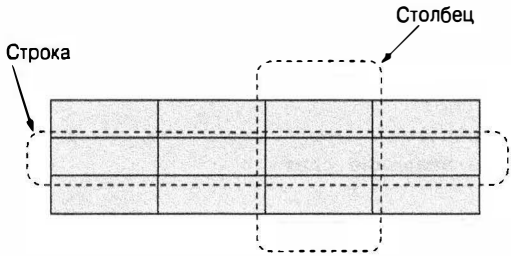
Вектор (например, Matrix<int> v(4));
именуется также одномерным
массивом или даже матрицей 1xN



Матрица 3x4 (например, Matrix<int,2> m(3,4));
именуется также двумерным массивом

Массивы имеют фундаментальное значение в большинстве вычислительных задач. Наиболее интересные научные, технические, статистические и финансовые вычисления тесно связаны с массивами.

Часто, говоря о массиве, его рассматривают как состоящий из строк и столбцов.



Матрица 3x4
(например, Matrix<int,2> m(3,4));
именуется также двумерным массивом.
3 строки, 4 столбца

Столбец — это последовательность элементов, имеющих одинаковые первые координаты (x-координаты). Строка — это множество элементов, имеющих одинаковые вторые координаты (y-координаты).

24.4. Многомерные массивы в стиле языка C

Встроенный массив C++ можно использовать как многомерный массив. В этом случае многомерный массив интерпретируется как массив массивов, т.е. массив, элементами которого являются массивы. Рассмотрим примеры.

```
int ai[4];          // Одномерный массив
double ad[3][4];    // Двумерный массив
char ac[3][4][5]; // Трехмерный массив
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

Этот подход наследует все преимущества и недостатки одномерного массива.

•Преимущества

- Непосредственное отображение на аппаратное обеспечение.

- Эффективен для низкоуровневых операций.
- Непосредственная языковая поддержка.

- Проблемы
  - Многомерные С-массивы являются массивами массивов (см. ниже).
  - Фиксированные на этапе компиляции размеры. Если вы хотите определять размер массива на этапе выполнения программы, используйте динамическую память.
  - Массивы невозможно аккуратно передать в функцию. При малейшей возможности массив превращается в указатель на свой первый элемент.
  - Нет проверки диапазона. Как обычно, массив не знает своего размера.
  - Нет операций над массивами, даже присваивания (копирования).

Встроенные массивы широко используются в числовых расчетах. Они также являются *основным* источником ошибок и сложностей. Создание и отладка таких программ у большинства людей вызывают головную боль. Если вы вынуждены использовать встроенные массивы, почитайте учебники (например, *The C++ Programming Language*). К сожалению, язык C++ использует многомерные массивы совместно с языком C, так что имеется масса "внешних" исходных текстов, которые их используют.

Основная проблема заключается в невозможности аккуратной передачи многомерного массива в функцию, так что приходится работать с указателями и выполнять явные вычисления, связанные с определением позиций в многомерном массиве. Рассмотрим пример.

```
void f1(int a[3][5]);              // Только для матриц [3][5]

void f2(int[][5], int dim1);       // Первая размерность может быть переменной

void f3(int [5][], int dim2);      // Ошибка: вторая размерность
                                   // не может быть переменной

void f4(int[][], int dim1, int dim2); // Ошибка (не будет
                                   // работать в любом случае)
void f5(int* m, int dim1, int dim2)    // Как ни странно,
{                                      // но работает
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Здесь мы передаем массив `m` как указатель `int*`, несмотря на то что на самом деле это двумерный массив. Поскольку вторая размерность должна быть переменной (параметром), у нас нет никакой возможности сообщить компилятору, что массив `m` является массивом с размерностями

(dim1,dim2), так что мы просто передаем указатель на начало памяти, в которой он хранится. Выражение m[i*dim2+j] на самом деле означает m[i,j], но, поскольку компилятор не знает, что m является двумерным массивом, мы должны сами вычислять позицию элемента m[i,j] в памяти.

Этот способ слишком сложен, примитивен и уязвим для ошибок. Он также слишком медленный, поскольку явное вычисление позиции элемента усложняет оптимизацию. Вместо того чтобы учить вас, как справиться с этой ситуацией, мы сконцентрируемся на библиотеке C++, которая вообще устраняет проблемы, связанные с встроенными массивами.

24.5. Библиотека Matrix

Что мы вообще хотим от массива/матрицы в численных расчетах?

- "Мой код должен выглядеть очень похожим на описание массивов, изложенное в большинстве учебников по математике".
  - Это относится также к векторам, матрицам и тензорам.
- Проверка на этапах компиляции и выполнения программы.
  - Массивы любой размерности.
  - Массивы с произвольным количеством элементов в любой размерности.
- Массивы являются полноценными переменными/объектами.
  - Их можно передавать куда угодно.
- Обычные операции над массивами.
  - Индексирование: ().
  - Срезка: [].
  - Присваивание: =.
  - Операции масштабирования (+=, -=, *=, %= и т.д.).
  - Смешанные векторные операции (например, res[i] = a[i]*c+b[2]).
  - Скалярное произведение (res = сумма a[i]*b[i]; известно также как inner_product).
- Должно обеспечиваться преобразование традиционных обозначений, связанных с массивами/векторами, в код, который иначе вы должны были бы с большими затратами труда писать самостоятельно (и обеспечивать как минимум такую же эффективность).
- Массивы при необходимости можно увеличивать (при их реализации не используются "магические" числа).

Библиотека Matrix делает это и только это. Если вы хотите большего, то должны самостоятельно написать сложные функции обработки массивов,

разреженных массивов, управления распределением памяти и так далее или использовать другую библиотеку, которая лучше соответствует вашим потребностям. Однако многие из этих потребностей можно удовлетворить с помощью алгоритмов и структур данных, надстроенных над библиотекой **Matrix**. Библиотека **Matrix** не является частью стандарта ISO C++. Вы можете найти ее на сайте поддержки книги в заголовочном файле **Matrix.h**. Ее возможности определены в пространстве имен **Numeric_lib**. Мы выбрали название **Matrix**, потому что слова "вектор" и "массив" и без того чрезмерно используются в библиотеках языка C++. Реализация библиотеки **Matrix** использует сложные методы, которые здесь не описываются.

24.5.1. Размерности и доступ

Рассмотрим простой пример.

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1); // Одномерная с элементами double
    Matrix<int,1> ai1(n1);    // Одномерная с элементами int
    ad1(7) = 0;      // Индексирование в стиле Fortran – ()
    ad1[7] = 8;      // Индексирование в стиле C – []

    Matrix<double,2> ad2(n1,n2);     // Двумерная
    Matrix<double,3> ad3(n1,n2,n3); // Трехмерная
    ad2(3,4) = 7.5;                  // Истинное многомерное
    ad3(3,4,5) = 9.2;                // индексирование
}
```

Таким образом, определяя матрицу (объект класса **Matrix**), вы должны указать тип элемента и количество размерностей. Очевидно, что класс **Matrix** является шаблонным, а тип элементов и количество размерностей представляют собой шаблонные параметры. Результатом передачи пары шаблонных параметров классу **Matrix** (например, **Matrix<double,2>**) является тип (класс), с помощью которого можно определить объекты, указав аргументы (например, **Matrix<double,2> ad2(n1,n2)**), которые задают размерности. Так, переменная **ad2** является двумерным массивом с размерностями **n1** и **n2**, которую также называют матрицей **n1×n2**. Для того чтобы получить элемент объявленного типа из одномерного объекта класса **Matrix**, следует указать один индекс. Для того чтобы получить элемент объявленного типа из двумерного объекта класса **Matrix**, следует указать два индекса, и т.д.

Как и во встроенных массивах и векторах **vector**, элементы в объекте класса **Matrix** индексируются с нуля (а не с единицы, как в языке Fortran);

иначе говоря, элементы **Matrix** нумеруются в диапазоне [0,max), где max — количество элементов.

Это просто и "взято прямо из учебника". Если у вас возникнут проблемы, нужно лишь обратиться к соответствующему учебнику по математике, а не к руководству по программированию. Единственная тонкость здесь заключается в том, что можно не указывать количество размерностей в **Matrix**: по умолчанию матрица является одномерной. Обратите внимание также на то, что мы можем использовать индексирование как с помощью оператора **[]** (в стиле языков C и C++), так и с помощью оператора **()** (в стиле языка Fortran).

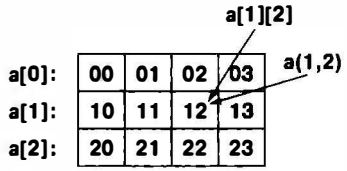
Это позволяет нам лучше справляться с большим количеством размерностей. Индекс **[x]** всегда означает единственный индекс, выделяя отдельную строку в объекте класса **Matrix**; если переменная **a** является *n*-мерной матрицей **Matrix**, то **a[x]** — это *n*–1-мерная матрица. Обозначение же **(x,y,z)** подразумевает одновременное использование нескольких индексов, выделяя соответствующий элемент объекта класса **Matrix**; количество индексов должно равняться количеству размерностей.

Посмотрим, что произойдет, если мы сделаем ошибку.

```
void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0;          // Ошибка: 0-мерная матрица
    Matrix<double,1> ad1(5);
    Matrix<int,1> ai(5);

     Matrix<double,1> ad11(7);
    ad1(7) = 0;                 // Исключение Matrix_error
                                // (7 - за пределами диапазона)
    ad1 = ai;                   // Ошибка: разные типы элементов
    ad1 = ad11;                 // Исключение Matrix_error
                                // (несовпадающие размерности)

    Matrix<double,2> ad2(n1); // Ошибка: не указана вторая размерность
    ad2(3) = 7.5;               // Ошибка: неверное количество индексов
    ad2(1,2,3) = 7.5;           // Ошибка: неверное количество индексов

    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad3 = ad33;                 // OK: одинаковые типы элементов,
                                // одинаковые размерности
}
```

Несоответствия между объявленным количеством размерностей и их использованием обнаруживается на этапе компиляции. Выход за пределы диапазона перехватывается на этапе выполнения программы; при этом генерируется исключение **Matrix_error**.

Первая размерность матрицы — это строка, а вторая — столбец, поэтому мы индексируем двумерную матрицу (двумерный массив) как **(строка, столбец)**. Можно также использовать обозначение **[строка][столбец]**, так как индексирование двумерной матрицы с помощью одномерного индекса порождает одномерную матрицу — строку. Эту ситуацию можно проиллюстрировать следующим образом.



Элементы матрицы размещаются в памяти построчно.

00	01	02	03	10	11	12	13	20	21	22	23

Поскольку класс **Matrix** знает свою размерность, его элементы можно передавать как аргумент без проблем:

```
void init(Matrix<int,2>& a) // Инициализация каждого элемента
{                           // характеризующим значением
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>& a) // Построчный вывод матрицы
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) <<'\t';
        cout << '\n';
    }
}
```

Вызов **dim1()** возвращает количество элементов в первой размерности, **dim2()** — количество элементов во второй размерности и т.д. Тип элементов и количество размерностей являются частью типа **Matrix**, поэтому невозможно написать функцию, получающую произвольную матрицу **Matrix** как аргумент (но можно написать соответствующий шаблон).

```
void init(Matrix& a); // Ошибка: пропущены тип элементов
                      // и количество размерностей
```

Обратите внимание на то, что библиотека **Matrix** не содержит матричных операций, таких как сложение двух четырехмерных матриц или умножение двумерных матриц на одномерные. Элегантная и эффективная

реализация этих операций выходит за рамки данной библиотеки. Соответствующие матричные библиотеки можно надстроить над библиотекой **Matrix** (см. упр. 12).

24.5.2. Одномерная матрица

Что можно сделать с простейшей матрицей — одномерной?

Количество размерностей в объявлении такого объекта можно не указывать, потому что по умолчанию оно равно единице.

```
Matrix<int,1> a1(8); // a1 – одномерная матрица целых чисел
Matrix<int> a(8);    // Означает Matrix<int,1> a(8);
```

Таким образом, объекты **a** и **a1** имеют одинаковый тип (**Matrix<int,1>**). У каждого объекта класса **Matrix** можно запросить общее количество элементов и количество элементов в определенном измерении. У одномерной матрицы эти параметры совпадают.

```
a.size(); // Количество элементов в Matrix
a.dim1(); // Количество элементов в первом измерении
```

Можно также обращаться к элементам матрицы, используя схему их размещения в памяти, т.е. через указатель на первый элемент.

```
int* p = a.data(); // Получаем данные через указатель на массив
```

Это полезно при передаче объектов класса **Matrix** функциям в стиле языка C, принимающим в качестве аргументов указатели. Матрицы можно индексировать.

```
a(i);   // i-й элемент (в стиле Fortran) с проверкой диапазона
a[i];   // i-й элемент (в стиле C) с проверкой диапазона
a(1,2); // Ошибка: a – одномерная матрица Matrix
```

Многие алгоритмы обращаются к части матрицы **Matrix**. Такая часть называется "срезкой" (slice), представляет собой подматрицу, или диапазон элементов, и создается функцией **slice()**. В классе **Matrix** есть два варианта этой функции.

```
a.slice(i);   // Элементы с a[i] по последний
a.slice(i,n); // n элементов с a[i] по a[i+n-1]
```

Индексы и срезки можно использовать как в левой части оператора присваивания, так и в правой. Они ссылаются на элементы объекта класса **Matrix**, не создавая их копии. Рассмотрим пример.

```
a.slice(4,4) = a.slice(0,4); // Присваиваем первую половину матрицы второй
```

Например, если объект **a** вначале выглядел как

```
{ 1 2 3 4 5 6 7 8 }
```

то в результате мы получим

```
{ 1 2 3 4 1 2 3 4 }
```

Обратите внимание на то, что чаще всего срезки задаются начальным и конечным элементами **Matrix**; т.е. **a.slice(0,j)** — это диапазон [0,j), а **a.slice(j)** — диапазон [j,**a.size()**). В частности, приведенный выше пример можно легко переписать:

```
a.slice(4) = a.slice(0,4); // Присваиваем первую половину
                           // матрицы второй
```

Иначе говоря, обозначения — дело вкуса.

Вы можете указать такие индексы **i** и **n**, что **a.slice(i,n)** выйдет за пределы диапазона матрицы **a**. Однако полученная срезка будет содержать только те элементы, которые действительно принадлежат объекту **a**. Например, срезка **a.slice(i,a.size())** означает диапазон [**i**,**a.size()**), а **a.slice(a.size())** и **a.slice(a.size(),2)** — пустые объекты класса **Matrix**. Это оказывается полезным во многих алгоритмах. Мы заимствовали такое соглашение из математики. Очевидно, что срезка **a.slice(i,0)** является пустым объектом класса **Matrix**. Мы бы не писали это преднамеренно, но существуют алгоритмы, которые становятся проще, если срезка **a.slice(i,n)** при параметре **n**, равном **0**, является пустой матрицей (а не приводит к ошибке, которой требовалось бы избегать).

При обычном копировании (объектов C++) выполняется копирование всех элементов.

```
Matrix<int> a2 = a; // Копирующая инициализация
a = a2;             // Копирующее присваивание
```

Можно применять встроенные операции к каждому элементу **Matrix**:

```
a *= 7; // a[i]*=7 для всех i (работает также
        // для +=, -=, /= и т.д.)
a = 7;  // a[i]=7 для всех i
```

Это относится к каждому оператору присваивания и каждому составному оператору присваивания (**=**, **+=**, **-=**, **/=**, ***=**, **%=**, **^=**, **&=**, **|=**, **>>=**, **<<=**) при условии, что тип элемента поддерживает соответствующий оператор. Кроме того, можно применять функции к каждому элементу **Matrix**:

```
a.apply(f);   // a[i]=f(a[i]) для каждого элемента a[i]
a.apply(f,7); // a[i]=f(a[i],7) для каждого элемента a[i]
```

Составные операторы присваивания и функция **apply()** модифицируют элементы своих аргументов типа **Matrix**. Если же мы захотим создать новую матрицу, то можем использовать другой вызов:

```
b = apply(abs,a); // Создаем новую матрицу, такую, что b(i)==abs(a(i))
```

Функция **abs** — это функция вычисления абсолютной величины из стандартной библиотеки (раздел 24.8). По сути вызов **apply(f,x)** связан с вызовом **x.apply(f)** точно так же, как оператор **+** связан с оператором **+=**. Рассмотрим пример.

```
b = a*7;          // b[i] = a[i]*7 для каждого i
a *= 7;           // a[i] = a[i]*7 для каждого i
y = apply(f,x);   // y[i] = f(x[i]) для каждого i
x.apply(f);       // x[i] = f(x[i]) для каждого i
```

В результате мы получим **a==b** и **x==y**.

В языке Fortran второй вариант функции **apply** называется *функцией трансляции* (broadcast function). В этом языке чаще пишут вызов **f(x)**, а не **apply(f,x)**. Для того чтобы эта возможность стала доступной для каждой функции **f** (а не только для некоторых отдельных функций, как в языке Fortran), мы должны присвоить операции трансляции конкретное имя, поэтому (повторно) использовали имя **apply**.

Кроме того, для того чтобы обеспечить соответствие с вариантом функции-члена **apply** с двумя аргументами (**a.apply(f,x)**), мы предоставили

```
b = apply(f,a,x); // b[i]=f(a[i],x) для каждого i
```

Например:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7); // b[i] = a[i]*7 для каждого i
```

Обратите внимание на то, что "автономная" функция **apply()** принимает в качестве аргумента функцию, возвращающую результат, вычисляя его по переданным ей аргументам, а затем использует этот результат для инициализации результирующего объекта типа **Matrix**. Как правило, это не приводит к изменению матрицы, к которой эта функция применяется. Функция же член **apply()** отличается тем, что принимает в качестве аргумента функцию, модифицирующую ее аргументы; т.е. эта функция модифицирует элементы матрицы, к которым применяется. Рассмотрим пример.

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // b[i] *= 7 для каждого i
```

В классе **Matrix** предусмотрены также наиболее полезные функции из традиционных математических библиотек:

```
Matrix<int> a3 =
    scale_and_add(a,8,a2); // Слитное умножение и сложение
int r = dot_product(a3,a); // Скалярное произведение
```

Операцию **scale_and_add()** часто называют *слитным умножением и сложением* (fused multiply-add), или просто *fma*; ее определение выглядит так: **result(i)=arg1(i)*arg2+arg3(i)** для каждого **i** в матрице. Скалярное произведение, известное также под именем **inner_pro**

duct, описано в разделе 21.5.3; его определение выглядит так: `result+=arg1(i)*arg2(i)` для каждого `i` в матрице, где изначальное значение `result` обычно равно нулю.

Одномерные массивы очень широко распространены; их можно представить как в виде встроенного массива, так и с помощью классов `vector` и `Matrix`. Класс `Matrix` следует применять тогда, когда необходимо выполнять матричные операции, такие как `*=`, или когда одномерная матрица должна взаимодействовать с другими матрицами с более высокой размерностью.

Полезность этой библиотеки можно объяснить тем, что она "лучше согласована с математическими операциями", а также тем, что при ее использовании для работы с каждым элементом матрицы не приходится писать циклы. В любом случае в итоге мы получаем более короткий код и меньше возможностей сделать ошибку. Операции класса `Matrix`, такие как копирование, присваивание всем элементам и операции над всеми элементами, позволяют не использовать циклы (а значит, можно не беспокоиться о связанных с ними проблемах).

Класс `Matrix` имеет два конструктора для копирования данных из встроенных массивов в матрицы. Рассмотрим пример.

```
void some_function(double* p, int n)
{
    double val[] = { 1.2, 2.3, 3.4, 4.5 };
    Matrix<double> data{p,n};
    Matrix<double> constants{val};
    // ...
}
```

Это часто бывает полезным, когда мы получаем данные в виде обычных массивов или векторов, созданных в других частях программы, не использующих объекты класса `Matrix`.

Обратите внимание на то, что компилятор может самостоятельно определить количество элементов в инициализированном массиве, поэтому это число при определении объекта `constants` указывать необязательно — оно равно `4`. С другой стороны, если элементы заданы всего лишь указателем, то компилятор не знает их количества, поэтому при определении объекта `data` мы должны задать как указатель `p`, так и количество элементов `n`.

24.5.3. Двумерные матрицы

Общая идея библиотеки `Matrix` заключается в том, что матрицы разной размерности на самом деле в большинстве случаев очень похожи, за исключением ситуаций, в которых особую роль играют размерности. Таким образом, большинство из того, что мы можем сказать об одномерных объектах класса `Matrix`, относится и к двумерным матрицам.

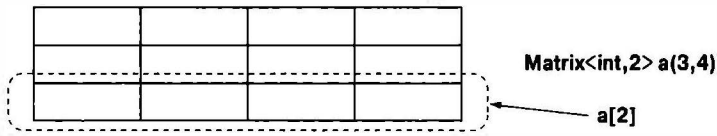
```
Matrix<int,2> a(3,4);
int s  = a.size(); // Количество элементов
int d1 = a.dim1(); // Количество элементов в строке
int d2 = a.dim2(); // Количество элементов в столбце
int* p = a.data(); // Указатель на данные
```

Мы можем запросить общее количество элементов и количество элементов в каждой размерности. Кроме того, можем получить указатель на элементы матрицы, размещенные в памяти.

Мы можем использовать индексы.

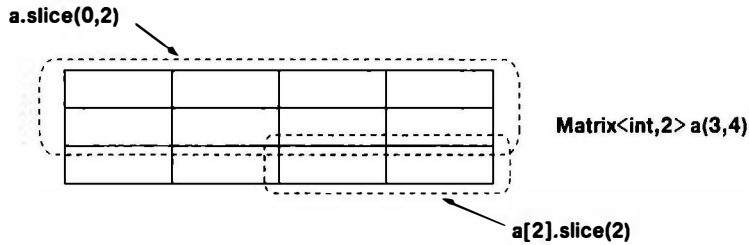
```
a(i,j);  // Элемент (i,j) (в стиле Fortran) с проверкой диапазона
a[i];    // i-я строка (в стиле C) с проверкой диапазона
a[i][j]; // Элемент (i,j) (в стиле C)
```

В двумерной матрице индексирование с помощью конструкции `[i]` создает одномерную матрицу, представляющую собой `i`-ю строку исходной матрицы. Это значит, что мы можем извлекать строки и передавать их операторам и функциям, требующим в качестве аргументов одномерные матрицы и даже встроенные массивы (с помощью `a[i].data()`). Обратите внимание на то, что индексирование вида `a(i,j)` может оказаться быстрее, чем индексирование вида `a[i][j]`, хотя это сильно зависит от компилятора и оптимизатора.



Мы можем получить срезки.

```
a.slice(i);   // Строки от a[i] до последней
a.slice(i,n); // Строки от a[i] до a[i+n-1]
```



Срезка двумерной матрицы сама является двумерной матрицей (возможно, с меньшим количеством строк).

Распределенные операции над двумерными матрицами такие же, как и над одномерными. Этим операциям не важно, как именно хранятся элементы; они просто применяются ко всем элементам в порядке их следования в памяти.

```
Matrix<int,2> a2 = a; // Копирующая инициализация
a = a2;               // Копирующее присваивание
a *= 7;               // Масштабирование (а также +=, -= и т.д.)
a.apply(f);           // a(i,j)=f(a(i,j)) для каждого a(i,j)
a.apply(f,7);         // a(i,j)=f(a(i,j),7) для каждого a(i,j)
b=apply(f,a);         // Новая матрица с b(i,j)==f(a(i,j))
b=apply(f,a,7);       // Новая матрица с b(i,j)==f(a(i,j),7)
```

Оказывается, что перестановка строк также бывает полезной, поэтому мы предусмотрели и ее.

```
a.swap_rows(1,2);     // Перестановка строк a[1] <-> a[2]
```

Перестановки столбцов **swap_columns()** нет. Если она вам потребуется, вы можете написать ее самостоятельно (см. упр. 11). Из-за построчной схемы хранения матриц в памяти строки и столбцы не совсем равноправны. Эта асимметрия проявляется также в том, что оператор **[i]** возвращает только строку (для столбцов аналогичный оператор не предусмотрен). В записи **(i,j)** первый индекс **i** выбирает строку. Эта асимметрия имеет глубокие математические корни.

Количество действий, которые можно было бы выполнить над двумерными матрицами, кажется бесконечным.

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8);    // Шахматная доска

const int white_start_row = 0;
const int black_start_row = 7;

Matrix<Piece> start_row
    = {rook, knight, bishop, queen, king, bishop, knight, rook};

Matrix<Piece> clear_row(8);    // 8 элементов со значениями по умолчанию
```

Инициализация объекта **clear_row** использует то, что **none == 0**, и то, что элементы по умолчанию инициализируются нулем.

Мы можем использовать **start_row** и **clear_row** следующим образом:

```
board[white_start_row] = start_row;  // Расстановка белых фигур
for (int i = 1; i<7; ++i)
    board[i] = clear_row;            // Очистка средины доски
board[black_start_row] = start_row;  // Расстановка черных фигур
```

Обратите внимание, что, когда мы извлекли строку, используя выражение **[i]**, мы получили lvalue (см. раздел 4.3); иначе говоря, мы можем выполнять присваивание результату операции **board[i]**.

24.5.4. Ввод-вывод матриц

Библиотека `Matrix` предоставляет *очень* простые средства ввода и вывода одно- и двумерных матриц:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

Этот фрагмент кода читает четыре разделенных пробелами числа типа `double`, заключенных в фигурные скобки; например:

```
{ 1.2 3.4 5.6 7.8 }
```

Вывод аналогичен, поэтому мы просто увидим то, что ввели.

Механизм ввода-вывода двумерных матриц просто считывает и записывает последовательности одномерных матриц, заключенные в фигурные скобки. Рассмотрим пример.

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

Этот код читает запись

```
{
{ 1 2 }
{ 3 4 }
}
```

Вывод оказывается таким же.

Операторы `<<` и `>>` из класса `Matrix` предоставлены для того, чтобы упростить написание простых программ. В более сложных ситуациях вам придется заменить их своими операторами. По этой причине определение операторов `<<` и `>>` класса `Matrix` помещено в заголовочный файл `MatrixIO.h` (а не в `Matrix.h`). Так что для того, чтобы использовать матрицы в своей программе, вам необязательно включать заголовок `MatrixIO.h`.

24.5.5. Трехмерные матрицы

По сути, трехмерные матрицы, как и матрицы более высоких размерностей, очень похожи на двумерные; просто они имеют больше размерностей. Рассмотрим пример.

```
Matrix<int,3> a(10,20,30);
a.size();              // Количество элементов
a.dim1();              // Количество элементов в размерности 1
a.dim2();              // Количество элементов в размерности 2
a.dim3();              // Количество элементов в размерности 3
int* p = a.data();     // Указатель на данные
a(i,j,k);              // (i,j,k)-й элемент (в стиле Fortran)
```

```
                        // с проверкой диапазона
a[i];                   // i-я строка (в стиле C) с проверкой
                        // диапазона
a[i][j][k];             // (i,j,k)-й элемент (в стиле C)
a.slice(i);             // Строки от i-й до последней
a.slice(i,j);           // Строки от i-й до j-й
Matrix<int,3> a2 = a;   // Копирующая инициализация
a = a2;                 // Копирующее присваивание
a *= 7;                 // Масштабирование (также +=, -= и т.д.)
a.apply(f);             // a(i,j,k)=f(a(i,j,k))
                        // для каждого элемента a(i,j,k)
a.apply(f,7);           // a(i,j,k)=f(a(i,j,k),7)
                        // для каждого элемента a(i,j,k)
b=apply(f,a);           // Создание новой матрицы, такой, что
                        // b(i,j,k)==f(a(i,j,k))
b=apply(f,a,7);         // Создание новой матрицы, такой, что
                        // b(i,j,k)==f(a(i,j,k),7)
a.swap_rows(7,9);       // Перестановка строк a[7] <-> a[9]
```

Если вы понимаете, как работать с двумерными матрицами, то сможете работать и с трехмерными. Например, здесь **a** — трехмерная матрица, поэтому **a[i]** — двумерная матрица (при условии, что индекс **i** не выходит за пределы допустимого диапазона); **a[i][j]** — матрица одномерная (при условии, что индекс **j** не выходит за пределы допустимого диапазона); ну а **a[i][j][k]** — элемент типа **int** (при условии, что индекс **k** не выходит за пределы допустимого диапазона).

Поскольку мы видим мир трехмерным, при моделировании чаще всего используются трехмерные матрицы (например, в физическом моделировании в декартовой системе координат).

```
int grid_nx;    // Разрешение сетки; задается в начале
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

Добавив в качестве четвертого измерения время, мы получим четырехмерное пространство, для описания которого нужны четырехмерные матрицы. И так далее.

Если вас интересует более совершенная версия класса **Matrix**, с поддержкой обобщенных *N*-мерных матриц, обратитесь к главе 29 книги *The C++ Programming Language*.

24.6. Пример: решение систем линейных уравнений

Код для математических вычислений имеет смысл, если вы понимаете математику, которую он выражает; если нет, код кажется полной бессмыслицей. Если вы знаете основы линейной алгебры, то приведенный ниже пример покажется вам простым; если же нет, то

просто полюбуйтесь, как решение из учебника воплощается в программе с минимальной перефразировкой.

Данный пример выбран для того, чтобы продемонстрировать реалистичное и важное использование класса `Matrix`. Мы решим систему линейных уравнений следующего вида:

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

Здесь буквы x обозначают n неизвестных, а буквы a и b — константы. Для простоты предполагаем, что неизвестные и константы являются числами с плавающей точкой.

Наша цель — найти неизвестные, которые удовлетворяют одновременно всем указанным n уравнениям. Эти уравнения с помощью матрицы и двух векторов можно выразить компактно:

$$\mathbf{Ax} = \mathbf{b}$$

Здесь $\mathbf{A}$ — квадратная матрица коэффициентов размером $n \times n$:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

Векторы $\mathbf{x}$ и $\mathbf{b}$ — векторы неизвестных и констант соответственно:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{и} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

В зависимости от матрицы $\mathbf{A}$ и вектора $\mathbf{b}$ эта система может не иметь ни одного решения, одно решение или бесконечно много решений. Существует много разных методов решения линейных систем. Мы используем классическую схему, которая называется исключением Гаусса (см., например, Кормен, Томас Х. и др. *Алгоритмы: построение и анализ, 3-е изд.*: — М: ООО "И.Д. Вильямс", 2013). Сначала мы преобразовываем матрицу $\mathbf{A}$ и вектор $\mathbf{b}$ так, что матрица $\mathbf{A}$ становится верхнетреугольной, т.е. все элементы матрицы ниже главной диагонали равны нулю. Иначе говоря, система принимает следующий вид:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Это несложно сделать. Для того чтобы элемент в позиции (i, j) стал равным нулю, необходимо умножить строку i на такую константу, чтобы элемент в

позиции (i, j) стал равным другому элементу в столбце j, скажем, $a(k, j)$. После этого мы просто вычтем одно уравнение из другого и получим $a(i, j)==0$. При этом все остальные значения в строке i изменятся соответственно.

Если все диагональные элементы окажутся ненулевыми, система имеет единственное решение, которое можно найти в ходе обратной подстановки. Сначала решим последнее уравнение (это просто):

$$a_{n,n}x_n = b_n$$

Очевидно, что значение $x[n]$ равно $b[n]/a(n,n)$. Теперь исключим строку n из системы, найдем значение $x[n-1]$ и будем продолжать процесс, пока не доберемся до значения $x[1]$. Для каждого значения n мы выполняем деление на $a(n, n)$, поэтому диагональные значения должны быть ненулевыми. Если это условие не выполняется, обратная подстановка завершится неудачей. Это значит, что система либо не имеет решения, либо имеет бесконечно много решений.

24.6.1. Классическое исключение Гаусса

Посмотрим теперь, как этот алгоритм выражается в виде кода на языке C++. Во-первых, упростим обозначения, введя удобные имена для двух типов матриц, которые собираемся использовать.

```
typedef Numeric_lib::Matrix<double,2> Matrix;
typedef Numeric_lib::Matrix<double,1> Vector;
```

Затем выразим сам алгоритм.

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
    classical_elimination(A,b);
    return back_substitution(A,b);
}
```

Иначе говоря, мы создаем копии входных матрицы `A` и вектора `b` (используя механизм передачи аргументов по значению), вызываем функцию для решения системы, а затем вычисляем результат с помощью обратной подстановки. Такое разделение задачи на части и система обозначений приняты во всех учебниках. Для того чтобы завершить программу, мы должны реализовать функции `classical_elimination()` и `back_substitution()`. Решение также можно найти в учебнике.

```
void classical_elimination(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    // Проходим от первого столбца до предпоследнего,
    // обнуляя элементы, стоящие ниже диагонали:
    for (Index j = 0; j < n-1; ++j) {
```

```
        const double pivot = A(j,j);
        if (pivot == 0) throw Elim_failure(j);

        // Обнуляем элементы, стоящие ниже диагонали в строке i
        for (Index i = j+1; i < n; ++i) {
            const double mult = A(i, j) / pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j),
                                          -mult, A[i].slice(j));
            b(i) -= mult * b(j); // Изменяем вектор b
        }
    }
}
```

Опорным (pivot) называется элемент, лежащий на диагонали в строке, которую мы в данный момент обрабатываем. Он должен быть ненулевым, потому что нам придется на него делить; если он равен нулю, генерируется исключение.

```
Vector back_substitution(const Matrix& A, const Vector& b)
{
    const Index n = A.diml();
    Vector x(n);

    for (Index i = n-1; i >= 0; --i) {
        double s = b(i)-dot_product(A[i].slice(i+1),
                                    x.slice(i+1));
        if (double m = A(i,i))
            x(i) = s/m;
        else
            throw Back_subst_failure(i);
    }
    return x;
}
```

24.6.2. Выбор опорного элемента

Для того чтобы избежать проблем с нулевыми диагональными элементами и повысить устойчивость алгоритма, можно переставить строки местами так, чтобы на диагонали не было нулевых и малых величин. Говоря "повысить устойчивость", мы имеем в виду снижение чувствительности к ошибкам округления. Однако по мере выполнения алгоритма элементы матрицы будут изменяться, поэтому перестановку строк приходится делать постоянно (иначе говоря, мы не можем лишь один раз переупорядочить матрицу, а затем применить классический алгоритм).

```
void elim_with_partial_pivot(Matrix& A, Vector& b)
{
    const Index n = A.diml();
```

```
    for (Index j = 0; j<n; ++j) {
        Index pivot_row = j;

        // Поиск подходящего опорного элемента:
        for (Index k = j+1; k<n; ++k)
            if (abs(A(k,j)) > abs(A(pivot_row,j))) pivot_row = k;

        // Обмен строк для лучшего опорного элемента:
        if (pivot_row!=j) {
            A.swap_rows(j,pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // Исключение:
        for (Index i = j+1; i<n; ++i) {
            const double pivot = A(j,j);
            if (pivot==0)
                error("Решения нет, нулевой опорный элемент");
            const double mult = A(i,j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j),
                                          -mult,A[i].slice(j));
            b(i) -= mult*b(j);
        }
    }
}
```

Для того чтобы не писать циклы явно и привести код к более традиционному виду, мы использовали функции `swap_rows()` и `scale_and_add()`.

24.6.3. Тестирование

Очевидно, что мы должны протестировать нашу программу. К счастью, сделать это несложно.

```
void solve_random_system(Index n)
{
    Matrix A = random_matrix(n); // См. раздел 24.7
    Vector b = random_vector(n);

    cout << "A = " << A << endl;
    cout << "b = " << b << endl;

    try {
        Vector x = classical_gaussian_elimination(A, b);
        cout << "Решение методом Гаусса: x = " << x << endl;
        Vector v = A * x;
        cout << " A * x = " << v << endl;
     }
    catch(const exception& e) {
        cerr << e.what() << std::endl;
    }
}
```

Попасть в раздел `catch` можно тремя путями.

- Из-за ошибки в программе (однако, будучи оптимистами, понадеемся, что этого никогда не произойдет).
- Из-за входных данных, приводящих к неработоспособности алгоритма `classical_elimination` (для многих таких входных данных функция `elim_with_partial_pivot` оказывается вполне работоспособной).
- Из-за ошибок округления.

Увы, наш тест не настолько реалистичен, как хотелось бы, поскольку случайные матрицы вряд ли вызовут проблемы у алгоритма `classical_elimination`.

Для того чтобы проверить наше решение, мы выводим на экране произведение `A*x`, которое должно быть равно вектору `b` (или достаточно близким к нему с учетом ошибок округления). Из-за вероятных ошибок округления мы не можем просто ограничиться инструкцией

```
if (A*x!=b) error("Неправильное решение");
```

Поскольку числа с плавающей точкой являются лишь приближением действительных чисел, получим лишь приближенный ответ. В общем случае не следует применять операторы == и != к результатам вычислений с плавающей точкой: такие числа являются лишь приближениями.

В библиотеке `Matrix` нет операции умножения матрицы на вектор, поэтому эту функцию нам придется написать самостоятельно.

```
Vector operator*(const Matrix& m, const Vector& u)
{
    const Index n = m.dim1();
    Vector v(n);
    for (Index i = 0; i < n; ++i)
        v(i) = dot_product(m[i], u);
    return v;
}
```

И вновь простая операция над объектом класса `Matrix` делает вместо нас большую часть работы. Как указывалось в разделе 24.5.3, операции вывода для класса `Matrix` описаны в заголовочном файле `MatrixIO.h`. Функции `random_matrix()` и `random_vector()` просто используют случайные числа (раздел 24.7) и остаются читателю в качестве упражнения. Имя `Index` является синонимом типа (раздел A.16) индекса, используемого в библиотеке `Matrix`. Этот тип вносится в область видимости с помощью объявления `using`:

```
using Numeric_lib::Index;
```

24.7. Случайные числа

Если вы попросите случайного человека назвать случайное число, то в большинстве случаев будет названо 7 или 17, так что эти числа считаются людьми "самыми случайными". Люди практически никогда не называют число нуль, так как оно кажется таким идеально круглым, что не воспринимается как случайное, поэтому его считают "наименее случайным" числом. С математической точки зрения это полная бессмыслица: ни одно отдельно взятое число нельзя назвать случайным. То, что нам обычно надо и что мы часто называем случайными числами, — это последовательность чисел, которые подчиняются определенному закону распределения и которые невозможно предсказать, зная предыдущие числа последовательности. Такие числа очень полезны при тестировании программ (они позволяют генерировать множество тестов), в играх (это один из способов гарантировать, что следующий шаг в игре не совпадет с предыдущим) и в моделировании (можно смоделировать сущность, которая ведет себя случайно в пределах изменения своих параметров).

В качестве практического инструмента и математической задачи случайные числа в настоящее время достигли высокой степени интеллектуальности, соответствующей их важности в реальном мире. Здесь мы коснемся лишь азов теории случайных чисел, необходимых для простого тестирования и моделирования. В заголовочном файле **<random>** стандартной библиотеки имеется набор сложных средств для генерации случайных чисел, соответствующих различным математическим распределениям. Эти средства стандартной библиотеки основаны на двух фундаментальных понятиях.

- *Механизмы*. Механизм представляет собой функциональный объект, который генерирует равномерно распределенную последовательность целочисленных значений.
- *Распределения*. Распределение представляет собой функциональный объект, который в соответствии с математической формулой генерирует последовательность значений для данной входной последовательности значений, генерируемой механизмом.

Например, рассмотрим функцию **random_vector()**, упомянутую в разделе 24.6.3. Вызов функции **random_vector(n)** порождает матрицу **Matrix <double,1>**, содержащую **n** элементов типа **double** со случайными значениями в диапазоне [0,n):

```
Vector random_vector(Index n)
{
    Vector v(n);

    default_random_engine ran{};  // Генерирует целые числа
    uniform_real_distribution<>
```

```
        ureal{0,max};              // Отображает int в double
                                   // в диапазоне [0,max)

    for (Index i = 0; i < n; ++i)
        v(i) = ureal(ran);
    return v;
}
```

Механизм по умолчанию (**default_random_engine**) прост, малозатратен при работе и достаточно хорош для несистематического применения. Для более профессионального применения стандартная библиотека предлагает различные механизмы с лучшими свойствами случайности и различной стоимостью выполнения. Примерами являются **linear_congurential_engine**, **mersenne_twister_engine** и **random_device**. Если вы хотите использовать механизмы стандартной библиотеки и вам требуется что-то получше **default_random_engine**, вам придется обратиться к документации. Чтобы получить представление о качестве генератора случайных чисел в вашей системе, выполните упр. 10.

Два генератора случайных чисел из **std_lib_facilities.h** определены следующим образом:

```
int randint(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>{min,max}(ran);
}

int randint(int max)
{
    return randint(0,max);
}
```

Эти простые функции наиболее полезны, но просто чтобы испытать что-то еще, давайте сгенерируем случайные числа, подчиняющиеся нормальному распределению:

```
auto gen = bind(normal_distribution<double>{15,4.0},
                default_random_engine{} );
```

Функция стандартной библиотеки **bind()** из заголовочного файла **<functional>** создает функциональный объект, который при использовании вызывает первый аргумент со вторым аргументом в качестве своего аргумента. Так что в данном случае вызовы **gen()** приводят к возврату значений с нормальным распределением, со средним значением **15** и дисперсией **4.0** с помощью механизма **default_random_engine**. Мы можем использовать этот функциональный объект следующим образом.

```
vector<int> hist(2*15);

for (int i = 0; i < 500; ++i)          // Генерация гистограммы
```

```
    ++hist[int(round(gen()))];          // из 500 значений
for (int i = 0; i != hist.size(); ++i){ // Вывод гистограммы
    cout << i << '\t';
    for (int j = 0; j != hist[i]; ++j)
        cout << '*';
    cout << '\n';
}
```

В результате мы получим следующую гистограмму:

```
 0
 1
 2
 3 **
 4 *
 5 *****
 6 ****
 7 ****
 8 ******
 9 ************
10 ****************************
11 ****************************
12 ************************************
13 ************************************************************
14 ****************************************************
15 ************************************************************
16 *********************************
17 ****************************************************
18 ************************************
19 ***********************************
20 ***************
21 ************
22 ************
23 *******
24 ******
25 *
26 *
27
28
29
```

Нормальное распределение очень распространено и известно также как распределение Гаусса или (по очевидной причине) "колоколообразная кривая". Прочие распределения включают `bernoulli_distribution`, `exponential_distribution` и `chi_squared_distribution`. Их описание можно найти в книге *The C++ Programming Language*. Целочисленные распределения возвращают значения в замкнутых интервалах [a,b], в то время как действительные распределения (распределения с плавающей точкой) возвращают значения в открытых интервалах [a,b).

По умолчанию механизм (за исключением, возможно, `random_device`) дает одну и ту же последовательность при каждом выполнении програм-

мы. Это наиболее удобно для первоначальной отладки. Если же требуется, чтобы механизм выдавал разные последовательности, его надо инициализировать разными значениями, которые обычно называют "семенами" (seed), например:

```
auto gen1 = bind(uniform_int_distribution<>{0,9}, default_random_engine{} );
auto gen2 = bind(uniform_int_distribution<>{0,9}, default_random_engine{10});
auto gen3 = bind(uniform_int_distribution<>{0,9}, default_random_engine{5});
```

Для получения непредсказуемых значений часто используется текущее время (до наносекунд; см. раздел 26.6.1) или что-то подобное.

24.8. Стандартные математические функции

В стандартной библиотеке содержатся не менее стандартные математические функции (`cos`, `sin`, `log` и т.д.). Их объявления можно найти в заголовочном файле `<cmath>`.

Стандартные математические функции	
`abs(x)`	Абсолютное значение
`ceil(x)`	Наименьшее целое число, удовлетворяющее условию `>= x`
`floor(x)`	Наибольшее целое число, удовлетворяющее условию `<= x`
`sqrt(x)`	Квадратный корень; значение `x` должно быть неотрицательным
`cos(x)`	Косинус
`sin(x)`	Синус
`tan(x)`	Тангенс
`acos(x)`	Арккосинус; результат неотрицательный
`asin(x)`	Арксинус; возвращается результат, ближайший к нулю
`atan(x)`	Арктангенс
`sinh(x)`	Гиперболический синус
`cosh(x)`	Гиперболический косинус
`tanh(x)`	Гиперболический тангенс
`exp(x)`	Экспонента (основание натуральных логарифмов *e* в степени `x`)
`log(x)`	Натуральный логарифм (с основанием *e*); значение `x` должно быть положительным
`log10(x)`	Десятичный логарифм

Стандартные математические функции могут иметь аргументы типов `float`, `double`, `long double` и `complex` (раздел 24.9). Эти функции очень полезны при вычислениях с плавающей точкой. Более подробная информация содержится в широко доступной документации, а для начала можно обратиться к справочной системе вашего компилятора.

Если стандартная математическая функция не может дать математически корректный результат, она устанавливает флажок **errno**. Рассмотрим пример.

```
errno = 0;
double s2 = sqrt(-1);
if (errno)cerr << "Что-то где-то пошло не так, как надо";
if (errno == EDOM) // Ошибка из-за выхода аргумента
                   // за пределы области определения
    cerr << "Фунция sqrt() для отрицательных "
         << "аргументов не определена";
pow(very_large,2); // Плохая идея
if (errno==ERANGE) // Ошибка из-за выхода за пределы допустимого диапазона
    cerr << "pow(" << very_large
         << ",2) слишком большое число для double";
```

Выполняя серьезные математические вычисления, вы всегда должны проверять значение **errno**, чтобы убедиться, что после возврата результата оно по-прежнему нулевое. В противном случае что-то пошло не так, как надо. Для того чтобы узнать, какие математические функции могут устанавливать флажок **errno** и чему он может быть равен, обратитесь к документации.

Как показано в примере, ненулевое значение флажка **errno** просто означает, что что-то пошло не так. Функции, не входящие в стандартную библиотеку, также довольно часто устанавливают флажок **errno** при выявлении ошибок, поэтому следует точнее анализировать разные значения переменной **errno**, чтобы понять, что именно случилось. Если вы проверяете значение **errno** сразу после выполнения функции стандартной библиотеки и установили ее равной нулю перед вызовом, то вы можете полагаться на значение этой переменной, как мы сделали это в приведенном выше примере для значений **EDOM** и **ERANGE**. Константа **EDOM** означает ошибку, возникшую из-за выхода аргумента за пределы области определения функции (domain error), т.е. проблему с аргументами. Константа **ERANGE** означает выход за пределы допустимого диапазона значений (range error), т.е. проблему с возвращаемым значением.

Обработка ошибок с помощью переменной **errno** имеет довольно примитивный характер. Она уходит корнями к первым математическим функциям языка программирования C (1975).

24.9. Комплексные числа

Комплексные числа широко используются в научных и инженерных вычислениях. Мы полагаем, что раз они вам необходимы, значит, вам известны их математические свойства, поэтому просто покажем, как комплексные числа выражаются в стандартной библиотеке языка ISO C++.

Объявления комплексных чисел и связанных с ними математических функций находятся в заголовочном файле **<complex>**.

```
template<class Scalar> class complex {
    // Комплексное число представляет собой пару скалярных
    // значений, по сути – пару координат
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i)
        :re(r), im(i) {}
    constexpr complex(const Scalar & r) :re(r),im(Scalar ()) {}
    complex() :re(Scalar ()), im(Scalar ()) { }
    constexpr Scalar real(){return re;} // Действительная часть
    constexpr Scalar imag(){return im;} // Мнимая часть
    // Операторы: = += -= *= /=
};
```

Стандартная библиотека **complex** поддерживает типы скалярных величин **float**, **double** и **long double**. Кроме членов класса **complex** и стандартных математических функций (раздел 24.8), заголовочный файл **<complex>** предлагает ряд полезных операций.

Операции с комплексными числами	
z1+z2	Сложение
z1-z2	Вычитание
z1*z2	Умножение
z1/z2	Деление
z1==z2	Проверка равенства
z1!=z2	Проверка неравенства
norm(z)	Квадрат **abs(z)**
conj(z)	Сопряженное число: если **z** равно **{re,im}**, то **conj(z)** равно **(re,-im)**
polar(x,y)	Перевод в полярные координаты (**rho**, **theta**)
real(z)	Действительная часть
imag(z)	Мнимая часть
abs(z)	Модуль, **rho**
arg(z)	Аргумент, **theta**
out << z	Вывод комплексного числа
in >> z	Ввод комплексного числа

Примечание: в классе **complex** нет операций **<** и **%**.

Класс **complex<T>** используется так же, как любой другой встроенный тип, например **double**:

```
using cmplx = complex<double>; // complex<double> слишком длинно

void f(cmplx z, vector<cmplx>& vc)
```

```
{
    cmplx z2 = pow(z,2);
    cmplx z3 = z2*9.3+vc[3];
    cmplx sum = accumulate(vc.begin(), vc.end(), cmplx{} );
    // ...
}
```

Помните, что не все операции над числами типов **int** и **double** определены для класса **complex**!

```
if (z2<z3) // Ошибка: операции < для комплексных чисел нет
```

Обратите внимание на то, что представление (размещение в памяти) комплексных чисел в стандартной библиотеке языка C++ совместимо с соответствующими типами в языках C и Fortran.

24.10. Ссылки

По сути, вопросы, поднятые в этой главе, такие как ошибки округления, операции над матрицами и арифметика комплексных чисел, сами по себе интереса не представляют и их изложение в отрыве от конкретных приложений имеет мало смысла. Мы просто описываем некоторые возможности, предоставляемые языком C++, для тех, кому необходимо выполнять математические вычисления и кто обладает соответствующими математическими знаниями.

Если вы подзабыли математику или просто интересуетесь ею, то можете обратиться к следующим источникам информации.

Архив MacTutor History of Mathematics, размещенный по адресу **http://www-gap.dcs.st-and.ac.uk/~history**.

- Отличная веб-страница для всех, кто любит математику или просто хочет ее применять.
- Отличная веб-страница для всех, кто хочет увидеть гуманитарный аспект математики; например, кто из крупных математиков выиграл Олимпийские игры?
  - Знаменитые математики: биографии, достижения.
  - Курьезы.
- Знаменитые кривые.
- Известные задачи.
- Математические темы.
  - Алгебра.
  - Анализ.
  - Теория чисел.

- Геометрия и топология.
- Математическая физика.
- Математическая астрономия.
- История математики.
- Многое другое

1. Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.
2. Gullberg, Jan. *Mathematics — From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X. Одна из наиболее интересных книг об основах и пользе математики, которую можно читать одновременно и с пользой (например, о матрицах), и с удовольствием.
3. Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN: 0202496842.[1]
4. Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.
5. Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020194291X.

Задание

1. Выведите на экран размеры типов **char**, **short**, **int**, **long**, **float**, **double**, **int*** и **double*** (используйте оператор **sizeof**, а не заголовочный файл **<limits>**).
2. Используя оператор **sizeof**, выведите на экран размеры объектов **Matrix <int> a(10)**, **Matrix<int> b(10)**, **Matrix<double> c(10)**, **Matrix <int,2> d(10,10)**, **Matrix<int,3> e(10,10,10)**.
3. Выведите на печать количество элементов в каждом из объектов, перечисленных в упр. 2.
4. Напишите программу, вводящую числа типа **int** из потока **cin** и выводящую результат применения функции **sqrt()** к каждому из этих чисел **int** или сообщение "квадратный корень не существует", если функцию **sqrt(x)** нельзя применять к некоторым значениям **x** (т.е. проверяйте значения, возвращаемые функцией **sqrt()**).
5. Считайте десять чисел с плавающей точкой из потока ввода и запишите их в матрицу **Matrix<double>**. Класс **Matrix** не имеет функции

[1] Имеется русский перевод: Дональд Э. Кнут. *Искусство программирования, том 2.* Получисленные алгоритмы, 3-е изд. — М.: Издательский дом "Вильямс", 2000.

`push_back()`, поэтому будьте осторожны и предусмотрите реакцию на попытку ввести неверное количество чисел типа `double`. Выведите эту матрицу на экран.

6. Вычислите таблицу умножения [0,n)*[0,m) и представьте ее в виде двумерной матрицы `Matrix`. Вводите числа `n` и `m` из потока `cin` и аккуратно выводите на экран полученную таблицу (предполагается, что число `m` достаточно мало, чтобы результаты поместились в одной строке).
7. Введите из потока `cin` десять объектов класса `complex<double>` (класс `cin` поддерживает оператор `>>` для типа `complex`) и поместите их в матрицу. Вычислите и выведите на экран сумму десяти комплексных чисел.
8. Считайте шесть чисел типа `int` в матрицу `Matrix<int,2> m(2,3)` и выведите их на экран.

Контрольные вопросы

1. Кто выполняет числовые расчеты?
2. Что такое точность?
3. Что такое переполнение?
4. Каковы обычные размеры типов `double` и `int`?
5. Как можно обнаружить переполнение?
6. Как определить пределы изменения чисел, например наибольшее число типа `int`?
7. Что такое массив? строка? столбец?
8. Что такое многомерный массив в стиле языка C?
9. Какими свойствами должна обладать языковая поддержка (например, библиотека) матричных вычислений?
10. Что такое размерность матрицы?
11. Сколько размерностей теоретически может иметь матрица?
12. Что такое срезка?
13. Что такое функция трансляции? Приведите пример.
14. В чем заключается разница между индексированием в стиле языков Fortran и C?
15. Как применить операцию к каждому элементу матрицы? Приведите примеры.
16. Что такое слитная операция (fused operation)?
17. Дайте определение *скалярного произведения*.
18. Что такое линейная алгебра?

19. Опишите метод исключения Гаусса.
20. Что такое опорный элемент в линейной алгебре? А в реальной жизни?
21. Что делает число случайным?
22. Что такое равномерное распределение?
23. Где найти стандартные математические функции? Для каких типов аргументов они определены?
24. Что такое мнимая часть комплексного числа?
25. Чему равен корень квадратный из –1?

Термины

`errno`	масштабирование	размерность
Fortran	матрица	система линейных уравнений
`Matrix`	метод Гаусса	скалярное произведение
`sizeof`	мнимая часть	слитная операция
действительная часть	многомерность	случайное число
индексирование	поэлементная операция	срезка
комплексное число	равномерное распределение	столбец
массив	размер	строка

Упражнения

1. Аргументы функции `f` в выражениях `a.apply(f)` и `apply(f,a)` являются разными. Напишите функцию `triple()` для каждого варианта и примените их для утроения элементов массива `{ 1 2 3 4 5 }`. Определите единую функцию `triple()`, которую можно было бы использовать как в выражении `a.apply(triple)`, так и в выражении `apply(triple,a)`. Объясните, почему писать для использования в качестве аргумента функции `apply()` все функции именно таким образом — плохая идея.
2. Повторите упр. 1, используя не функции, а функциональные объекты. *Указание*: примеры можно найти в заголовочном файле `Matrix.h`.
3. Только для экспертов (средствами, описанными в книге, эту задачу решить невозможно). Напишите функцию `apply(f,a)`, принимающую в качестве аргумента функции `void(T&)`, `T(const T&)`, а также эквивалентные им функциональные объекты. *Указание*: см. `boost::bind`.
4. Выполните программу метода исключения Гаусса, т.е. завершите ее, скомпилируйте и протестируйте на простом примере.
5. Примените программу метода исключения Гаусса к системе `A=={{0 1} {1 0}}` и `b=={5 6}` и убедитесь, что программа не в состоянии найти решение. Затем попробуйте применить в ней функцию `elim_with_partial_pivot()`.

6. Замените циклами векторные операции `dot_product()` и `scale_and_add()` в программе метода исключения Гаусса. Протестируйте и прокомментируйте эту программу.
7. Перепишите программу метода исключения Гаусса без помощи библиотеки `Matrix`. Иначе говоря, используйте встроенные массивы или класс `vector`, но не класс `Matrix`.
8. Анимируйте метод исключения Гаусса.
9. Перепишите функцию `apply()`, не являющуюся членом класса `Matrix`, так, чтобы она возвращала объект класса `Matrix`, элементы которого имели бы тип возврата примененной функции. Иначе говоря, функция `apply(f,a)` должна возвращать объект класса `Matrix<R>`, где `R` — тип возвращаемого функцией `f` значения. *Предупреждение*: это решение требует информации о шаблонах, которая в данной книге отсутствует.
10. Насколько случайным является механизм `default_random_engine`? Напишите программу, принимающую два целых числа `n` и `d` из потока ввода, `d` раз вызывающую функцию `randint(n)` и записывающую результат. Выведите на экран количество каждого из выпавших чисел из диапазона `[0,n)` и оцените, насколько одинаковым является их количество. Выполните программу с небольшими значениями `n` и небольшими значениями `d`, чтобы убедиться в том, что очевидные перекосы возникают только при небольшом количестве испытаний.
11. Напишите функцию `swap_columns()`, аналогичную функции `swap_rows()` из раздела 24.5.3. Очевидно, что для этого необходимо изучить код библиотеки `Matrix`. Не беспокойтесь об эффективности своей программы: быстродействие функции `swap_columns()` в принципе не может превышать быстродействие функции `swap_rows()`.
12. Реализуйте операторы

```
Matrix<double> operator*(Matrix<double,2>&, Matrix<double>&);
```

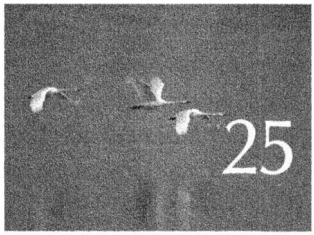
и

```
Matrix<double,N> operator+(Matrix<double,N>&, Matrix<double,N>&)
```

При необходимости посмотрите их математические определения в учебниках.

Послесловие

Если вы не любите математику, то, возможно, вам не понравилась эта глава и вы выберете для себя область приложений, в которой изложенная выше информация не понадобится. С другой стороны, если вы любите математику, то мы надеемся, что вы оцените точность выражения математических концепций в представленном в этой главе коде.



25

Программирование встроенных систем

Слово "опасный" означает, что кто-то может умереть.

— Сотрудник службы безопасности

В этой главе мы рассмотрим вопросы программирования встроенных систем; иначе говоря, обсудим темы, связанные, в первую очередь, с написанием программ для устройств, которые не являются традиционными компьютерами с экранами и клавиатурами. Основное внимание уделяется принципам и методам программирования таких устройств, языковым возможностям и стандартам кодирования, необходимым для непосредственной работы с аппаратным обеспечением. К этим темам относятся управление ресурсами и памятью, использование указателей и массивов, а также манипулирование битами. Главный акцент делается на безопасном использовании, а также на альтернативах использованию низкоуровневых средств. Мы не пытаемся описывать специализированные архитектуры устройств или способы прямого доступа к аппаратному обеспечению, для этого существует специализированная литература. В качестве иллюстрации мы выбрали реализацию алгоритма шифрования/расшифровки.

25.1. Встроенные системы

Большая часть существующих компьютеров не распознается как компьютеры. Они просто являются частью более крупной системы или устройства. Рассмотрим примеры.

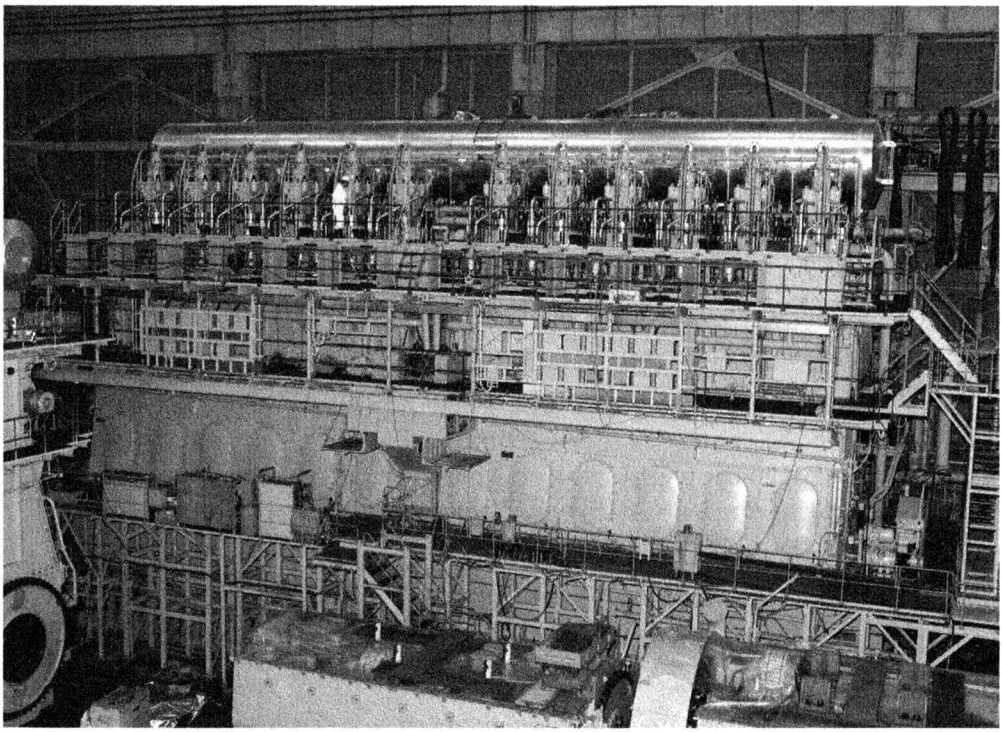
- *Автомобили.* В современный автомобиль могут быть встроены десятки компьютеров, управляющих впрыском топлива, следящих за работой двигателя, настраивающих радио, контролирующих тормоза, наблюдающих за давлением в шинах, управляющих дворниками на ветровом стекле и т.д.
- *Телефоны.* Мобильный телефон содержит как минимум два компьютера; один из них обычно специализируется на обработке сигналов.
- *Самолеты.* Современный самолет оснащен компьютерами, управляющими буквально всем: от системы развлечения пассажиров до закрылков, оптимизирующих подъемную силу крыла.
- *Фотоаппараты.* Существуют фотоаппараты с пятью процессорами, в которых даже каждая линза имеет собственный процессор.
- *Кредитные карточки* (и все семейство карточек с микропроцессорами).
- *Мониторы и контроллеры медицинского оборудования* (например, сканеры для компьютерной томографии).

- *Грузоподъемники* (лифты).
- *Карманные компьютеры.*
- *Контроллеры принтеров.*
- *Звуковые системы.*
- *MP3-плееры.*
- *Кухонное оборудование* (например, скороварки и хлебопечки).
- *Телефонные коммутаторы* (как правило, состоящие из тысяч специализированных компьютеров).
- *Контроллеры насосов* (например, водяных или нефтяных).
- *Сварочные роботы*, которые используются в труднодоступных или опасных местах, где человек работать не может.
- *Ветряки.* Некоторые из них способны вырабатывать мегаватты электроэнергии и имеют высоту до 200 метров.
- *Контроллеры шлюзов на дамбах.*
- *Мониторы качества на конвейерах.*
- *Устройства считывания штриховых кодов.*
- *Автосборочные роботы.*
- *Контроллеры центрифуг* (используемых во многих процессах медицинского анализа).
- *Контроллеры дисководов.*

Эти компьютеры являются частью более крупных систем, которые обычно не похожи на компьютеры и о которых мы никогда не думаем как о компьютерах. Когда мы видим автомобиль, проезжающий по улице, мы не говорим: "Смотрите, поехала распределенная компьютерная система!" И хотя автомобиль *в том числе* является и распределенной компьютерной системой, ее действия настолько тесно связаны с работой механической, электронной и электрической систем, что мы не можем считать ее изолированным компьютером. Ограничения, наложенные на работу этой системы (временные и пространственные), и понятие корректности ее программ не

могут быть отделены от содержащей ее более крупной системы. Часто встроенный компьютер управляет физическим устройством, и корректное поведение компьютера определяется как корректное поведение самого физического устройства. Рассмотрим крупный дизельный судовой двигатель.

Обратите внимание на крышку пятого цилиндра, на котором стоит человек. Это большой двигатель, приводящий в движение большой корабль. Если такой двигатель выйдет из строя, мы узнаем об этом на первых страницах утренних газет. В крышке каждого цилиндра такого двигателя находится управляющая система цилиндра, состоящая из трех компьютеров.



Каждая система управления цилиндром соединена с системой управления двигателем (еще три компьютера) посредством двух независимых сетей. Кроме того, система управления двигателем связана с центром управления, в котором механики могут отдавать двигателю команды с помощью специализированной системы графического интерфейса. Всю эту систему можно контролировать дистанционно с помощью радиосигналов (через спутники) из центра управления морским движением. Другие примеры использования компьютеров приведены в главе 1.

Так что же, с точки зрения программиста, особенного в программах, выполняемых такими компьютерами? Обобщим вопрос: какие проблемы, не беспокоящие нас в "обычных" программах, выходят на первый план в разнообразных встроенных системах?

- Часто *критически важной является надежность*. Отказ может привести к тяжелым последствиям: большим убыткам (миллионы долларов) и, возможно, чьей-то смерти (людей на борту корабля, терпящего бедствие, или животных, погибших вследствие разлива топлива в морских водах).
- Часто *ресурсы (память, циклы процессора, мощность) ограничены*. Для компьютера, управляющего двигателем, вероятно, это не проблема, но для мобильных телефонов, сенсоров, карманных компьютеров, компьютеров на космических зондах и так далее это важно.

В мире, где двухпроцессорные портативные компьютеры с частотой 2 ГГц и объемом ОЗУ 8 Гбайт уже не редкость, главную роль в работе самолета или космического зонда могут играть компьютеры с частотой процессора 60 МГц и объемом памяти 256 Кбайт, а в небольших устройствах — с частотой ниже 1 МГц и объемом оперативной памяти, измеряемой несколькими сотнями слов. Компьютеры, устойчивые к внешним воздействиям (вибрации, ударам, нестабильной поставке электричества, жаре, холоду, влаге, топтанию на нем и т.д.), обычно работают намного медленнее, чем студенческие ноутбуки.

- Часто *важна реакция в реальном времени*. Если инжектор топлива не попадет в инъекционный цикл, то с очень сложной системой мощностью 100 тысяч лошадиных сил может случиться беда; если инжектор пропустит несколько циклов, т.е. будет неисправен около секунды, то с пропеллером 10 метров в диаметре и весом 130 тонн могут произойти странные вещи. Мы бы очень не хотели, чтобы это случилось.
- Часто *система должна бесперебойно работать много лет*. Эти системы могут быть дорогими, как, например, спутник связи, вращающийся на орбите, или настолько дешевыми, что их ремонт не имеет смысла (например, MP3-плееры, кредитные карточки или инжекторы автомобильных двигателей). В США критерием надежности телефонных коммутаторов считается 20 минут простоя за двадцать лет (даже не думайте о том, чтобы отключать его каждый раз, когда захотите изменить его программу).
- Часто *ремонт может быть невозможным или очень редким*. Вы можете приводить корабли в гавань для ремонта его компьютеров или других систем каждые два года и обеспечить, чтобы компьютерные специалисты были в нужном месте в нужное время. Однако выполнить незапланированный ремонт часто невозможно (если корабль попадет в шторм посреди Тихого океана, то ошибки в программе могут сыграть роковую роль). Вы просто не сможете отправить кого-то ремонтировать космический зонд, вращающийся на орбите вокруг Марса.

Некоторые системы подпадают под все перечисленные выше ограничения, а некоторые — только под одно. Это дело экспертов в конкретной прикладной области. Наша цель — вовсе не сделать из вас эксперта по всем вопросам, это было бы глупо и очень безответственно. Наша цель — ознакомить вас с основными проблемами и концепциями, связанными с их решением, чтобы вы оценили сложность навыков, которые вам потребуются при создании таких систем. Возможно, вы захотите приобрести более глубокие знания. Люди, разрабатывающие и реализующие встроенные системы, играют очень важную роль в развитии технической цивилизации. Это область, в которой профессионалы могут добиться многого.

Относится ли это к новичкам и к программистам на языке C++? Да, и еще раз да. Встроенных систем намного больше, чем обычных персональных компьютеров. Огромная часть программистской работы связана с программированием именно встроенных систем. Более того, список примеров встроенных систем, приведенный в начале раздела, составлен на основе моего личного опыта программирования на языке C++.

25.2. Основные концепции

Большая часть процесса программирования компьютеров, являющихся частями встроенных систем, ничем не отличается от обычного программирования, поэтому к ним можно применить большинство идей, сформулированных в книге. Однако акцент часто оказывается иным: мы должны адаптировать использование средств языка программирования так, чтобы учесть ограничения задачи, и зачастую нам приходится работать с аппаратным обеспечением на самом низком уровне.

- *Корректность.* Это понятие становится еще более важным, чем обычно. Корректность — это не просто абстрактное понятие. В контексте встроенной системы программа считается корректной не тогда, когда она просто выдает правильные результаты, а тогда, когда она делает это за указанное время, в заданном порядке и с использованием только имеющегося набора ресурсов. В принципе, детали понятия *корректность* тщательно формулируются в каждом конкретном случае, но часто такую спецификацию можно создать только после ряда экспериментов. Часто важные эксперименты можно провести только тогда, когда вся система (вместе с компьютером, на котором будет выполняться программа) уже построена. Исчерпывающая формулировка понятия корректности встроенной системы может быть одновременно чрезвычайно трудной и крайне важной. Слова "чрезвычайно трудная" могут означать "невозможная за имеющееся время и при заданных ресурсах"; мы должны попытаться сделать все возможное с помощью имеющихся средств и методов. К счастью, количество спецификаций, методов моделирования и тестирования и других технологий в заданной области весьма впечатляющее. Слова "крайне важная" могут означать "сбой приводит к повреждению или разрушению".

- *Устойчивость к сбоям.* Необходимо тщательно указать набор условий, которым должна удовлетворять программа. Например, при сдаче обычной студенческой программы вы можете считать совершенно нечестным, если преподаватель во время ее демонстрации выдернет провод питания из розетки. Исчезновение электропитания не входит в список условий, на которые должны реагировать обычные при-

кладные программы на персональных компьютерах. Однако потеря электропитания во встроенных системах может быть обычным делом и ваша программа должна это учитывать. Например, жизненно важные части системы могут иметь двойное электропитание, резервные батареи и т.д. Что еще хуже, в некоторых приложениях фраза "Я предполагал, что аппаратное обеспечение будет работать без сбоев" не считается оправданием. Долгое время и в часто изменяющихся условиях аппаратное обеспечение просто не способно работать без сбоев. Например, программы для некоторых телефонных коммутаторов и аэрокосмических аппаратов написаны в предположении, что рано или поздно часть памяти компьютера просто решит изменить свое содержание (например, заменит нуль единицей). Кроме того, компьютер может решить, что ему нравится единица, и игнорировать попытки изменить ее на нуль. Если у вас много памяти и вы используете ее достаточно долгое время, то в конце концов такие ошибки возникнут. Если память компьютера подвергается радиационному облучению за пределами земной атмосферы, то это произойдет намного раньше. Работая с системой (встроенной или нет), мы должны решить, как реагировать на сбои оборудования. Обычно по умолчанию считают, что аппаратное обеспечение будет работать без сбоев. Если же мы имеем дело с более требовательными системами, то это предположение следует уточнить.

- *Отсутствие простоев.* Встроенные системы обычно должны долго работать без замены программного обеспечения или вмешательства опытного оператора. "Долгое время" может означать дни, месяцы, годы или все время функционирования аппаратного обеспечения. Это обстоятельство вполне характерно для встроенных систем, но не применимо к огромному количеству "обычных приложений", а также ко всем примерам и упражнениям, приведенным в книге. Требование "должно работать вечно" выдвигает на первый план обработку ошибок и управление ресурсами. Что такое "ресурс"? Ресурс — это нечто такое, что имеется у машины в ограниченном количестве; программа может получить ресурс путем выполнения явного действия (выделить память) и вернуть его системе (освободить память) явно или неявно. Примерами ресурсов являются память, дескрипторы файлов, сетевые соединения (сокеты) и блокировки. Программа, являющаяся частью долговременной системы, должна освобождать свои ресурсы, за исключением тех, которые необходимы ей постоянно. Например, программа, забывающая закрывать файл раз в день, в большинстве операционных систем не выживет более месяца. Программа, не освобождающая каждый день по 100 байтов, за год исчерпает 32 Кбайт — этого

достаточно, чтобы через несколько месяцев небольшое устройство перестало работать. Самое ужасное в такой "утечке" ресурсов заключается в том, что многие месяцы такая программа работает идеально, а потом неожиданно дает сбой. Если уж программа обречена потерпеть крах, то хотелось бы, чтобы это произошло пораньше и у нас было время устранить проблему. В частности, было бы лучше, если бы сбой произошел до того, как программа попадет к пользователям.

- *Ограничения реального времени.* Встроенную систему можно отнести к системам с *жесткими условиями реального времени* (hard real time), если она должна всегда давать ответ до наступления заданного срока. Если она должна давать ответ до наступления заданного срока лишь в большинстве случаев, но иногда может позволить себе просрочить время, такую систему можно отнести к системам с *мягкими условиями реального времени* (soft real time). Примерами систем с мягкими условиями реального времени являются контроллеры автомобильных окон и усилитель стереосистемы. Обычный человек все равно не заметит миллисекундной задержки в движении стекол, и только опытный слушатель способен уловить миллисекундное изменение высоты звука. Примером системы с жесткими условиями реального времени является инжектор топлива, который должен впрыскивать бензин в точно заданные моменты времени с учетом движения поршня. Если произойдет хотя бы миллисекундная задержка, мощность двигателя упадет и он станет портиться; в итоге двигатель может выйти из строя, что, возможно, повлечет за собой дорожное происшествие или катастрофу.

- *Предсказуемость.* Это ключевое понятие во встроенных системах. Очевидно, что этот термин имеет много интуитивных толкований, но здесь — в контексте программирования встроенных систем — мы используем лишь техническое значение: операция считается *предсказуемой* (predictable), если на данном компьютере она всегда выполняется за одно и то же время и если все такие операции выполняются за одно и то же время. Например, если **x** и **y** — целочисленные переменные, то инструкция **x+y** всегда будет выполняться за фиксированное время, а инструкция **xx+yy** будет выполняться за точно такое же время, при условии, что **xx** и **yy** — две другие целочисленные переменные. Как правило, можно пренебречь небольшими колебаниями скорости выполнения операции, связанными с машинной архитектурой (например, отклонениями, вызванными особенностями кеширования и конвейерной обработки), и просто ориентироваться на верхний предел заданного времени. Непредсказуемые операции (в данном смысле этого слова) нельзя использовать

в системах с жесткими условиями реального времени и можно лишь с очень большой осторожностью применять в остальных системах реального времени. Классическим примером непредсказуемой операции является линейный поиск по списку (например, выполнение функции `find()`), если количество элементов списка неизвестно и не может быть легко оценено сверху. Такой поиск можно применять в системах с жесткими условиями реального времени, только если мы можем надежно предсказать количество или хотя бы максимальное количество элементов списка. Иначе говоря, для того чтобы *гарантировать*, что ответ поступит в течение фиксированного интервала времени, мы должны — возможно, с помощью инструментов анализа кода — вычислить время, необходимое для выполнения любой последовательности команд.

- *Параллелизм.* Встроенные системы обычно реагируют на события, происходящие во внешнем мире. Это значит, что в программе многие события могут происходить одновременно, поскольку они соответствуют одновременно происходящим событиям в реальном мире. Программа, одновременно выполняющая несколько действий, называется *параллельной* (concurrent, parallel). К сожалению, эта очень интересная, сложная и важная тема выходит за рамки рассмотрения нашей книги.

25.2.1. Предсказуемость

С точки зрения предсказуемости язык C++ достаточно хорош, но не идеален. Практически все средства языка C++ (включая вызовы виртуальных функций) вполне предсказуемы, за исключением указанных ниже.

- Работа с динамической памятью с помощью операторов **`new`** и **`delete`** (см. раздел 25.3).
- Исключения (раздел 19.5).
- Оператор **`dynamic_cast`** (раздел А.5.7).

В приложениях с жесткими условиями реального времени эти средства использовать не следует. Проблемы, связанные с операторами **`new`** и **`delete`**, подробно описаны в разделе 25.3; они носят принципиальный характер. Обратите внимание на то, что класс **`string`** из стандартной библиотеки и стандартные контейнеры (**`vector`**, **`map`** и др.) неявно используют динамическую память, поэтому они также непредсказуемы. Проблема с оператором **`dynamic_cast`** связана с трудностями его параллельной реализации, но не является фундаментальной.

Проблемы с исключениями заключаются в том, что, глядя на конкретный раздел **`throw`**, программист не может сказать, сколько времени займет

поиск соответствующего раздела `catch` и даже существует ли такой раздел `catch`, не проанализировав более крупный фрагмент программы. В программах для встроенных систем было бы лучше, если бы такой раздел `catch` существовал, поскольку мы не можем рассчитывать на то, что программист сможет использовать средства отладки языка C++. В принципе, проблемы, связанные с исключениями, можно решить с помощью того же механизма, который определяет, какой именно раздел `catch` будет вызван для конкретного раздела `throw` и как долго ему будет передаваться управление, но в настоящее время эта задача еще исследуется, поэтому, если вам нужна предсказуемость, вы должны обрабатывать ошибки, основываясь на возвращаемых кодах и других устаревших и утомительных, но вполне предсказуемых методах.

25.2.2. Идеалы

При создании программ для встроенных систем существует опасность, что в погоне за высокой производительностью и надежностью программист станет использовать исключительно низкоуровневые средства языка. Эта стратегия вполне оправдана при разработке небольших фрагментов кода. Однако она легко превратит весь проект в непролазное болото, затруднит проверку корректности кода и повысит затраты времени и денег, необходимых для создания системы.

Как всегда, наша цель — работать на как можно более высоком уровне с учетом поставленных ограничений, связанных с нашей задачей. Не позволяйте себе опускаться до хваленого ассемблерного кода! Всегда стремитесь как можно более прямо выражать свои идеи в программе (при заданных ограничениях). Всегда старайтесь писать ясный, понятный и простой в сопровождении код. Не оптимизируйте его, пока вас к этому не вынуждают. Эффективность (по времени или по объему памяти) часто имеет большое значение для встроенных систем, но не следует пытаться выжимать максимум возможного из каждого маленького фрагмента кода. Кроме того, во многих встроенных системах в первую очередь требуется, чтобы программа работала правильно и достаточно быстро; часто "достаточно быстро" означает, что система просто простаивает, ожидая следующего действия. Постоянные попытки написать несколько строчек кода как можно более эффективно отнимают много времени, порождают много ошибок и часто затрудняют оптимизацию программ, поскольку алгоритмы и структуры данных становится трудно понимать и модифицировать. Например, низкоуровневая оптимизация часто приводит к невозможности оптимизировать размер программы, поскольку во многих местах возникает почти одинаковый код, который остальные части программы не могут использовать совместно из-за второстепенных различий.

Джон Бентли (John Bentley), известный своими очень эффективными программами, сформулировал два закона оптимизации.

- Первый закон: "Не делай этого!"
- Второй закон (только для экспертов): "Пока что не делай этого!"

Прежде чем приступить к оптимизации, следует убедиться в том, что вы понимаете, как работает система. Только когда вы будете уверены в этом, оптимизация станет (или может стать) правильной и надежной. Сосредоточьтесь на алгоритмах и структурах данных. Как только будет запущена первая версия системы, тщательно измерьте ее показатели и настройте ее как следует. К счастью, часто происходят приятные неожиданности: хороший код иногда работает достаточно быстро и не затрачивает слишком много памяти. Тем не менее не рассчитывайте на это; измеряйте. Неприятные сюрпризы также случаются достаточно часто.

25.2.3. Сохранение работоспособности после сбоя

Представьте себе, что необходимо разработать и реализовать систему, которая не должна выходить из строя. Под словами "не выходить из строя" мы подразумеваем "месяц работать без вмешательства человека". Какие сбои мы должны предотвратить? Мы можем не беспокоиться о том, что солнце вдруг потухнет или на систему наступит слон. Однако в целом мы не можем предвидеть, что может пойти не так, как надо. Для конкретной системы мы можем и должны выдвигать предположения о наиболее вероятных ошибках. Перечислим типичные примеры.

- Сбой или исчезновение электропитания.
- Вибрация разъема.
- Попадание в систему тяжелого предмета, приводящее к разрушению процессора.
- Падение системы с высоты (от удара диск может быть поврежден).
- Радиоактивное облучение, вызывающее непредсказуемое изменение некоторых значений, записанных в ячейках памяти.

Труднее всего найти преходящие ошибки. *Преходящей ошибкой* (transient error) мы называем событие, которое случается иногда, а не каждый раз при выполнении программы. Например, процессор может работать неправильно, только если температура превысит 54°C. Такое событие кажется невозможным, однако оно однажды действительно произошло, когда систему случайно забыли в заводском цехе на полу, что, конечно, никогда не происходило при тестировании в лаборатории.

Ошибки, которые не возникают в лабораторных условиях, исправить труднее всего. Вы представить себе не можете, какие усилия были предприняты, чтобы инженеры из лаборатории реактивных двигателей могли

диагностировать сбои программного и аппаратного обеспечения на марсоходе (сигнал до которого идет двадцать минут) и поняв, в чем дело, обновить программное обеспечение, чтобы устранить проблему.

Знание предметной области, т.е. сведений о системе, ее окружении и применении, играют важную роль при разработке и реализации систем, устойчивых к ошибкам. Здесь мы коснемся лишь самых общих вопросов. Подчеркнем, что каждый из этих общих вопросов был предметом тысяч научных статей и десятилетних исследований.

- *Предотвращение утечки ресурсов.* Не допускайте утечек. Старайтесь точно знать, какие ресурсы использует ваша программа, и стремитесь их экономить (в идеале). Любая утечка в конце концов выведет вашу систему или подсистему из строя. Самыми важными ресурсами являются время и память. Как правило, программа использует и другие ресурсы, например блокировки, каналы связи и файлы.
- *Дублирование.* Если для функционирования системы крайне важно, чтобы какое-то устройство работало нормально (например, компьютер, устройство вывода, колесо), то перед проектировщиком возникает фундаментальная проблема выбора: не следует ли продублировать критически важный ресурс? Мы должны либо смириться со сбоем, если аппаратное обеспечение выйдет из строя, либо предусмотреть резервное устройство и предоставить его в распоряжение программного обеспечения. Например, контроллеры топливных инжекторов в судовых дизельных двигателях снабжены тремя резервными компьютерами, связанными продублированной сетью. Подчеркнем, что резерв не обязан быть идентичным оригиналу (например, космический зонд может иметь мощную основную антенну и слабую запасную). Отметим к тому же, что в обычных условиях резерв можно использовать и для повышения производительности системы.
- *Самопроверка.* Необходимо знать, когда программа (или аппаратное обеспечение) работает неправильно. В этом отношении могут оказаться очень полезными компоненты аппаратного обеспечения (например, запоминающие устройства), которые сами себя контролируют, исправляют незначительные ошибки и сообщают о серьезных неполадках. Программное обеспечение может проверять целостность структур данных, инварианты (см. раздел 9.4.3) и полагаться на внутренний "санитарный контроль" (операторы утверждений (assertions)). К сожалению, самопроверка сама по себе является ненадежной, поэтому следует опасаться, чтобы сообщение об ошибке само не вызвало ошибку. Полностью проверить средства проверки ошибок — это действительно трудная задача.
- *Быстрый способ выхода из неправильно работающей программы.* Составляйте системы из модулей. В основу обработки ошибок дол-

жен быть положен модульный принцип: каждый модуль должен иметь собственную задачу. Если модуль решит, что не может выполнить свое задание, он может сообщить об этом другому модулю. Обработка ошибок внутри модуля должна быть простой (это повышает вероятность того, что она будет правильной и эффективной), а обработкой серьезных ошибок должен заниматься другой модуль. Высоконадежные системы модульные и многоуровневые. Сообщения о серьезных ошибках, возникших на каждом уровне, передаются на следующий уровень, и в конце концов, возможно, человеку. Модуль, получивший сообщение о серьезной ошибке (которую не может исправить никакой другой модуль), может выполнить соответствующее действие, возможно, связанное с перезагрузкой ошибочного модуля или запуском менее сложного (но более надежного) резервного модуля. Выделить модуль в конкретной системе — задача проектирования, но в принципе модулем может быть класс, библиотека, программа или все программы в компьютере.

- *Мониторинг подсистем* в ситуациях, когда они не могут самостоятельно сообщить о проблеме. В многоуровневой системе за системами более низкого уровня следят системы более высоких уровней. Многие системы, сбой которых недопустим (например, судовые двигатели или контроллеры космической станции), имеют по три резервные копии критических подсистем. Такое утроение означает не просто наличие двух резервных копий, но и то, что решение о том, какая из подсистем вышла из строя, принимается голосованием "два против одного". Утроение особенно полезно, когда многоуровневая организация представляет собой слишком сложную задачу (например, когда самый высокий уровень системы или подсистемы никогда не должен выходить из строя).

Мы можем спроектировать систему так, как хотели, и реализовать ее так, как умели, и все равно она может работать неверно. Прежде чем передавать систему пользователям, ее следует систематически и тщательно тестировать (подробнее об этом речь пойдет в главе 26).

25.3. Управление памятью

Двумя основными ресурсами компьютера являются время (на выполнение инструкций) и память (для хранения данных и кода). В языке C++ есть три способа выделения памяти для хранения данных (см. разделы 17.4 и A.4.2).

- *Статическая память*. Выделяется редактором связей и существует, пока выполняется программа.

- *Стековая (автоматическая) память.* Выделяется при вызове функции и освобождается после возвращения управления из функции.
- *Динамическая память (куча).* Выделяется оператором **new** и освобождается для возможного повторного использования с помощью оператора **delete**

Рассмотрим каждую из них с точки зрения программирования встроенных систем. В частности, изучим вопросы управления памятью с точки зрения задач, в которых важную роль играет предсказуемость (см. раздел 25.2.1), например при программировании систем с жесткими условиями реального времени и систем с особыми требованиями к обеспечению безопасности.

Статическая память не порождает каких-то особых проблем при программировании встроенных систем: вся память тщательно распределяется еще до старта программы и задолго до развертывания системы.

Стековая память может вызывать проблемы, поскольку ее может оказаться недостаточно, но эту проблему устранить несложно. Разработчики системы должны сделать так, чтобы в ходе выполнения программы стек никогда не превышал допустимый предел. Как правило, это означает, что количество вложенных вызовов функций должно быть ограниченным; иначе говоря, мы должны иметь возможность показать, что цепочки вызовов (например, **f1** вызывает **f2** вызывает ... вызывает **fn**) никогда не станут слишком длинными. В некоторых системах это приводит к запрету на рекурсивные вызовы. В некоторых системах такие запреты в отношении некоторых рекурсивных функций являются вполне оправданными, но их нельзя считать фундаментальными. Например, я *знаю*, что вызов **factorial(10)** вызовет функцию **factorial** не более десяти раз. Однако программист, разрабатывающий встроенную систему, может предпочесть итеративный вариант функции **factorial** (см. раздел 15.5), чтобы избежать сомнений или случайностей.

Динамическое распределение памяти обычно запрещено или строго ограничено; иначе говоря, либо оператор **new** запрещен, либо его использование ограничено периодом начальной инициализации программы, а оператор **delete** запрещен. Укажем основные причины этих ограничений.

- *Предсказуемость.* Размещение данных в динамической памяти непредсказуемо; иначе говоря, нет гарантии, что эта операция будет выполняться за постоянное время. Как правило, это не так: во многих реализациях оператора **new** время, необходимое для размещения нового объекта, может резко возрастать после размещения и удаления многих объектов.

- *Фрагментация.* Динамическая память может быть фрагментированной; другими словами, после размещения и удаления объектов оставшаяся память может содержать большое количество "дыр",

представляющих собой неиспользуемую память, которая бесполезна, потому что каждая "дыра" слишком мала для того, чтобы в ней поместился хотя бы один объект, используемый в приложении. Таким образом, размер полезной динамической памяти может оказаться намного меньше разности между первоначальным размером памяти и размером размещенных объектов.

В следующем разделе мы продемонстрируем, как может возникнуть такая неприемлемая ситуация. Отсюда следует, что мы должны избегать методов программирования, использующих операторы **new** и **delete** в системах с жесткими условиями реального времени или в системах с особыми требованиями к обеспечению безопасности. В следующем разделе мы покажем, как избежать проблем, связанных с динамической памятью, используя стеки и пулы.

25.3.1. Проблемы с динамической памятью

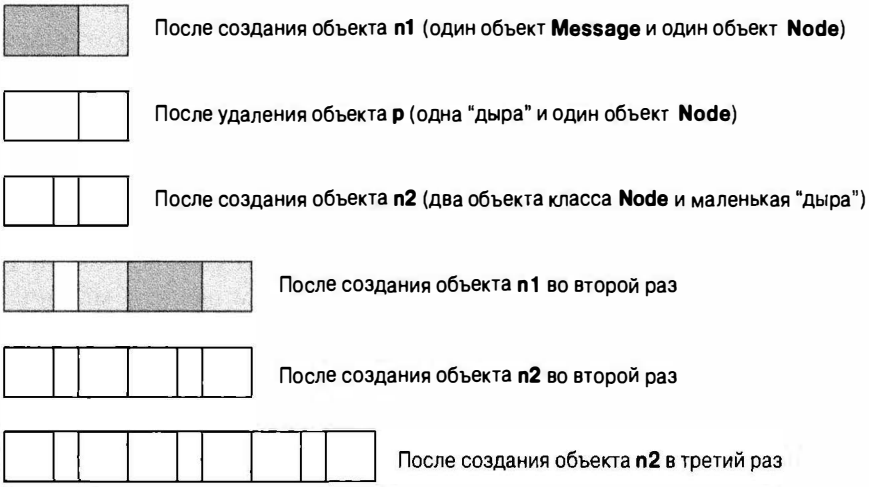
В чем заключается проблема, связанная с оператором **new**? На самом деле эта проблема порождается операторами **new** и **delete**, используемыми вместе. Рассмотрим результат следующей последовательности размещений и удалений объектов.

```
Message* get_input(Device&); // Создаем объект Message в динамической памяти

while(/* ... */) {
     Message* p = get_input(dev);
     // ...
     Node* n1 = new Node(arg1,arg2);
     // ...
     delete p;
     Node* n2 = new Node (arg3,arg4);
     // ...
}
```

Каждый раз, выполняя этот цикл, мы создаем два объекта класса **Node**, причем в процессе их создания возникает и удаляется объект класса **Message**. Такой фрагмент кода вполне типичен для построения структуры данных на основе ввода данных, поступающих от какого-то устройства. Глядя на этот код, можно предположить, что каждый раз при выполнении цикла мы тратим **2*sizeof(Node)** байтов памяти (плюс накладные расходы системы управления динамической памятью). К сожалению, нет никаких гарантий, что наши затраты памяти ограничатся ожидаемыми и желательными **2*sizeof(Node)** байтами. В действительности это маловероятно.

Представим себе простой (хотя и вполне вероятный) механизм управления памятью. Допустим также, что объект класса **Message** немного больше, чем объект класса **Node**. Эту ситуацию можно проиллюстрировать следующим образом: темно-серым цветом выделим память, занятую объектом

класса **Message**, светло-серым — память, занятую объектами класса **Node**, а белым — "дыры" (т.е. неиспользуемую память).

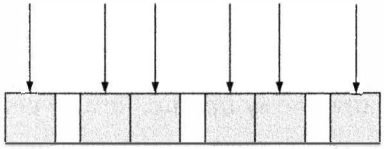


Итак, на каждой итерации остается неиспользованным некоторое количество памяти ("дыра"). Эта память может составлять всего несколько байтов, но если мы не можем использовать их, то это равносильно утечке памяти, а даже малая утечка рано или поздно выводит из строя долговременно работающие системы. Разбиение динамической памяти на многочисленные "дыры", слишком маленькие для того, чтобы в них можно было разместить объекты, называется *фрагментацией памяти* (memory fragmentation). В конечном итоге механизм управления динамической памятью займет все "дыры", достаточно большие для того, чтобы разместить объекты, используемые программой, оставив только слишком маленькие, а потому бесполезные "дыры". Это серьезная проблема для всех достаточно долго работающих программ, широко использующих операторы **new** и **delete**; фрагментация памяти встречается довольно часто. Она сильно увеличивает время, необходимое для выполнения оператора **new**, поскольку он должен выполнить поиск подходящего места для размещения объектов. Совершенно очевидно, что такое поведение для встроенной системы недопустимо. Это может также создать серьезную проблему в небрежно спроектированной системе, не являющейся встроенной.

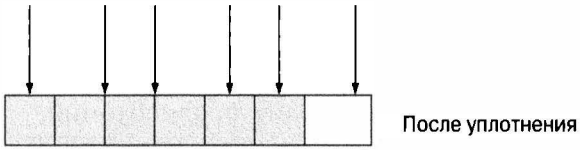
Почему ни язык, ни система не могут решить эту проблему? И нельзя ли написать программу, которая вообще не создавала бы "дыр" в памяти? Сначала рассмотрим наиболее очевидное решение проблемы маленьких бесполезных "дыр" в памяти: попробуем переместить все объекты класса **Node** так, чтобы вся динамическая память была компактной непрерывной областью, в которой можно разместить много объектов.

К сожалению, система не может этого сделать. Причина заключается в том, что код на языке C++ непосредственно ссылается на объекты,

размещенные в памяти. Например, указатели **n1** и **n2** содержат реальные адреса ячеек памяти. Если мы переместим объекты, на которые они указывают, то эти адреса станут некорректными. Допустим, что мы (где-то) храним указатели на созданные объекты. Мы могли бы представить соответствующую часть нашей структуры данных следующим образом.



Теперь мы уплотняем память, перемещая объекты так, чтобы неиспользуемая память стала одним непрерывным фрагментом.



К сожалению, переместив объекты и не обновив указатели, которые на них указывали, мы создали путаницу. Почему же мы не обновляем указатели, перемещая объекты? Мы могли бы написать такую программу, но только зная детали структуры данных. В общем случае система поддержки времени выполнения языка C++ не знает, где хранятся указатели; иначе говоря, если у нас есть объект, то вопрос "Какие указатели указывают на данный объект в данный момент?" не имеет ответа. Но даже если бы эту проблему можно было легко решить, такой подход (известный как *уплотняющая сборка мусора* (compacting garbage collection)) не всегда оправдывает себя. Например, для того чтобы он хорошо работал, обычно требуется, чтобы динамической памяти было в два раза больше, чем памяти, необходимой системе для отслеживания указателей и перемещения объектов. Этой избыточной памяти во встроенной системе может не оказаться. Кроме того, от эффективного механизма уплотняющей сборки мусора трудно добиться предсказуемости.

Мы можем ответить на вопрос "Где находятся указатели?" для наших собственных структур данных и уплотнить их, но более простой подход состоит в том, чтобы вообще изначально избежать фрагментации. В данном примере мы могли бы просто выделить память для обоих объектов **Node** до размещения объекта **Message**.

```
while( ... ) {
    Node* n1 = new Node;
    Node* n2 = new Node;
    Message* p = get_input(dev);
    // ... сохраняем информацию в узлах ...
```

```
    delete p;
    // ...
}
```

Однако перестройка кода для предотвращения фрагментации — в общем случае не такая простая задача. Решить ее надежно как минимум очень трудно, и часто это приводит к противоречиям с другими правилами создания хороших программ. Вследствие этого мы предпочитаем ограничивать использование динамической памяти только способами, не приводящими к фрагментации. Часто предотвратить проблему проще, чем ее решить.

ПОПРОБУЙТЕ

Выполните программу, приведенную выше, и выведите на печать адреса и размеры созданных объектов, чтобы увидеть, как возникают "дыры" в памяти и возникают ли они вообще. Если у вас есть время, можете нарисовать схему памяти, подобную показанным выше, чтобы лучше представить себе, как происходит фрагментация.

25.3.2. Альтернативы динамической памяти

Итак, мы не должны вызывать фрагментацию памяти. Что для этого необходимо сделать? Во-первых, сам по себе оператор **new** не может порождать фрагментацию; для того чтобы возникли "дыры", необходим оператор **delete**. Следовательно, для начала запретим оператор **delete**. В таком случае объект, размещенный в памяти, остается там навсегда.

Если оператор **delete** запрещен, то оператор **new** становится предсказуемым; иначе говоря, все операторы **new** выполняются за одинаковое время. Это правило выполняется во всех доступных реализациях языка, но оно не гарантируется стандартом. Обычно встроенная система имеет последовательность загрузочных команд, приводящую ее в состояние готовности после включения или перезагрузки. На протяжении периода загрузки мы можем распределять память как нам угодно, вплоть до ее полного исчерпания. Мы можем выполнить оператор **new** на этапе загрузки. В качестве альтернативы (или дополнения) можем также зарезервировать глобальную (статическую память) для использования в будущем. Из-за особенностей структуры программы глобальных данных часто лучше избегать, но иногда этот механизм благоразумно использовать для заблаговременного выделения памяти. Точные правила работы данного механизма устанавливаются стандартами программирования данной системы (см. раздел 25.6).

Существуют две структуры данных, которые особенно полезны для предсказуемого выделения памяти.

- *Стеки.* Стек (stack) — это структура данных, в которой можно разместить любое количество данных (не превышающее максимального размера), причем удалить можно только данные, которые были размещены последними; т.е. стек может расти и уменьшаться только на вершине. Он не вызывает фрагментации памяти, поскольку между двумя его ячейками не может быть "дыр".
- *Пулы.* Пул (pool) — это коллекция объектов одинаковых размеров. Мы можем размещать объекты в пуле и удалять их из него, но не можем разместить в нем больше объектов, чем позволяет его размер. Фрагментация памяти при этом не возникает, поскольку объекты имеют одинаковые размеры.

Операции размещения и удаления объектов в стеках и пулах выполняются предсказуемо и быстро.

Таким образом, в системах с жесткими условиями реального времени и в системах, предъявляющих особые требования к обеспечению безопасности, при необходимости можно использовать стеки и пулы. Кроме того, желательно иметь возможность использовать стеки и пулы, разработанные, реализованные и протестированные независимыми поставщиками (при условии, что их спецификации соответствуют нашим требованиям).

Обратите внимание на то, что стандартные контейнеры языка C++ (`vector`, `map` и др.), а также стандартные строки `string` не могут использоваться во встроенных системах непосредственно, потому что они неявно используют оператор `new`. Для того чтобы обеспечить предсказуемость, вы можете создать (или купить, или заимствовать) контейнеры, аналогичные стандартным, но обычные стандартные контейнеры, содержащиеся в вашей реализации языка C++, не предназначены для использования во встроенных системах.

Следует подчеркнуть, что встроенные системы обычно выдвигают очень строгие требования к надежности, поэтому, принимая решение, вы ни в коем случае не должны отказываться от нашего стиля программирования, опускаясь на уровень низкоуровневых средств. Программа, заполненная указателями, явными преобразованиями и другими подобными вещами, редко бывает правильной.

25.3.3. Пример пула

Пул — это структура данных, из которой мы можем доставать объекты заданного типа, а затем удалять их оттуда. Пул содержит максимальное количество объектов, которое задается при его создании. Используя темно-серый цвет для размещенного объекта и светло-серый для места, готового для размещения объекта, мы можем проиллюстрировать пул следующим образом.

Класс **Pool** можно определить так:

```
template<typename T, int N>
class Pool {                    // Пул из N объектов типа T
public:
    Pool();                     // Создаем пул из N объектов типа T
    T* get();                   // Получаем T из пула; если доступных
                                // объектов нет, возвращает 0
    void free(T*);              // Возвращаем T, полученный из пула
                                // с помощью функции get()
    int available() const; // Количество доступных объектов T
private:
    // Память для T[N] и данные, позволяющие определить, какие
    // объекты извлечены из пула, а какие нет (например,
    // список доступных объектов)
};
```

Каждый объект класса **Pool** характеризуется типом элементов и максимальным количеством объектов. Его можно использовать примерно так, как показано ниже.

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

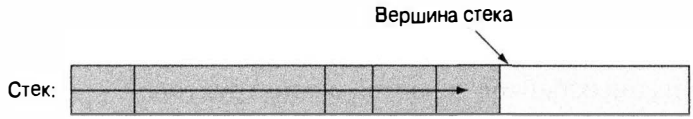
Гарантировать, что пул никогда не исчерпается, — задача программиста. Точный смысл слова "гарантировать" зависит от приложения. В некоторых системах программист должен написать специальный код, например функцию **get()**, которая никогда не будет вызываться, если объектов в пуле больше нет. В других системах программист может проверить результат работы функции **get()** и сделать какие-то корректировки, если результат равен нулю. Характерным примером второго подхода является телефонная система, разработанная для одновременной обработки более ста тысяч звонков. Для каждого звонка выделяется некий ресурс, например буфер номеронабирателя. Если система исчерпывает количество номеронабирателей (например, функция **dial_buffer_pool.get()** возвращает 0), то она запрещает создавать новые соединения (и может прервать несколько существующих соединений, для того чтобы освободить память). В этом случае потенциальный абонент может вновь попытаться установить соединение чуть позднее.

Естественно, наш шаблонный класс **Pool** представляет собой всего лишь один из вариантов общей идеи о пуле. Например, если ограничения на использование памяти не такие строгие, можно определить пулы, в

которых количество элементов определяется конструктором, и даже пулы, количество элементов в которых может впоследствии изменяться, если нам потребуется больше объектов, чем было указано вначале.

25.3.4. Пример стека

Стек — это структура данных, из которой можно брать порции памяти и освобождать последнюю занятую порцию. Используя темно-серый цвет для размещенного объекта и светло-серый для места, готового для размещения объекта, мы можем проиллюстрировать пул следующим образом.



Как показано на рисунке, этот стек "растет" вправо. Стек объектов можно определить так же, как мы определяли пул.

```
template<typename T, int N>
class Stack {   // Стек объектов типа T
    // ...
};
```

Однако в большинстве систем необходимо выделять память для объектов разных размеров. В стеке это можно сделать, а в пуле нет, поэтому мы покажем определение стека, из которого можно брать "необработанную" память для объектов, имеющих разные размеры.

```
template<int N>class Stack { // Стек из N байтов
public:
     Stack();                // Создает стек из N байтов
     void* get(int n);       // Выделяет n байтов из стека; если
                             // свободной памяти нет, возвращает 0
     void free();            // Освобождает последнее значение,
                             // возвращенное функцией get()
     int available() const;  // Количество доступных байтов
private:
    // Память для char[N] и данные, позволяющие определить, какие объекты
    // извлечены из стека, а какие нет (например, указатель на вершину стека)
};
```

Поскольку функция **get()** возвращает указатель **void***, указывающий на требуемое количество байтов, мы должны преобразовать эту память в тип требуемого объекта. Такой стек можно использовать, например, следующим образом.

```
Stack<50*1024> my_free_store; // Стек с 50 Кбайтами памяти

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);
```

```
void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn = new(pv2)Connection(incoming,outgoing,buffer);
```

Использование оператора `static_cast` описано в разделе 17.8. Конструкция `new(pv2)` называется размещающим `new`. Она означает следующее: "Создать объект в памяти, на которую указывает указатель `pv2`". Сама по себе эта конструкция не выделяет никакой памяти. Предполагается, что в классе `Connection` есть конструктор со списком аргументов `(incoming, outgoing,buffer)`. Если это условие не выполняется, то программа не скомпилируется.

Естественно, наш шаблон `Stack` представляет собой всего лишь один из вариантов общей идеи о стеке. Например, если ограничения на использование памяти не такие строгие, мы можем определить стек, в котором количество доступных байтов задается в конструкторе.

25.4. Адреса, указатели и массивы

Предсказуемость требуется в некоторых встроенных системах, а надежность — во всех. Это заставляет нас избегать некоторых языковых конструкций и методов программирования, уязвимых для ошибок (по крайней мере в контексте программирования встроенных систем, если не в общем случае). В языке C++ основным источником проблем является неосторожное использование указателей. Выделим две главные проблемы.

- Явные (непроверяемые и опасные) преобразования.
- Передача указателей на элементы массива.

Первую проблему можно решить, строго ограничив использование явных преобразований типов (приведений). Проблемы, связанные с указателями и массивами, более тонкие, требуют понимания и лучше всего решаются с помощью (простых) классов или библиотечных средств (таких, как класс `array`; см. раздел 20.9). По этой причине в данном разделе мы сосредоточимся на решении второй проблемы.

25.4.1. Непроверяемые преобразования

Физические ресурсы (например, регистры контроллеров во внешних устройствах) и их базовые программные средства управления в низкоуровневой системе обычно находятся по конкретным адресам. Мы должны указать эти адреса в наших программах и присвоить этим данных некоторый тип, например:

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

(См. также раздел 17.8.) Именно этот вид программирования требует постоянного использования документации или справочников. Соответствие между аппаратным ресурсом — адресом регистра/регистров ресурса (выраженного в виде целого числа, часто шестнадцатеричного) — и указателями на программное обеспечение, управляющее аппаратным ресурсом, весьма хрупкое. Вы должны обеспечить его корректность без помощи компилятора (поскольку эта проблема не относится к языку программирования). Обычно основным звеном в цепочке связей между приложением и нетривиальными аппаратными ресурсами является простой (ужасный, полностью непроверяемый) оператор **reinterpret_cast**, преобразующий значение типа **int** в указатель.

Если явные преобразования (**reinterpret_cast**, **static_cast** и т.д.; см. раздел A.5.7) не являются обязательными, избегайте их. Такие преобразования (приведения) бывают необходимыми намного реже, чем обычно считают программисты, работающие в основном на языке C и языке C++ в стиле C.

25.4.2. Проблема: дисфункциональный интерфейс

Как упоминалось ранее (раздел 18.6.1), массив часто передается функции как указатель на элемент (обычно как указатель на первый элемент). В результате он "теряет" размер, поэтому получающая его функция не может непосредственно определить количество элементов, на которые ссылается указатель. Это может вызвать много трудноуловимых и сложно исправимых ошибок. Здесь мы рассмотрим проблемы, связанные с массивами и указателями, и покажем альтернативный вариант. Начнем с примера очень плохого (но, к сожалению, довольно часто встречающегося) интерфейса и попытаемся его улучшить.

```
void poor(Shape* p, int sz)            // Плохой интерфейс
{
    for (int i = 0; i<sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0)   // Очень плохой код
{
    Polygon s1[10];
    Shape s2[10];
    // Инициализация
    Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
    poor(&s0[0],s0.size()); // #1 (передача массива из вектора)
    poor(s1,10);            // #2
    poor(s2,20);            // #3
    poor(p1,1);             // #4
    delete p1;
    p1 = 0;
```

```
    poor(p1,1);              // #5
    poor(q,max);             // #6
}
```

Функция `poor()` представляет собой пример неудачно спроектированного интерфейса: она предоставляет вызывающему модулю массу возможностей для ошибок и не оставляет никаких надежд защититься от них на этапе реализации.

ПОПРОБУЙТЕ

Прежде чем читать дальше, попробуйте выяснить, сколько ошибок вы сможете найти в функции `f()`? В частности, какой из вызовов функции `poor()` может привести к краху программы?

На первый взгляд, приведенные вызовы выглядят отлично, но это тот код, который приносит программистам бессонные ночи отладки и вызывает кошмары у инженеров по качеству.

1. Передается элемент неправильного типа (например, `poor(&s0[0],s0.size())`). Кроме того, вектор `s0` может быть пустым, и в этом случае выражение `&s0[0]` является неверным.
2. Используется "магическая константа" (в данном случае правильная): `poor(s1,10)`. Но тип элемента снова неправильный.
3. Используется "магическая константа" (в данном случае неправильная): `poor(s2,20)`.
4. Первый вызов `poor(p1,1)` правильный (в чем легко убедиться).
5. Передача нулевого указателя при втором вызове `poor(p1,1)`.
6. Вызов `poor(q,max)`, возможно, правильный. Об этом трудно судить, глядя лишь на фрагмент кода. Для того чтобы выяснить, указывает ли указатель `q` на массив, содержащий как минимум `max` элементов, мы должны найти определение указателя `q` и переменной `max` и их значения при данном вызове.

В каждом из перечисленных случаев ошибки оказались очень простыми. Мы не столкнулись с какими-либо скрытыми ошибками, связанными с алгоритмами и структурами данных. Проблема заключается в интерфейсе функции `poor()`, который предусматривает передачу массива по указателю и открывает возможности для появления массы ошибок. Кроме того, вы могли убедиться в том, насколько затрудняют анализ такие малопонятные имена, как `p1` и `s0`. Однако мнемонические, но неправильные имена могут породить еще более сложные проблемы.

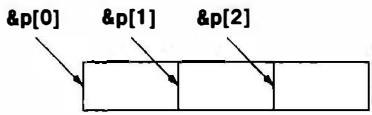
Теоретически компилятор может выявить некоторые из этих ошибок (например, второй вызов `poor(p1,1)`, где `p1==0`), но на практике мы

избежали катастрофы в данном конкретном случае только потому, что компилятор перехватил попытку создания объектов абстрактного класса **Shape**. Однако эта ошибка никак не связана с плохим интерфейсом функции **poor()**, поэтому мы не должны расслабляться. В дальнейшем будем использовать вариант класса **Shape**, который не является абстрактным, так что избежать проблем с интерфейсом нам не удастся.

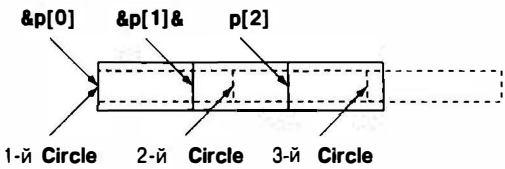
Как мы пришли к выводу, что вызов **poor(&s0[0],s0.size())** является ошибкой? Адрес **&s0[0]** относится к первому элементу массива объектов класса **Circle**; он является значением указателя **Circle***. Мы ожидаем аргумент типа **Shape*** и передаем указатель на объект класса, производного от класса **Shape** (в данном случае **Circle***). Это вполне допустимо: нам необходимо такое преобразование, чтобы можно было обеспечить объектно-ориентированное программирование и доступ к объектам разных типов с помощью общего интерфейса (в данном случае с помощью класса **Shape**) (раздел 14.2). Однако функция **poor()** не просто использует переменную **Shape*** как указатель; она использует ее как массив, индексируя ее элементы.

```
for (int i = 0; i<sz; ++i) p[i].draw();
```

Иначе говоря, она ищет элементы, начиная с ячеек **&p[0]**, **&p[1]**, **&p[2]** и т.д.



В терминах адресов ячеек памяти эти указатели находятся на расстоянии **sizeof(Shape)** друг от друга (см. раздел 17.3.1). К сожалению для модуля, вызывающего функцию **poor()**, значение **sizeof(Circle)** больше, чем **sizeof(Shape)**, поэтому схему распределения памяти можно проиллюстрировать так.



Другими словами, функция **poor()** вызывает функцию **draw()** с указателем, указывающим куда-то в средину объекта класса **Circle**! Скорее всего, это приведет к немедленной катастрофе.

Вызов функции **poor(s1,10)** носит более коварный характер. Он использует "магическую константу", поэтому сразу возникает подозрение, что могут появиться проблемы при сопровождении данной программы, но это более глубокая проблема. Единственная причина, по которой использование массива объектов класса **Polygon** сразу не

привело к проблемам, которые мы обнаружили при использовании объектов класса **Circle**, заключается в том, что класс **Polygon** не добавляет члены класса к базовому классу **Shape** (в отличие от класса **Circle**; см. разделы 13.8 и 13.12), т.е. выполняется условие **sizeof(Shape)==sizeof(Polygon)** и — говоря более общо — класс **Polygon** имеет ту же самую схему распределения памяти, что и класс **Shape**. Иначе говоря, нам просто повезло, так как небольшое изменение определения класса **Polygon** приведет программу к краху. Итак, вызов **poor(s1,10)** работает, но это ошибка, ждущая своего часа. Такой код категорически нельзя назвать качественным.

То, с чем мы столкнулись, является основанием для формулировки универсального правила, согласно которому из утверждения "класс **D** — это разновидность класса **B**" не следует, что "класс **Container<D>** — это разновидность класса **Container<B>**" (см. раздел 19.3.3). Рассмотрим пример.

```
class Circle : public Shape { /* ... */ };

void fv(vector<Shape>&);
void f(Shape &);

void g(vector<Circle>& vd, Circle & d)
{
    f(d);   // OK: неявное преобразование Circle в Shape
    fv(vd); // Ошибка: нет преобразования vector<Circle> в vector<Shape>
}
```

Хорошо, применение функции **poor()** — очень плохой код, но можно ли рассматривать этот код как код встроенной системы? Иначе говоря, следует ли беспокоиться о таких проблемах в приложениях, в которых играют роль безопасность или производительность? Можем ли мы объявить программистам некритичных систем этот код опасным и просто сказать им:

"Не делайте так"? Многие современные встроенные системы основаны на графическом пользовательском интерфейсе, который практически всегда организован в соответствии с принципами объектно-ориентированного программирования. Примеры включают пользовательский интерфейс устройств iPod, интерфейсы некоторых мобильных телефонов и дисплеи операторов в разных системах — вплоть до самолетов. Еще одним примером являются контроллеры схожих устройств (например, множества электромоторов), которые образуют классическую иерархию классов. Другими словами, этот вид кода — и в частности, данный вид объявлений функции — вызывает особые опасения. Нам нужен более безопасный способ передачи информации о наборах данных, который не порождал бы значительных проблем.

Словом, мы не хотим передавать функциям встроенные массивы с помощью указателей и размера массива. Чем же их заменить? Проще всего передать ссылку на контейнер, такой как объект класса **vector**. Проблемы, которые возникают в связи с интерфейсом функции

```
void poor(Shape* p, int sz);
```

просто невозможны в случае

```
void general(vector<Shape>&);
```

Если вы программируете систему, в которой допускаются объекты класса `std::vector` (или его эквиваленты), то просто последовательно используйте в интерфейсах `vector` (или его эквиваленты) и никогда не передавайте встроенный массив с помощью указателя и количества элементов.

Если вы не можете ограничиться использованием класса `vector` или его эквивалентов, то оказываетесь на территории, где не бывает простых решений, — несмотря на всю простоту использования предоставленного нами далее класса (`Array_ref`).

25.4.3. Решение: интерфейсный класс

К сожалению, во многих встроенных системах мы не можем использовать класс `std::vector`, потому что он использует динамическую память. Мы можем решить эту проблему, либо предложив специальную реализацию класса `vector`, либо (что более просто) используя контейнер, который ведет себя как класс `vector`, но не занимающийся управлением памятью.

Прежде чем описать такой интерфейсный класс, перечислим, что мы от него хотели бы.

- Он ссылается на объекты в памяти (но не владеет объектами, не размещает их, не удаляет и т.д.).
- Он знает свой размер (а значит, способен проверять выход за пределы допустимого диапазона).
- Он знает точный тип своих элементов (а значит, не может быть источником ошибок, связанных с типами).
- Его несложно передать (скопировать) как пару (указатель, количество).
- Его *нельзя* неявно преобразовать в указатель.
- Он позволяет легко выделить поддиапазон в целом диапазоне.
- Его столь же легко использовать, как встроенные массивы.

Свойство "столь же легко использовать, как встроенные массивы" можно обеспечить лишь приближенно. Мы бы не хотели, чтобы с нашим классом можно было столь же легко допускать ошибки, как и со встроенными массивами.

Рассмотрим пример такого класса.

```
template<typename T>
class Array_ref {
public:
```

```
    Array_ref(T* pp, int s):p{pp},sz{s} {}

    T& operator[](int n) { return p[n]; }
    const T& operator[](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
        if (a.sz!=sz) return false;
        for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
        return true;
    }

    void reset(Array_ref a) { reset(a.p,a.sz); }
    void reset(T* pp, int s) { p=pp; sz=s; }

    int size() const { return sz; }
    // Операции копирования – по умолчанию:
    // Array_ref не владеет никакими ресурсами
    // Array_ref имеет семантику ссылки
private:
    T* p;
    int sz;
};
```

Класс **Array_ref** близок к минимальному.

- В нем нет функций **push_back()** (для нее нужна динамическая память) и **at()** (требующая исключения).
- Класс **Array_ref** является разновидностью ссылки, поэтому операция копирования просто копирует пары (**p**,**sz**).
- С помощью инициализации разными массивами мы можем получить объекты класса **Array_ref**, которые имеют один и тот же тип, но разные размеры.
- Обновляя пару (**p**,**size**) с помощью функции **reset()**, можно изменить размер существующего класса **Array_ref** (многие алгоритмы требуют указания поддиапазонов).
- В классе **Array_ref** нет интерфейса итераторов (но при необходимости этот недостаток легко устранить). Фактически концепция класса **Array_ref** очень близка к диапазону, заданному двумя итераторами.

Класс **Array_ref** не владеет своими элементами и не управляет памятью, он просто представляет собой механизм доступа к последовательности элементов и их передачи функциям. В этом он отличается от класса **array** из стандартной библиотеки (см. раздел 20.9).

Для того чтобы облегчить создание объектов класса **Array_ref**, добавим несколько вспомогательных функций.

```
template<typename T> Array_ref<T> make_ref(T* pp, int s)
{
```

```
    return (pp) ? Array_ref<T>{pp,s} : Array_ref<T>{nullptr,0};
}
```

Если мы инициализируем объект класса **Array_ref** указателем, то должны явно указать его размер. Это очевидный недостаток, поскольку, задавая размер, легко ошибиться. Кроме того, он открывает возможности для использования указателя, представляющего собой результат неявного преобразования массива производного класса в указатель базового класса, например указателя **Polygon[10]** в указатель **Shape*** (ужасная проблема, описанная в разделе 25.4.2), но иногда у нас просто не остается выхода, кроме как довериться программисту.

Мы решили проявить осторожность в отношении нулевых указателей (поскольку это обычный источник проблем) и пустых векторов.

```
template<typename T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>{&v[0],v.size()}
                      : Array_ref<T>{0,0};
}
```

Идея заключается в том, чтобы передавать массив элементов вектора. Мы выбрали класс **vector**, хотя он часто не подходит для систем, в которых класс **Array_ref** может оказаться полезным. Дело в том, что он обладает ключевыми свойствами многих контейнеров, которые могут быть здесь использованы (например, контейнеров, основанных на пулах; см. раздел 25.3.3).

В заключение предусмотрим обработку встроенных массивов в ситуациях, в которых компилятор знает их размер.

```
template <typename T, int s> Array_ref<T> make_ref(T(&pp)[s])
{
    return Array_ref<T>{pp,s};
}
```

Забавное выражение **T(&pp)[s]** объявляет аргумент **pp** ссылкой на массив из **s** элементов типа **T**. Это позволяет нам инициализировать объект класса **Array_ref** массивом с запоминанием его размера. Мы не можем объявить пустой массив, поэтому не обязаны проверять, есть ли в нем элементы.

```
Polygon ar[0]; // Ошибка: элементов нет
```

Используя данный вариант класса **Array_ref**, мы можем переписать наш пример следующим образом.

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i<a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
```

```
{
    Polygon s1[10];
    Shape s2[20];
    // Инициализация
    Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
    better(make_ref(s0));    // Ошибка: требуется Array_ref<Shape>
    better(make_ref(s1));    // Ошибка: требуется Array_ref<Shape>
    better(make_ref(s2));    // ОК (преобразование не требуется)
    better(make_ref(p1,1)); // ОК: один элемент
    delete p1;
    p1 = 0;
    better(make_ref(p1,1)); // ОК: нет элементов
    better(make_ref(q,max));// ОК (если max задано корректно)
}
```

Мы видим улучшения.

- Код стал проще. Программисту редко приходится заботиться о размерах объектов, но когда это приходится делать, то только в одном месте (при создании объекта класса **Array_ref**), а не в разных местах программы.
- Решена проблема с типами, связанная с преобразованиями **Circle[]** в **Shape[]** и **Polygon[]** в **Shape[]**.
- Неявно решены проблемы с неправильным количеством элементов объектов **s1** и **s2**.
- Потенциальная проблема с переменной **max** (и другими счетчиками количества элементов, необходимыми при использовании указателей) становится более явной — это единственное место, где мы должны явно указывать размер.
- Мы получаем неявную систематическую обработку нулевых указателей и пустых векторов.

25.4.4. Наследование и контейнеры

Но что если мы хотим обрабатывать коллекцию объектов класса **Circle** как коллекцию класса **Shape**, т.е. если действительно хотим, чтобы функция **better()** (представляющая собой вариант нашей старой знакомой функции **draw_all()**; см. разделы 19.3.2 и 22.1.3) реализовала полиморфизм? По существу, мы не можем этого сделать. В разделах 19.3.3 и 25.4.2 показано, что система типов имеет веские основания отказаться воспринимать тип **vector<Circle>** как **vector<Shape>**. По той же причине она отказывается принимать тип **Array_ref<Circle>** как **Array_ref<Shape>**. Если вы не помните, почему, то перечитайте раздел 19.3.3, поскольку данный момент очень важен, даже если это кажется неудобным и мешающим.

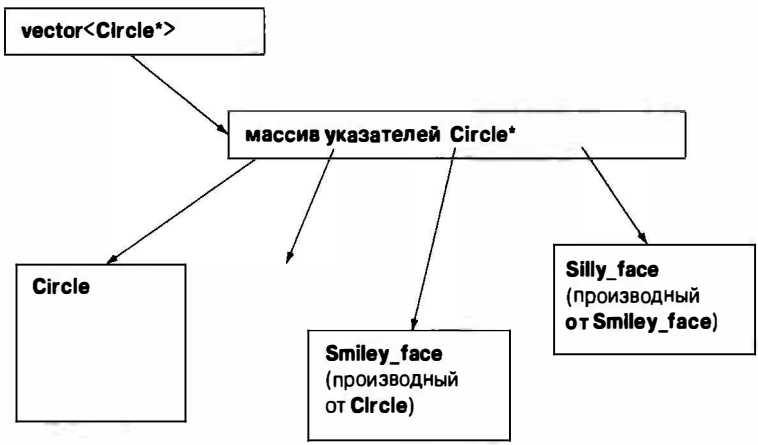
Кроме того, чтобы сохранить полиморфизм времени выполнения, мы должны работать с нашими полиморфными объектами с помощью указателей (или ссылок), но точка в выражении `a[i].draw()` в функции `better()` указывает на обратное. Когда мы видим в этом выражении точку, а не стрелку (`->`), следует ожидать проблем с полиморфизмом.

Что же нам делать? Во-первых, мы должны работать с указателями (или ссылками), а не с самими объектами, поэтому следует попытаться использовать классы `Array_ref<Circle*>`, `Array_ref<Shape*>` и тому подобные, а не `Array_ref<Circle>`, `Array_ref<Shape>` и т.п.

Однако мы по-прежнему не можем конвертировать класс `Array_ref<Circle*>` в класс `Array_ref<Shape*>`, поскольку затем мы можем поместить в контейнер `Array_ref<Shape*>` элементы, которые не имеют тип `Circle*`. Правда, существует одна лазейка.

- Мы не хотим модифицировать наш объект класса `Array_ref<Shape*>`; мы хотим просто рисовать объекты класса `Shape`! Это интересный и полезный частный случай: наш аргумент против преобразования типа `Array_ref<Circle*>` в `Array_ref<Shape*>` не относится к ситуациям, в которых мы не будем модифицировать `Array_ref<Shape*>`.
- Все массивы указателей имеют одну и ту же схему размещения в памяти (независимо от объектов, на которые они указывают), поэтому нас не должна волновать проблема, упомянутая в разделе 25.4.2.

Иначе говоря, не произойдет ничего плохого, если объект класса `Array_ref<Circle*>` будет интерпретирован как *неизменяемый* объект класса `Array_ref<Shape*>`. Итак, нам надо "просто" найти способ это сделать. Рассмотрим диаграмму.



Нет никаких логических препятствий интерпретировать данный массив указателей типа `Circle*` как неизменяемый массив указателей типа `Shape*` (из контейнера `Array_ref`).

Похоже, что мы забрели на территорию экспертов. Эта проблема очень сложная, и ее невозможно решить с помощью рассмотренных ранее средств. Однако давайте рассмотрим, как получить почти идеальную альтернативу дисфункциональному, но весьма популярному стилю интерфейса (указатель плюс количество элементов; см. раздел 25.4.2). Пожалуйста, запомните: никогда не заходите на территорию экспертов просто для того, чтобы продемонстрировать, какой вы умный. В большинстве случаев намного лучше найти библиотеку, которую некие эксперты уже спроектировали, реализовали и протестировали для вас.

Сначала мы переделаем функцию **better()** так, чтобы она использовала указатели и гарантировала, что мы ничего не натворим с контейнером-аргументом.

```
void better2(const Array_ref<Shape*const> a)
{
    for (int i = 0; i<a.size(); ++i)
        if (a[i])
            a[i]->draw();
}
```

Теперь мы работаем с указателями, поэтому должны предусмотреть проверку на нулевой указатель. Для того чтобы гарантировать, что функция **better2()** не модифицирует наши массивы и векторы небезопасным путем через контейнер **Array_ref**, мы добавили несколько квалификаторов **const**. Первый квалификатор **const** гарантирует, что мы не применим к объекту класса **Array_ref** модифицирующие операции, такие как **assign()** и **reset()**. Второй квалификатор **const** размещен после звездочки (*). Это значит, что мы хотим иметь константный указатель (а не указатель на константы); иначе говоря, мы не будем модифицировать указатели на элементы, даже если у нас есть операции, позволяющие это сделать.

Далее, мы должны решить главную задачу: как выразить идею, что объект класса **Array_ref<Circle*>** можно конвертировать

- в нечто подобное объекту класса **Array_ref<Shape*>** (который можно использовать в функции **better2()**);
- но только в неизменяемую версию объекта класса **Array_ref<Shape*>**.

Это можно сделать, добавив в класс **Array_ref** оператор преобразования.

```
template<typename T>
class Array_ref {
public:
    // Как и ранее

    template<typename Q>
    operator const Array_ref<const Q>()
    {
```

```
        // Проверка неявного преобразования элементов:
        static_cast<Q>(*static_cast<T*>(nullptr));

        // Приведение класса Array_ref:
        return Array_ref<const Q>{reinterpret_cast<Q*>(p),sz};
    }

    // Как и ранее
};
```

Это похоже на головоломку, но все же перечислим ее основные моменты.

- Оператор выполняет приведение к типу **Array_ref<const Q>** для каждого типа **Q**, для которого допустимо приведение элемента контейнера **Array_ref<T>** к элементу контейнера **Array_ref<Q>** (мы не используем результат этого приведения, а только убеждаемся, что такое приведение возможно).
- Мы создаем новый объект класса **Array_ref<const Q>**, используя метод грубой силы (оператор **reinterpret_cast**), чтобы получить указатель на элемент требуемого типа. Зачастую решения методом грубой силы оказываются затратными; в нашем случае никогда не следует использовать преобразование **Array_ref** из класса, использующего множественное наследование (раздел А.12.4).
- Обратите внимание на квалификатор **const** в выражении **Array_ref <const Q>**: именно он гарантирует, что мы не можем копировать объект класса **Array_ref<const Q>** в старый, допускающий изменения объект класса **Array_ref<Q>**.

Мы предупреждали вас о том, что зашли на территорию экспертов и столкнулись с головоломкой. Однако эту версию класса **Array_ref** легко использовать (сложными оказываются только его определение и реализация).

```
void f(Shape* q, vector<Circle*>& s0)
{
    Polygon* s1[10];
    Shape* s2[20];
    // Инициализация
    Shape* p1 = new Rectangle{Point{0,0},10};
    better2(make_ref(s0));     // OK: преобразование
                               // в Array_ref<Shape*const>
    better2(make_ref(s1));     // OK: преобразование
                               // в Array_ref<Shape*const>
    better2(make_ref(s2));     // OK (преобразование не требуется)
    better2(make_ref(p1,1));   // Ошибка
    better2(make_ref(q,max)); // Ошибка
}
```

Попытки использовать указатели приводят к ошибкам, потому что они имеют тип **Shape***, а функция **better2()** ожидает аргумент типа **Array_ref<Shape*>**; иначе говоря, функция **better2()** ожидает нечто, содержащее

указатель, а не сам указатель. Если хотите передать функции `better2()` указатель, то поместите его в контейнер (например, во встроенный массив или вектор) и только затем передайте его функции. Для отдельного указателя мы можем использовать неуклюжее выражение `make_ref(&p1,1)`. Однако это решение не подходит для массивов (содержащих более одного элемента), поскольку не предусматривает создание контейнера указателей на объекты.

В заключение резюмируем, что мы можем создавать простые, безопасные, удобные и эффективные интерфейсы, компенсируя недостатки массивов. Это была основная цель данного раздела. Цитата Дэвида Уилера (David Wheeler) "Каждая проблема решается с помощью новой косвенности" считается "первым законом компьютерных наук". Именно так мы решили данную проблему интерфейса.

25.5. Биты, байты и слова

Выше мы уже упоминали о понятиях, связанных с устройством компьютерной памяти, таких как биты, байты и слова, но в общем случае они не относятся к основным концепциям программирования. Вместо этого программисты думают об объектах конкретных типов, таких как `double`, `string`, `Matrix` или `Simple_window`. В этом разделе мы заглянем на уровень программирования, на котором должны лучше разбираться в реальном устройстве памяти компьютера.

Если вы плохо помните двоичное и шестнадцатеричное представления целых чисел, то обратитесь к разделу A.2.1.1.

25.5.1. Операции с битами и байтами

Байт — это последовательность, состоящая из восьми битов.

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Биты в байте нумеруются справа (от самого младшего бита) налево (к самому старшему). Теперь представим слово как последовательность, состоящую из четырех байтов.

3:	2:	1:	0:
0xff	0x10	0xde	0xad

Нумерация байтов в слове также ведется справа налево, т.е. от младшего байта к старшему. Этот рисунок слишком упрощен: существовали компьютеры, в которых байт состоит из девяти битов (правда, за последние десять лет мы не видели ни одного такого компьютера), а машины, в

которых слово состоит из двух байтов, — совсем не редкость. Главное — чтобы вы не забыли обратиться к документации по вашей системе, прежде чем использовать преимущества 8-битового байта или 4-байтового слова.

Для того чтобы ваш код был переносимым, используйте заголовочный файл `<limits>` (см. раздел 24.2.1), чтобы гарантировать правильность ваших предположений о размерах. Вы можете разместить в коде соответствующие проверки:

```
static_assert(4<=sizeof(int),"int слишком мал");
static_assert(!numeric_limits<char>::is_signed,"знаковый char");
```

Первый аргумент `static_assert` представляет собой константное выражение, которое предполагается истинным. Если оно не истинно, т.е. если утверждение ложно, компилятор выводит второй аргумент (строку) в качестве части сообщения об ошибке.

Как представить набор битов в языке C++? Ответ зависит от того, сколько битов вам требуется и какие операции вы хотите выполнять удобно и эффективно. В качестве наборов битов можно использовать целочисленные типы.

- `bool` — 1 бит, правда, занимающий ячейку памяти длиной 8 битов.
- `char` — 8 битов.
- `short` — 16 битов.
- `int` — обычно 32 бита, но во встроенных системах могут быть и 16-битовые целые числа.
- `long int` — 32 или 64 бита (но не меньше количества битов в `int`).
- `long long int` — 32 или 64 бита (но не меньше количества битов в `long int`).

Указанные выше размеры являются типичными, но в разных реализациях они могут быть разными, поэтому в каждом конкретном случае лучше выполнить проверку. Кроме того, стандартная библиотека предоставляет свои средства для работы с битами.

- `std::vector<bool>` — при необходимости иметь больше, чем `8*sizeof(long)` битов.
- `std::bitset` — при необходимости иметь больше, чем `8*sizeof(long)` битов.
- `std::set` — неупорядоченная коллекция именованных битов (см. раздел 21.6.5).
- Файл: множество битов (раздел 25.5.6).

Кроме того, для представления битов можно использовать два средства языка C++.

- Перечисления (`enum`); см. раздел 9.5.
- Битовые поля; см. раздел 25.5.5.

Это разнообразие способов представления битов объясняется тем, что в конечном счете все, что существует в компьютерной памяти, представляет собой набор битов, так что люди испытывают необходимость иметь разные способы их просмотра, именования и выполнения операций над ними. Обратите внимание на то, что все встроенные средства языка работают с фиксированным количеством битов (например, 8, 16, 32 или 64), чтобы компьютер мог выполнять логические операции над ними с оптимальной скоростью, используя операции, обеспечиваемые непосредственно аппаратным обеспечением. В противоположность им средства стандартной библиотеки позволяют работать с произвольным количеством битов. Это может ограничивать производительность, но не следует беспокоиться об

этом заранее: библиотечные средства могут быть — и часто бывают — оптимизированными для эффективной работы, когда количество выбранных вами битов соответствует требованиям аппаратного обеспечения.

Рассмотрим сначала целые числа. Для них в языке C++ предусмотрены побитовые логические операции, непосредственно реализуемые аппаратным обеспечением. Эти операции применяются к каждому биту своих операндов.

	Побитовые операции	
`|`	Или	`n`-й бит числа `x|y` равен 1, если `n`-й бит числа `x` или `n`-й бит числа `y` равен 1
`&`	И	`n`-й бит числа `x&y` равен 1, если и `n`-й бит числа `x`, и `n`-й бит числа `y` равны 1.
`^`	Исключающее или	`n`-й бит числа `x^y` равен 1, если либо `n`-й бит числа `x`, либо `n`-й бит числа `y` равен 1, но не оба одновременно
`<<`	Сдвиг влево	`n`-й бит числа `x<<s` равен (`n+s`)-му биту числа `x`
`>>`	Сдвиг вправо	`n`-й бит числа `x>>s` равен (`n-s`)-му биту числа `x`
`~`	Дополнение	`n`-й бит числа `~x` противоположен `n`-му биту числа `x`

Вам может показаться странным то, что в число фундаментальных операций мы включили "исключающее или" (операцию `^`, которую иногда называют "хог"). Однако эта операция играет важную роль во многих графических и криптографических программах.

Компилятор, в отличие от человека, никогда не перепутает побитовый логический оператор `<<` с оператором вывода. Для того чтобы этого не случалось, помните, что левым операндом оператора вывода является объект класса `ostream`, а левым операндом логического оператора — целое число.

Следует подчеркнуть, что оператор `&` отличается от оператора `&&`, а оператор `|` отличается от оператора `||` тем, что они применяются к каждому

биту своих операндов по отдельности (раздел A.5.5), а их результат состоит из такого же количества битов, что и операнды. В противоположность этому операторы `&&` и `||` просто возвращают значение `true` или `false`.

Рассмотрим несколько примеров. Обычно битовые комбинации выражаются в шестнадцатеричном виде. Для полубайта (четыре бита) мы имеем следующее.

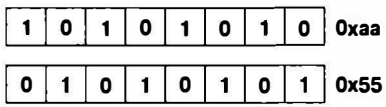
Шестнадцатеричный код	Биты	Шестнадцатеричный код	Биты
`0x0`	`0000`	`0x8`	`1000`
`0x1`	`0001`	`0x9`	`1001`
`0x2`	`0010`	`0xa`	`1010`
`0x3`	`0011`	`0xb`	`1011`
`0x4`	`0100`	`0xc`	`1100`
`0x5`	`0101`	`0xd`	`1101`
`0x6`	`0110`	`0xe`	`1110`
`0x7`	`0111`	`0xf`	`1111`

Для представления чисел, не превышающих девяти, можно было бы просто использовать десятичные цифры, но применение шестнадцатеричного префикса позволяет не забывать, что мы работаем с битовыми комбинациями. Для байтов и слов шестнадцатеричное представление становится действительно полезным. Биты, входящие в состав байта, можно выразить с помощью двух шестнадцатеричных цифр.

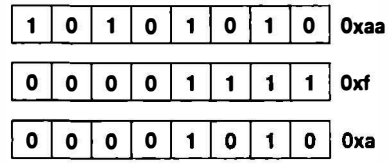
Шестнадцатеричный код	Биты
`0x00`	`0000 0000`
`0x0f`	`0000 1111`
`0xf0`	`1111 0000`
`0xff`	`1111 1111`
`0xaa`	`1010 1010`
`0x55`	`0101 0101`

Так что, используя для простоты тип `unsigned` (раздел 25.5.3), можно написать следующие фрагменты кода.

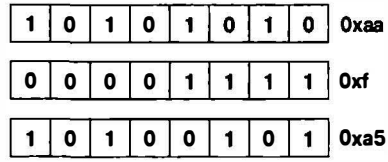
```
unsigned char a = 0xaa;
unsigned char x0 = ~a;  // Дополнение a
```



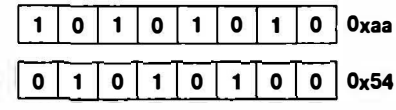

```
unsigned char b = 0x0f;
unsigned char x1 = a&b; // a и b
```




```
unsigned char x2 = a^b; // Исключающее или: a xor b
```

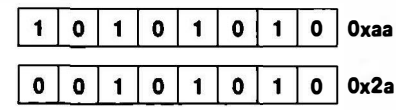



```
unsigned char x3 = a<<1; // Сдвиг влево на один разряд
```



Вместо бита, который был "вытолкнут" с самой старшей позиции, в самой младшей позиции появляется нуль, так что байт остается заполненным, а крайний слева бит (бит 7) просто исчезает.

```
unsigned char x4 == a>>2; // Сдвиг вправо на два разряда
```



В позициях двух старших битов появились нули, которые обеспечивают заполнение байта, а крайние справа биты (первый и нулевой) просто исчезают.

Мы можем написать много битовых комбинаций и потренироваться в выполнении операций над ними, но это занятие скоро наскучит. Рассмотрим маленькую программу, переводящую целые числа в их битовое представление.

```
int main()
{
    for (int i; cin>>i; )
        cout << dec << i << "=="
             << hex << "0x" << i << "=="
             << bitset<8*sizeof(int)>(i) << '\n';
}
```

Для того чтобы вывести на печать отдельные биты целого числа, используется класс `bitset` из стандартной библиотеки.

```
bitset<8*sizeof(int)>{i}
```

Класс `bitset` хранит фиксированное количество битов. В данном случае мы использовали количество битов, равное размеру типа `int` — `8*sizeof(int)`, — и инициализировали объект класса `bitset` целым числом `i`.

ПОПРОБУЙТЕ

Скомпилируйте приведенную выше программу и попробуйте поработать с ней, чтобы почувствовать двоичные и шестнадцатеричные представления чисел. Если вас удивляет представление отрицательных чисел, попробуйте еще раз после чтения раздела 25.5.3.

25.5.2. Класс `bitset`

Для представления наборов битов и работы с ними используется стандартный шаблон класса `bitset` из заголовочного файла `<bitset>`. Каждый объект класса `bitset` имеет фиксированный размер, указываемый при его создании.

```
bitset<4>     flags;
bitset<128>   dword_bits;
bitset<12345> lots;
```

Объект класса `bitset` по умолчанию инициализируется нулями, но обычно у него есть инициализатор. Инициализаторами объектов класса `bitset` могут быть целые числа без знака или строки, состоящие из нулей и единиц:

```
bitset<4>     flags = 0xb;
bitset<128>   dword_bits{string{"1010101010101010"}};
bitset<12345> lots;
```

Здесь объект `lots` будет содержать одни нули, а `dword_bits` — 112 нулей, за которыми следуют 16 явно заданных битов. Если вы попытаетесь проинициализировать объект класса `bitset` строкой с символами, отличающимися от `'0'` и `'1'`, будет сгенерировано исключение `std::invalid_argument`.

```
string s;
cin>>s;
bitset<12345> my_bits(s); // Может генерировать исключение
                          // std::invalid_argument
```

К объектам класса `bitset` можно применять обычные операции над битами. Предположим, что переменные `b1`, `b2` и `b3` являются объектами класса `bitset`.

```
b1 = b2&b3; // И
b1 = b2|b3; // Или
b1 = b2^b3; // xor
b1 = ~b2;   // Дополнение
b1 = b2<<2; // Сдвиг влево
b1 = b2>>3; // Сдвиг вправо
```

По существу, при выполнении побитовых операций объект класса **bitset** ведет себя как переменная типа **unsigned int** (раздел 25.5.3), имеющая произвольный, заданный пользователем размер. Все, что можно делать с переменной типа **unsigned int** (за исключением арифметических операций), вы можете делать и с объектом класса **bitset**. В частности, объекты класса **bitset** полезны при вводе и выводе.

```
cin>>b;                 // Считывание объекта класса bitset из потока ввода
cout<<bitset<8>{'c'}; // Вывод битового представления символа 'c'
```

Считывая данные в объект класса **bitset**, поток ввода ищет нули и единицы. Рассмотрим пример.

```
10121
```

В этом случае будут введены символы **101** , а символы **21** останутся в потоке.

Как в байтах и в словах, биты в объектах класса **bitset** нумеруются справа налево (начиная с самого младшего бита и заканчивая самым старшим), поэтому, например, числовое значение седьмого бита равно 2^7.

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Поскольку класс **bitset** поддерживает индексирование битов, для объектов класса **bitset** нумерация является не просто соглашением.

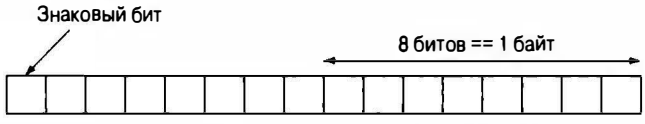
```
int main()
{
    constexpr int max = 10;
    bitset<max> b;
    for (bitset<max> b; cin>>b; ) {
        cout << b << '\n';
        for (int i =0; i<max; ++i)
            cout << b[i];  // Обратный порядок
        cout << '\n';
    }
}
```

Если вам нужна более полная информация о классе **bitset**, ищите ее в документации, справочной системе и учебниках повышенной сложности.

25.5.3. Целые числа со знаком и без знака

Как и во многих языках программирования, целые числа в языке C++ бывают двух видов: со знаком и без него. Целые числа без знака легко представить в памяти компьютера: нулевой бит означает единицу, первый бит — двойку, второй бит — четверку и т.д. Однако представление целого числа со знаком уже создает проблему: как отличить положительные числа от отрицательных? Язык C++ предоставляет разработчикам аппаратного обеспечения определенную свободу выбора, но практически во всех

реализациях используется представление в виде дополнения до двойки. Крайний слева бит (самый старший) считается знаковым.



16-битовое (знаковое) целое число

Если знаковый бит равен единице, то число считается отрицательным. Почти везде для представления целых чисел со знаком используется двоичное дополнение. Для того чтобы сэкономить бумагу, рассмотрим представление четырехбитового целого числа со знаком.

Положительные	0	1	2	4	7
	0000	0001	0010	0100	0111
Отрицательные	1111	1110	1101	1011	1000
	–1	–2	–3	–5	–8

Битовую комбинацию числа `-(x+1)` можно описать как дополнение битов числа `x` (известное также, как `~x`; см. раздел 25.5.1).

До сих пор мы использовали только целые числа со знаком (например, `int`). С учетом беззнаковых чисел правила использования целых чисел можно сформулировать следующим образом.

- Для числовых расчетов используйте знаковые целые числа (например, `int`).
- Для работы с битовыми наборами используйте беззнаковые целые числа (например, `unsigned int`).

Это неплохое эмпирическое правило, но ему трудно следовать, потому что есть люди, которые предпочитают в некоторых арифметических вычислениях работать с целыми числами без знака, и нам иногда приходится использовать их программы. В частности, по историческим причинам, которые возникли еще в первые годы существования языка C, когда числа типа `int` состояли всего из 16 битов и каждый бит был на счету, функция-член `v.size()` класса `vector` возвращает целое беззнаковое число:
Рассмотрим пример.

```
vector<int> v;
// ...
for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';
```

"Услужливый" компилятор может предупредить, что мы смешиваем знаковые значения (т.е. переменную `i`) и беззнаковые (т.е. `v.size()`). Смешение знаковых и беззнаковых чисел, в принципе, может привести к катастрофе. Например, счетчик цикла `i` может оказаться переполненным; иначе говоря, значение `v.size()` может оказаться больше, чем максимально

большое число типа `int` со знаком. В этом случае переменная `i` может достигнуть максимально возможного положительного значения, которое можно представить с помощью типа `int` со знаком (два в степени, равной количеству битов в типе `int`, минус один и еще минус единица, т.е. $2^{15}-1$). При этом следующая операция `++` не сможет вычислить целое число, следующее за максимальным, и вместо этого вернет отрицательное значение. Такой цикл никогда не закончится! Каждый раз, достигая максимально возможного целочисленного значения, мы будем начинать цикл заново с наименьшего отрицательного значения типа `int`. Таким образом, для 16-битовых чисел типа `int` этот цикл содержит ошибку (вероятно, весьма серьезную), если значение `v.size()` равно 32*1024 или больше; для 32-битовых целых чисел типа `int` эта проблема возникнет, только когда счетчик `i` достигнет значения 2*1024*1024*1024.

Таким образом, с формальной точки зрения большинство циклов в этой книге ошибочны и могут вызвать проблемы. Другими словами, для встроенных систем мы должны либо проверять, что цикл никогда не достигнет критической точки, либо заменять его другой конструкцией. Для того чтобы избежать этой проблемы, мы можем использовать тип `size_type`, предоставленный классом `vector`, итераторы либо цикл `for` для диапазона.

```
for (vector<int>::size_type i = 0; i<v.size(); ++i)
    cout << v[i] << '\n';

for (auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << '\n';

for (int x : v) cout << x << '\n';
```

Тип `size_type` — беззнаковый, поэтому в первом варианте имеется на один бит больше, чем при использовании `int`. Это может иметь значение, но соответствующее увеличение слишком мало — только на один бит (что удваивает количество итераций, которые могут быть выполнены). Циклы, использующие итераторы, таких ограничений не имеют.

ПОПРОБУЙТЕ

Следующий пример может показаться безобидным, но он содержит бесконечный цикл:

```
void infinite()
{
    unsigned char max = 160;    // Очень большое
    for (signed char i=0; i<max; ++i)
        cout << int(i) << '\n';
}
```

Выполните его и объясните, почему это происходит.

В основном есть две причины, оправдывающие использование беззнаковых типов для представления обычных целых, а не наборов битов (не использующих операции +, –, * и /).

- Такой выбор позволяет повысить точность на один бит.
- Такой выбор позволяет отразить логические свойства целых чисел в ситуациях, когда они не могут быть отрицательными.

Проблема, связанная с использованием как знаковых, так и беззнаковых целых чисел, заключается в том, что в языке C++ (как и в языке C) они преобразовываются одно в другое удивительным и труднозапоминаемым образом. Рассмотрим пример.

```
unsigned int ui = -1;
int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

Удивительно, но факт: первая инициализация проходит успешно, и переменная `ui` становится равной 4 294 967 295. Это число представляет собой 32-битовое целое число без знака с тем же битовым представлением, что и целое число –1 без знака (одни единицы). Одни программисты считают это вполне допустимым и используют число –1 как сокращенную запись числа, состоящего из одних единичных битов; другие рассматривают это как проблему. То же самое правило преобразования применимо к переводу беззнаковых чисел в знаковые, так что переменная `si` примет значение –1. То, что переменная `si2` станет равной 1 (–1+2=1), мы ожидали, но переменная `ui2` снова нас удивляет: почему 4 294 967 295+2 равно 1? Посмотрим на 4 294 967 295 как на шестнадцатеричное число (`0xffffffff`), и ситуация станет понятнее: 4 294 967 295 — это наибольшее 32-битовое беззнаковое целое число, поэтому 4 294 967 297 невозможно представить в виде 32-битового целого числа — не важно, со знаком или без. Поэтому следует сказать, что операция 4 294 967 295+2 приводит к переполнению или (что точнее) что целые числа без знака поддерживают модульную арифметику. Иначе говоря, арифметика 32-битовых целых чисел является арифметикой по модулю 2^{32}.

Вам пока что все понятно? Даже если так, мы все равно убеждены, что использование целых чисел без знака ради дополнительного повышения точности на один бит — это игра с огнем. Она может привести к путанице и стать источником ошибок.

Что произойдет при переполнении целого числа? Рассмотрим пример.

```
Int i = 0;
while (++i) print(i); // Выводим i как целое с пробелом
```

Какая последовательность значений будет выведена на экран? Очевидно, что это зависит от определения типа `Int` (на всякий случай отметим,

что прописная буква `I` не является опечаткой). Работая с целочисленным типом, имеющим ограниченное количество битов, мы в конечном итоге получим переполнение. Если тип `Int` беззнаковый (например, `unsigned char`, `unsigned int` или `unsigned long long`), то операция ++ является операцией модульной арифметики, и после наибольшего числа, которое мы можем представить, мы получим нуль (и цикл завершится). Если же тип `Int` является целым числом со знаком (например, `signed char`), то числа внезапно станут отрицательными и цикл будет продолжаться, пока счетчик не станет равным нулю (и цикл завершится). Например, для типа `signed char` мы увидим на экране числа 1 2 ... 126 127 –128 –127 ... –2 –1.

Что происходит при переполнении целых чисел? В этом случае мы работаем так, будто в нашем распоряжении есть достаточное количество битов, и отбрасываем ту часть целого числа, которая не помещается в память, где мы храним результат. Эта стратегия приводит к потере крайних слева (самых старших) битов. Тот же эффект можно получить при присваивании.

```
int si = 257;              // Не помещается в тип char
char c = si;               // Неявное преобразование в char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';

si = 129;                  // Не помещается в signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

Получаем следующий результат:

```
257   1     1     1
129   -127  129   -127
```

Объяснение этого результата таково: число 257 на два больше, чем можно представить с помощью восьми битов (255 представляет собой "восемь единичных битов"), а число 129 на два больше, чем можно представить с помощью семи битов (127 равно "семь единичных битов"), поэтому устанавливается знаковый бит. Кстати, эта программа демонстрирует, что тип `char` на нашем компьютере имеет знак (переменная `c` ведет себя как переменная `sc` и отличается от переменной `uc`).

ПОПРОБУЙТЕ

Напишите эти битовые комбинации на листке бумаги. Затем попытайтесь вычислить результат для `si=128`. После этого выполните программу и сравните свое предположение с результатом вычислений на компьютере.

Кстати, почему мы использовали функцию `print()`? Ведь мы могли бы использовать оператор вывода.

```
cout << i << ' ';
```

Однако, если бы переменная `i` имела тип `char`, мы увидели бы на экране символ, а не целое число. По этой причине, для того чтобы единообразно обрабатывать все целочисленные типы, мы и определили

```
template<typename T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }

void print(signed char i) { cout << int(i) << '\t'; }

void print(unsigned char i) { cout << int(i) << '\t'; }
```

Вывод: вы можете использовать целые беззнаковые числа вместо знаковых (включая обычную арифметику), но лучше этого избегать, так как это ненадежно и приводит к ошибкам.

- Никогда не используйте целые числа без знака просто для того, чтобы получить еще один бит точности.
- Если вам необходим один дополнительный бит, то вскоре вам потребуется еще один.

К сожалению, мы не можем полностью избежать использования арифметики целых чисел без знака.

- Индексирование контейнеров в стандартной библиотеке осуществляется беззнаковыми целыми числами.
- Некоторые программисты любят беззнаковую арифметику.

25.5.4. Работа с битами

Зачем вообще нужно работать с отдельными битами? Ну, большинство из нас предпочли бы этого не делать. Возня с битами относится к низкому уровню и открывает возможности для ошибок, поэтому, если у нас есть альтернатива, следует использовать ее. Однако биты настолько важны и полезны, что многие программисты не могут их игнорировать. Это может звучать как довольно грозное и обескураживающее предупреждение, но оно хорошо продумано. Некоторые люди действительно *любят* возиться с битами и байтами, так что следует помнить, что работа с битами иногда необходима (и даже может приносить удовольствие[1]), но ею не следует злоупотреблять. Процитируем Джона Бентли: "Люди, развлекающиеся с битами, будут биты" ("People who play with bits will be bitten").

---

[1] Не зря одна из лучших книг на эту тему (Уоррен Генри С. *Алгоритмические трюки для программистов, 2-е изд.* — М.: "ООО И.Д. Вильямс", 2014) в оригинале так и называется: Hacker's Delight (Удовольствие хакера). — *Примеч. ред.*

Итак, когда мы должны работать с битами? Иногда они являются естественными объектами нашей предметной области, так что операции над битами являются естественными операциями в таких приложениях. Примерами могут служить аппаратные индикаторы ("флаги"), низкоуровневые коммуникации (в которых мы должны извлекать значения разных типов из потока байтов), графика (где мы должны составлять рисунки из нескольких уровней образов) и шифрование (подробнее о нем — в следующем разделе).

В качестве примера рассмотрим, как извлечь (низкоуровневую) информацию из целого числа (возможно, потому, что мы хотим передать его как набор байтов через бинарный механизм ввода-вывода).

```
void f(short val)    // Пусть размер int - 16 битов (2 байта)
{
    unsigned char right = val&0xff; // Крайний справа (младший) байт
    unsigned char left = val>>8;    // Крайний слева (старший) байт
    // ...
    bool negative = val&0x8000;     // Знаковый бит
    // ...
}
```

Такие операции — не редкость. Они известны как "сдвиг и наложение маски" ("shift and mask"). Мы выполняем сдвиг ("shift"), используя операторы `<<` или `>>`, чтобы переместить требуемые биты вправо (в младшую часть слова), где ими легко манипулировать. Мы накладываем маску ("mask"), используя оператор "и" (`&`) вместе с битовой комбинацией (в данном случае `0xff`), чтобы исключить (установить равными нулю) биты, которых не должно быть в результате.

При необходимости именовать биты часто используются перечисления:

```
enum Printer_flags {
    acknowledge =1,
    paper_empty =1<<1,
    busy        =1<<2,
    out_of_black=1<<3,
    out_of_color=1<<4,
    // ...
};
```

Этот код определяет перечисление, в котором каждый элемент равен именно тому значению, которому соответствует его имя.

```
out_of_color     16     0x10     0001 0000
out_of_black      8      0x8     0000 1000
busy              4      0x4     0000 0100
paper_empty       2      0x2     0000 0010
acknowledge       1      0x1     0000 0001
```

Такие значения полезны, потому что их можно комбинировать независимо друг от друга.

```
unsigned char x = out_of_color | out_of_black; // x = 24 (16+8)
x |= paper_empty;                              // x = 26 (24+2)
```

Заметим, что оператор `|=` можно прочитать как "установить бит" (или "установить некоторые биты"). Аналогично оператор `&` можно прочитать как "установлен ли бит?":

```
if (x& out_of_color) { // Установлен ли бит out_of_color?
// ...                 // Да, этот бит установлен
}
```

Оператор `&` можно использовать и для работы с маской.

```
unsigned char y = x &(out_of_color | out_of_black); // y = 24
```

Теперь переменная `y` содержит копию битов числа `x` из позиций 4 и 3 (`out_of_color` и `out_of_black`).

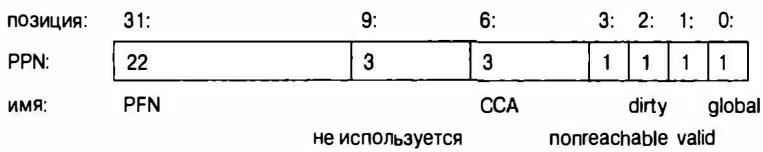
Перечисления `enum` очень часто используются как набор битов. При этом для того, чтобы результат имел вид перечисления, необходимо выполнить обратное преобразование:

```
// Необходимо приведение
Flags z = Printer_flags(out_of_color | out_of_black);
```

Приведение необходимо потому, что компилятор не может знать, что результат выражения `out_of_color | out_of_black` является корректным значением переменной типа `Flags`. Скептицизм компилятора обоснован: помимо всего прочего, ни один из элементов перечисления не имеет значения, равного 24 (`out_of_color | out_of_black`), но в данном случае мы знаем, что выполненное присваивание имеет смысл (а компилятор — нет).

25.5.5. Битовые поля

Как упоминалось ранее, биты часто встречаются при работе с интерфейсами аппаратного обеспечения. Как правило, такие интерфейсы определяются как комбинация битов и чисел разных размеров. Эти биты и числа обычно имеют имена и находятся в заданных позициях слова, которое часто называют *регистром устройства* (device register). В языке C++ есть специальные языковые конструкции для работы с такими фиксированными схемами — *битовые поля* (bitfields). Рассмотрим номер страницы, используемый менеджером страниц глубоко внутри операционной системы. Вот как выглядит диаграмма, приведенная в руководстве по работе с операционной системой.



32-битовое слово состоит из двух числовых полей (одно длиной 22 бита и другое — 3 бита) и четырех флагов (по одному биту каждый). Размеры и позиции этих фрагментов фиксированы. Внутри слова есть также неиспользуемое (и неименованное) поле. Такую схему можно описать с помощью следующей структуры:

```
struct PPN {                    // Номер физической страницы R6000
    unsigned int PFN : 22 ;     // Номер страничного кадра
    int : 3 ;                   // Не используется
    unsigned int CCA : 3 ;      // Алгоритм когерентности кеша
    bool nonreachable : 1 ;
    bool dirty  : 1 ;
    bool valid  : 1 ;
    bool global : 1 ;
};
```

Для того чтобы узнать, что переменные **PFN** и **CCA** должны интерпретироваться как целые числа без знака, необходимо прочитать документацию. Но мы могли бы восстановить структуру непосредственно по диаграмме. Битовые поля заполняют слово слева направо. Количество битов указывается как целое число после двоеточия. Указать абсолютную позицию (например, бит номер 8) нельзя. Если битовые поля занимают больше памяти, чем размер слова, то поля, которые не помещаются в первое слово, записываются в следующее. Надеемся, что это именно то, что вам нужно. После определения битовое поле используется точно так же, как все прочие переменные.

```
void part_of_VM_system(PPN * p )
{
    // ...
    if (p->dirty) {     // Содержимое изменилось
        // Копируем на диск
        p->dirty = 0 ;
    }
    // ...
}
```

Битовые поля позволяют обойтись без сдвигов и наложения масок для того, чтобы получить информацию, размещенную в средине слова. Например, если объект класса **PPN** называется **pn**, то битовое поле **CCA** можно извлечь следующим образом:

```
unsigned int x = pn.CCA;        // Извлекаем битовое поле CCA
```

Если бы для представления тех же самых битов мы использовали целое число типа **int** с именем **pni**, то нам пришлось бы написать следующий код:

```
unsigned int y = (pni>>4)&0x7; // Извлекаем битовое поле CCA
```

Этот код сдвигает структуру **pni** вправо так, чтобы поле **CCA** стало крайним слева битом, а затем накладывает на оставшиеся биты маску **0x7** (т.е.

оставляет только последние три бита). Если вы посмотрите на машинный код, сгенерированный компилятором для присваивания значения `pn.CCA`, то, скорее всего, обнаружите, что он идентичен только что описанному.

Множество аббревиатур (таких как малоинформативные вне своего контекста `CCA`, `PPN`, `PFN`) типично для низкоуровневых кодов.

25.5.6. Пример: простое шифрование

В качестве примера работы с данными на уровне битов и байтов рассмотрим простой алгоритм шифрования — Tiny Encryption Algorithm (TEA). Он был изобретен Дэвидом Уилером (David Wheeler) в Кембриджском университете (см. раздел 22.2.1). Он небольшой, но обеспечивает превосходную защиту от несанкционированной расшифровки.

Не стоит вникать в этот код слишком глубоко (если только вы не слишком любознательны или не хотите заработать головную боль). Мы приводим его просто для того, чтобы вы почувствовали вкус реального приложения и ощутили пользу работы с битами. Если хотите изучать вопросы шифрования, найдите другой учебник. Более подробную информацию об этом алгоритме и варианты его реализации на других языках программирования можно найти на веб-странице `http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm` или на сайте, посвященном алгоритму TEA и созданном профессором Саймоном Шепердом (Simon Shepherd) из Университета Брэдфорда (Bradford University), Англия. Этот код не является самоочевидным (без комментариев!).

Основная идея шифрования/дешифрования (известного также как кодирование/декодирование) проста. Я хочу отослать вам некий текст, но не хочу, чтобы его прочитал кто-то другой. Поэтому я преобразовываю свой текст так, чтобы он стал непонятным для людей, которые не знают, как именно я его модифицировал, но так, чтобы вы могли выполнить обратное преобразование и прочитать мой текст. Эта процедура называется шифрованием. Для того чтобы зашифровать текст, я использую алгоритм (который мы должны считать известным нежелательным соглядатаям) и строку, которая называется ключом. И у вас, и у меня этот ключ есть (и мы надеемся, что его нет у нежелательного соглядатая). Получив зашифрованный текст, вы расшифруете его с помощью ключа; другими словами, восстановите исходный текст, который я вам отослал.

Алгоритм TEA получает в качестве аргумента два беззнаковых числа типа `long` (`v[0]`, `v[1]`), представляющих собой восемь символов, которые должны быть зашифрованы; массив, состоящий из двух беззнаковых чисел типа `long` (`w[0]`, `w[1]`), в который будет записан результат шифрования; а также массив из четырех беззнаковых чисел типа `long` (`k[0]..k[3]`), который является ключом.

```
void encipher(const unsigned long *const v,
              unsigned long *const w,
              const unsigned long * const k)
{
    static_assert(sizeof(long)==4,
                  "неверный размер long для TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0;
    const unsigned long delta = 0x9E3779B9;

    for (unsigned long n = 32; n-->0; ) {
        y += (z<<4 ^ z>>5) + z^sum + k[sum&3];
        sum += delta;
        z += (y<<4 ^ y>>5) + y^sum + k[sum>>11 & 3];
    }
    w[0]=y;
    w[1]=z;
}
```

Поскольку все данные беззнаковые, мы можем выполнять побитовые операции, не опасаясь сюрпризов, связанных с отрицательными числами. Основные вычисления выполняются с помощью сдвигов (<< и >>), исключающего "или" (^) и побитовой операции "и" (&) наряду с обычным (беззнаковым) сложением. Этот код написан специально для машины, в которой тип `long` занимает четыре байта. Код замусорен "магическими" константами (например, он предполагает, что значение `sizeof(long)` равно 4). В общем случае так поступать не рекомендуется, но в данном конкретном коде все это ограничено одной страницей кода, которую программист с хорошей памятью должен запомнить как математическую формулу. Дэвид Уиллер хотел иметь возможность шифрования в путешествиях без ноутбуков и других устройств. Программа кодирования и декодирования должна быть не только маленькой, но и быстрой. Переменная `n` определяет количество итераций: чем больше количество итераций, тем сильнее шифр. Насколько нам известно, при `n==32` алгоритм TEA никогда не был взломан.

Приведем соответствующую функцию дешифровки.

```
void decipher(const unsigned long *const v,
              unsigned long *const w,
              const unsigned long * const k)
{
    static_assert(sizeof(long)==4,
                  "неверный размер long для TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    const unsigned long delta = 0x9E3779B9;
```

```
    // sum = delta<<5, в целом sum = delta * n
    for (unsigned long n = 32; n-- > 0; ) {
        z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    }
    w[0]=y;
    w[1]=z;
}
```

Мы можем использовать алгоритм TEA для того, чтобы создать файл, который можно передавать по незащищенной линии связи.

```
int main() // Отправитель
{
    const int nchar = 2*sizeof(long); // 64 бита
    const int kchar = 2*nchar;        // 128 битов

    string op;
    string key;
    string infile;
    string outfile;
    cout << "Введите имена входного "
         << "и выходного файлов и ключ:\n";
    cin >> infile >> outfile >> key;
    while (key.size()<kchar) key += '0'; // Дополнение ключа
    ifstream inf(infile.c_str());
    ofstream outf(outfile.c_str());
    if (!inf || !outf) error("Неправильное имя файла");

    const unsigned long* k =
        reinterpret_cast<const unsigned long*>(key.data());

    unsigned long outptr[2];
    char inbuf[nchar];
    unsigned long* inptr =
        reinterpret_cast<unsigned long*>(inbuf);
    int count = 0;

    while (inf.get(inbuf[count])) {
        outf << hex; // Используется шестнадцатеричный вывод
        if (++count == nchar) {
            encipher(inptr,outptr,k);
            // Заполнение ведущими нулями:
            outf << setw(8) << setfill('0') << outptr[0] << ' '
                 << setw(8) << setfill('0') << outptr[1] << ' ';
            count = 0;
        }
    }

    if (count) { // Заполнение
        while(count != nchar) inbuf[count++] = '0';
```

```
        encipher(inptr,outptr,k);
        outf << outptr[0] << ' ' << outptr[1] << ' ';
    }
}
```

Основной частью кода является цикл `while`; остальная часть носит вспомогательный характер. Цикл `while` считывает символы в буфер ввода `inbuf` и каждый раз, когда алгоритму TEA нужны очередные восемь символов, передает их функции `encipher()`. Алгоритм TEA не проверяет символы; фактически он не имеет представления об информации, которая шифруется. Например, вы можете зашифровать фотографию или телефонный разговор. Алгоритму TEA требуется лишь, чтобы на его вход поступало 64 бита (два числа типа `long` без знака), которые он будет преобразовывать. Итак, берем указатель на строку `inbuf`, превращаем его в указатель типа `unsigned long*` и передаем алгоритму TEA. То же самое мы делаем с ключом; алгоритм TEA использует первые 128 битов (четыре числа типа `unsigned long`), поэтому мы дополняем введенную информацию так, чтобы она занимала 128 битов. Последняя инструкция дополняет текст нулями, чтобы его длина была кратной 64 битам (8 байтов) в соответствии с требованием алгоритма TEA.

Как передать зашифрованный текст? Здесь у нас есть выбор, но поскольку текст представляет собой простой набор битов, а не символы кодировки ASCII или Unicode, то мы не можем рассматривать его как обычный текст. Можно было бы использовать бинарный ввод-вывод (раздел 11.3.2), но мы решили выводить числа в шестнадцатеричном виде.

```
5b8fb57c 806fbcce 2db72335 23989d1d 991206bc 0363a308
8f8111ac 38f3f2f3 9110a4bb c5e1389f 64d7efe8 ba133559
4cc00fa0 6f77e537 bde7925f f87045f0 472bad6e dd228bc3
a5686903 51cc9a61 fc19144e d3bcde62 4fdb7dc8 43d565e5
f1d3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dcd8
7115211f dbe32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256da1bf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c866aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 d1eadde7
55f20835 1a6d3a4b 202c36b8 66a1e0f2 771993f3 11d1d0ab
74a8cfd4 4ce54f5a e5fda09d acbdf110 259a1a19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44fcd 96e0425a 72839f71 d5b6427c 214340f9 8745882f
0602c1a2 b437c759 ca0e3903 bd4d8460 edd0551e 31d34dd3
c3f943ed d2cae477 4d9d0b61 f647c377 0d9d303a ce1de974
```

```
f9449784  df460350  5d42b06c  d4dedb54  17811b5f  4f723692
14d67edb  11da5447  67bc059a  4600f047  63e439e3  2e9d15f7
4f21bbbe  3d7c5e9b  433564f5  c3ff2597  3a1ea1df  305e2713
9421d209  2b52384f  f78fbae7  d03c1f58  6832680a  207609f3
9f2c5a59  ee31f147  2ebc3651  e017d9d6  d6d60ce2  2be1f2f9
eb9de5a8  95657e30  cad37fda  7bce06f4  457daf44  eb257206
418c24a5  de687477  5c1b3155  f744fbff  26800820  92224e9d
43c03a51  d168f2d1  624c54fe  73c99473  1bce8fbb  62452495
5de382c1  1a789445  aa00178a  3e583446  dcbd64c5  ddda1e73
fa168da2  60bc109e  7102ce40  9fed3a0b  44245e5d  f612ed4c
b5c161f8  97ff2fc0  1dbf5674  45965600  b04c0afa  b537a770
9ab9bee7  1624516c  0d3e556b  6de6eda7  d159b10e  71d5c1a6
b8bb87de  316a0fc9  62c01a3d  0a24a51f  86365842  52dabf4d
372ac18b  9a5df281  35c9f8d7  07c8f9b4  36b6d9a5  a08ae934
239efba5  5fe3fa6f  659df805  faf4c378  4c2048d6  e8bf4939
31167a93  43d17818  998ba244  55dba8ee  799e07e7  43d26aef
d5682864  05e641dc  b5948ec8  03457e3f  80c934fe  cc5ad4f9
0dc16bb2  a50aa1ef  d62ef1cd  f8fbbf67  30c17f12  718f4d9a
43295fed  561de2a0
```

ПОПРОБУЙТЕ

Ключом было слово **bs**; что представляет собой текст?

Любой эксперт по безопасности скажет вам, что хранить исходный текст вместе с зашифрованным — очень глупо. Кроме того, он обязательно сделает замечания о процедуре заполнения, двухбуквенном ключе и так далее, но наша книга посвящена программированию, а не компьютерной безопасности.

Мы проверили свою программу, прочитав зашифрованный текст и преобразовав его в исходный. Когда вы пишете программу, никогда не пренебрегайте простыми проверками ее корректности.

Центральная часть программы расшифровки выглядит следующим образом:

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0;                    // Завершающий нуль
unsigned long* outptr =
        reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex,
         ios_base::basefield); // Шестнадцатеричный ввод

while (inf>>inptr[0]>>inptr[1]) {
```

```
    decipher(inptr,outptr,k);
    outf<<outbuf;
}
```

Обратите внимание на использование функции

```
inf.setf(ios_base::hex,ios_base::basefield);
```

для чтения шестнадцатеричных чисел. Для дешифровки имеется буфер вывода `outbuf`, который мы обрабатываем как набор битов, используя приведение.

Следует ли рассматривать алгоритм TEA как пример программирования встроенной системы? Не обязательно, но мы можем представить себе ситуацию, в которой необходимо обеспечить безопасность или защитить финансовые транзакции с помощью многих устройств. Алгоритм TEA демонстрирует много свойств хорошего встроенного кода: он основан на понятной математической модели, корректность которой не вызывает сомнений; кроме того, он небольшой, быстрый и непосредственно использует особенности аппаратного обеспечения.

Стиль интерфейса функций `encipher()` и `decipher()` не вполне соответствует нашим вкусам. Однако эти функции были разработаны так, чтобы обеспечить совместимость программ, написанных как на языке C, так и на языке C++, поэтому в них нельзя было использовать возможности языка C+, которыми не обладает язык C. Кроме того, многие "магические константы" являются прямым переводом математических формул.

25.6. Стандарты кодирования

Существует множество источников ошибок. Самые серьезные и трудно исправимые ошибки связаны с проектными решениями высокого уровня, такими как общая стратегия обработки ошибок, соответствие определенным стандартам (или его отсутствие), алгоритмы, представление данных и т.д. Эти проблемы здесь *не* рассматриваются. Вместо этого мы сосредоточимся на ошибках, возникающих из-за плохого стиля, т.е. из-за кода, в котором средства языка программирования используются слишком небрежно или некорректно.

Стандарты кодирования[2] направлены на устранение последней проблемы, устанавливая "фирменный стиль", в соответствии с которым программисты должны использовать средства языка C++, подходящие для конкретного приложения. Например, стандарты кодирования для встроенных систем с жесткими ограничениями реального времени или систем, пред-

[2] В данном случае мы используем термин "кодирование", а не "программирование", как лучше отражающий суть — обеспечение хорошего стиля исходного текста, в то время как программирование включает проектирование, выбор алгоритмов и другие решения, не относящиеся к используемому стилю. — *Примеч. ред.*

назначенных для "вечной" работы, могут запрещать использование оператора `new`. Обычно стандарт кодирования пытается обеспечить схожесть исходных текстов, написанных разными программистами, в отличие от исходных текстов, авторы которых ничем себя не ограничивали, смешивая всевозможные стили. Например, стандарт кодирования может потребовать, чтобы для организации циклов использовались только операторы `for`, запрещая применение операторов `while`. Благодаря этому программы становятся более единообразными, а в больших проектах вопросы сопровождения могут быть крайне важными. Обратите внимание на то, что стандарты предназначены для улучшения кодов в конкретных областях программирования, написанных конкретными программистами. Не существует одного общего стандарта кодирования, приемлемого для всех приложений языка C++ и всех программистов, работающих на этом языке.

Таким образом, проблемы, для устранения которых предназначены стандарты кодирования, порождаются способами, которыми мы пытаемся выразить наши решения, а не внутренней сложностью решаемых задач. Можно сказать, что стандарты кодирования пытаются устранить случайно привнесенную, а не внутренне присущую сложность.

Перечислим основные источники такой дополнительной случайной сложности.

- *Слишком умные программисты*, использующие возможности, которые они не понимают, или получающие удовольствия от чрезмерно усложненных решений.
- *Недостаточно образованные программисты*, не использующие наиболее подходящие возможности языка и библиотек.
- *Необоснованные вариации стилей кодирования*, в которых для решения похожих задач применяются разные инструменты, запутывающие программистов, занимающихся сопровождением системы.
- *Неправильный выбор языка программирования*, приводящий к использованию языковых конструкций, не подходящих для данного приложения или данной группы программистов.
- *Недостаточно широкое использование библиотек*, приводящее к многочисленным специфическим манипуляциям низкоуровневыми ресурсами.
- *Неправильный выбор стандартов кодирования*, порождающий дополнительный объем работы или не позволяющий найти наилучшее решение для определенных классов задач, что само по себе становится источником проблем, для устранения которых вводились стандарты кодирования.

25.6.1. Каким должен быть стандарт кодирования?

Хороший стандарт кодирования должен способствовать написанию хороших программ; т.е. он должен давать программистам ответы на множество мелких вопросов, решение которых в каждом конкретном случае привело бы к большой потере времени. Старая поговорка программистов гласит: "Форма освобождает". В идеале стандарт кодирования должен быть инструктивным, указывающим, что должно делаться. Это кажется очевидным, но многие стандарты кодирования представляют собой всего лишь простые списки запретов, не содержащие объяснений, что же *надо* делать. Простой запрет редко бывает полезным и часто раздражает.

Правила хорошего стандарта кодирования должны допускать проверку, желательно программную. Другими словами, как только вы написали программу, вы должны иметь возможность легко ответить на вопрос: "Не нарушил ли я какое-нибудь правило стандарта кодирования?" Хороший стандарт кодирования должен содержать обоснование своих правил. Нельзя просто заявить программистам: "Потому что вы должны делать именно так!" В ответ на это они возмущаются. И что еще хуже, программисты постоянно стараются опровергнуть те части стандарта кодирования, которые они считают бессмысленными, и эти попытки отвлекают их от полезной работы. Не ожидайте, что стандарты кодирования ответят на все ваши вопросы. Даже самые хорошие стандарты являются результатом компромиссов и часто запрещают делать то, что может вызвать проблемы, даже если в вашей практике этого никогда не случалось. Например, очень часто источником недоразумений становятся противоречивые правила именования; однако программисты часто отдают предпочтение определенным соглашениям об именах и категорически отвергают остальные. Например, лично я считаю, что имена идентификаторов вроде `CamelCodingStyle` весьма уродливы, и очень люблю имена наподобие `underscore_style`, которые намного понятнее, и многие люди со мной согласны. С другой стороны, очень многие разумные люди со мной не согласны. Очевидно, ни один стандарт именования не может удовлетворить всех, но в данном случае, как и во многих других, последовательность намного лучше отсутствия какой-либо систематичности.

Подведем итоги.

- Хороший стандарт кодирования предназначен для конкретной предметной области и конкретной группы программистов.
- Хороший стандарт кодирования должен быть инструктивным, а не запретительным.
  - Рекомендация некоторых базовых библиотечных возможностей часто является самым эффективным способом применения инструктивных правил.

- Стандарт кодирования — это совокупность правил, описывающих, как должен выглядеть исходный текст, в частности:
  - регламентирующие способ именования идентификаторов и выравнивания строк, например "Используйте схему Страуструпа";
  - указывающие конкретное подмножество языка, например "Не используйте операторы `new` и `throw`";
  - задающие правила комментирования, например "Каждая функция должна содержать описание того, что она делает";
  - требующие использовать конкретные библиотеки, например "Используйте библиотеку `<iostream>`, а не `<stdio.h>`", или "Используйте классы `vector` и `string`, а не встроенные массивы и С-строки".
- Большинство стандартов кодирования имеет общие цели, состоящие в повышении
  - надежности,
  - переносимости,
  - удобства сопровождения,
  - удобства тестирования,
  - возможности повторного использования,
  - возможности расширения,
  - удобочитаемости.

- Хороший стандарт кодирования лучше, чем отсутствие стандарта. Не начинайте ни один большой промышленный проект (т.е. проект, в котором задействовано много людей и который продолжается несколько лет), не установив стандарт кодирования.

- Плохой стандарт кодирования может оказаться хуже, чем полное отсутствие стандарта. Например, стандарты кодирования на языке C++, сужающие его до языка С, таят в себе угрозу. К сожалению, плохие стандарты кодирования встречаются чаще, чем хотелось бы.
- Программисты не любят стандарты кодирования, даже хорошие. Большинство программистов хотят писать свои программы только так, как им нравится.

25.6.2. Примеры правил

В этом разделе мы хотели бы дать читателям представление о стандартах кодирования, перечислив некоторые правила. Естественно, мы выбрали те правила, которые считаем полезными для вас. Однако мы не видели ни одного реального стандарта кодирования, который занимал бы меньше

35 страниц. Большинство из них намного длиннее. Так что мы не будем даже пытаться привести здесь полный набор правил. Кроме того, каждый хороший стандарт кодирования предназначен для конкретной предметной области и конкретной группы программистов. По этой причине мы ни в коем случае не претендуем на универсальность.

Правила пронумерованы и содержат (краткое) обоснование. Мы провели различия между *рекомендациями*, которые программист может иногда игнорировать, и *жесткими правилами*, которым он обязан следовать. Такие правила обычно нарушаются только с письменного согласия руководителя. Каждое нарушение рекомендации или жесткого правила требует отдельного комментария в программе. Любые исключения из правила должны быть перечислены в его описании. Ниже каждое жесткое правило выделяется прописной буквой *R* в его номере. Номер рекомендации содержит строчную букву *r*.

Правила разделяются на несколько категорий.

- Общие.
- Препроцессора.
- Использования имен и размещения текста.
- Для классов.
- Для функций и выражений.
- Для систем с жесткими условиями реального времени.
- Для систем, критичных в плане безопасности.

Правила для систем с жесткими условиями реального времени и систем, критичных в плане безопасности, применяются только в проектах, которые явно объявлены таковыми.

По сравнению с хорошими реальными стандартами кодирования наша терминология является недостаточно точной (например, что значит "система, критичная в плане безопасности"?), а правила слишком лаконичны. Сходство между этими правилами и правилами JSF++ (см. раздел 25.6.3) не является случайным; я лично помогал формулировать правила JSF++. Однако примеры исходных текстов в данной книге не следуют этим правилам; в конце концов, книга — это не встроенная система с жесткими требованиями.

Общие правила

R100. Любая функция (или класс) должна содержать не более 200 логических строк кода (без учета комментариев).

Причина. Длина функции (или класса) свидетельствует о ее сложности, поэтому ее трудно понять и протестировать.

r101. Любая функция (или класс) должна помещаться на одном экране и служить для одной логической цели.

Причина. Программист, видящий только часть функции или класса, может не увидеть проблему. Функция, решающая сразу несколько задач, скорее всего, длиннее и сложнее, чем функция, решающая только одну задачу.

R102. Любая программа должна соответствовать стандарту языка C++ ISO/IEC 14882:2011(E).

Причина. Расширения языка или отклонения от стандарта ISO/IEC 14882 менее устойчивы, хуже определены и уменьшают переносимость программ.

Правила препроцессора

R200. Нельзя использовать никаких макросов, за исключением директив управления исходными текстами `#ifdef` и `#ifndef`.

Причина. Макрос не учитывает области видимости и не подчиняется правилам работы с типами. Использование макросов трудно определить визуально, просматривая исходный текст.

R201. Директива `#include` должна использоваться только для включения заголовочных (`*.h`) файлов.

Причина. Директива `#include` используется для доступа к объявлениям интерфейса, а не к деталям реализации.

R202. Директивы `#include` должны предшествовать всем объявлениям, не относящимся к препроцессору.

Причина. Директива `#include`, находящаяся в средине файла, скорее всего, будет не замечена читателем и вызовет недоразумения, связанные с тем, что область видимости разных имен в разных местах разрешается по-разному.

R203. Заголовочные файлы (`*.h`) не должны содержать определения неконстантных переменных, а также функций, не являющихся встраиваемыми или шаблонами.

Причина. Заголовочные файлы должны содержать объявления интерфейсов, а не детали реализации. Однако константы часто рассматриваются как часть интерфейса; некоторые очень простые функции для повышения производительности должны быть встраиваемыми (и потому быть определены в заголовочных файлах), а текущие реализации шаблонов требуют, чтобы полные определения шаблонов содержались в заголовочных файлах.

Правила именования и форматирования текста

R300. В пределах одного и того же исходного файла следует использовать согласованное выравнивание.

Причина. Удобочитаемость и стиль.

R301. Каждая новая инструкция должна начинаться с новой строки.

Причина. Удобочитаемость.

Пример

```
int a = 7; x = a+7; f(x,9); // Нарушение
int a = 7;                  // OK
```

```
x = a+7;                    // ОК
f(x,9);                     // ОК
```

Пример

```
if (p<q) cout << *p;        // Нарушение
```

Пример

```
if (p<q)
cout << *p;             // ОК
```

R302. Имена идентификаторов должны быть информативными.

Идентификаторы могут состоять из общепринятых аббревиатур и акронимов.

При традиционном использовании имена `x`, `y`, `i`, `j` и другие являются информативными.

Следует использовать стиль именования `number_of_elements`, а не `numberOfElements`.

Венгерскую нотацию использовать не следует.

С прописной буквы могут начинаться только имена типов, шаблонов и пространств имен.

Избегайте слишком длинных имен.

Пример. `Device_driver` или `Buffer_pool`.
Причина. Удобочитаемость.
Примечание. Идентификаторы, начинающиеся с символа подчеркивания, зарезервированы стандартом языка C++ и, следовательно, запрещены для использования.
Исключение. При вызове функций из одобренной библиотеки могут использоваться ее имена.
R303. Не следует использовать идентификаторы, которые различаются только перечисленными ниже признаками.

- Комбинация прописных и строчных букв.
- Наличие/отсутствие символа подчеркивания.
- Замена буквы *O* цифрой 0 или буквой *D*.
- Замена буквы *I* цифрой 1 или буквой *l*.
- Замена буквы *S* цифрой 5.
- Замена буквы *Z* цифрой 2.
- Замена буквы *n* буквой *h*.

Пример

```
Head и head             // Нарушение
```

Причина. Удобочитаемость.

R304. Идентификаторы не должны состоять только из прописных букв или прописных букв с подчеркиваниями.
Пример

```
BLUE и BLUE_CHEESE    // Нарушение
```

Причина. Имена, состоящие исключительно из прописных букв, широко используются для названия макросов, которые могут встретиться в заголовочных файлах применяемой библиотеки, включенных директивой.
Исключение. Имена макросов, используемые для защиты директив `#include`.

Правила для функций и выражений

r400. Идентификаторы во вложенной области видимости не должны совпадать с идентификаторами во внешней области видимости.
Пример

```
int var = 9; { int var = 7; ++var; } // Нарушение: var скрывает var
```

Причина. Удобочитаемость.
R401. Объявления должны иметь как можно более малую область видимости.
Причина. Инициализация и использование переменной должны быть как можно ближе друг к другу, чтобы минимизировать вероятность путаницы; выход переменной за пределы области видимости освобождает ее ресурсы.
R402. Переменные должны быть инициализированы.
Пример

```
int var; // Нарушение: переменная var не инициализирована
```

Причина. Неинициализированные переменные являются традиционным источником ошибок.
Исключение. Переменную, которая будет немедленно заполнена входными данными, инициализировать необязательно.
Примечание. Множество типов, таких как `vector` или `string`, имеют конструкторы по умолчанию, которые гарантируют инициализацию.
R403. Не следует использовать операторы приведения.
Причина. Операторы приведения часто являются источником ошибок.
Исключение. Разрешается использовать оператор `dynamic_cast`.
Исключение. Именованные приведения можно использовать для преобразования адресов аппаратного обеспечения в указатели, а также для преобразования указателей типа `void*`, полученных из внешних источников (например, от библиотеки графического пользовательского интерфейса), в указатели соответствующих типов.
R404. Встроенные массивы не должны использоваться в интерфейсах. Иначе говоря, указатель, используемый как аргумент функции, должен рассматриваться только как указатель на отдельный элемент. Для передачи массивов используйте класс `Array_ref`.

Причина. Когда массив передается в вызываемую функцию с помощью указателя, а количество его элементов не передается, может возникнуть ошибка. Кроме того, комбинация неявного преобразования массива в указатель и неявного преобразования объекта производного класса в объект базового класса может привести к повреждению памяти.

Правила для классов

R500. Для классов без открытых членов-данных используйте ключевое слово `class`, а для классов без закрытых членов-данных — ключевое слово `struct`. Не используйте классы, в которых смешаны открытые и закрытые члены-данные.

Причина. Ясность.

r501. Если класс имеет деструктор или член, являющийся указателем или ссылкой, то он должен иметь копирующий конструктор, а оператор копирующего присваивания должен быть либо определен, либо запрещен.

Причина. Деструктор обычно освобождает ресурс. Семантика копирования по умолчанию редко бывает правильной по отношению к членам класса, являющимся указателями или ссылками, а также по отношению к классам с деструкторами.

R502. Если класс содержит виртуальную функцию, то он должен иметь виртуальный деструктор.

Причина. Если класс имеет виртуальную функцию, то, скорее всего, он будет использоваться через интерфейс базового класса. Функция, обращающаяся к этому объекту только через этот базовый класс, может его удалять, поэтому производные классы должны иметь возможность выполнить завершающую очистку с помощью своих виртуальных деструкторов.

r503. Конструктор, принимающий один аргумент, должен быть объявлен с модификатором `explicit`.

Причина. Во избежание непредвиденных неявных преобразований.

Правила для систем с жесткими условиями реального времени

R800. Не следует применять исключения.

Причина. Результат непредсказуем.

R801. Оператор `new` можно использовать только на этапе инициализации приложения.

Причина. Результат непредсказуем.

Исключение. Для памяти, выделяемой в стеке, может быть использован синтаксис размещающего `new` (со стандартной семантикой).

R802. Не следует использовать оператор `delete`.

Причина. Результат непредсказуем; может возникнуть фрагментация памяти.

R803. Не следует использовать оператор `dynamic_cast`.

Причина. Результат непредсказуем (при традиционном способе реализации).

R804. Не следует использовать стандартные библиотечные контейнеры, за исключением класса `std::array`.

Причина. Результат непредсказуем (при традиционном способе реализации).

Правила для систем, критичных в плане безопасности

R900. Операции инкремента и декремента не следует использовать в качестве подвыражений.

Пример:

```
int x = v[++i];    // Нарушение
```

Пример:

```
++i;
int x = v[i];      // ОК
```

Причина. Такой инкремент легко не заметить.

R901. Код не должен зависеть от правил приоритета операций ниже уровня арифметических выражений.

Пример:

```
x = a*b+c;         // ОК
```

Пример:

```
if( a<b || c<=d ) // Нарушение: поместите в скобки инструкции (a<b) и (c<=d)
```

Причина. Путаница с приоритетами постоянно встречается в программах, авторы которых неуверенно знают язык C/C++.

Наша нумерация непоследовательна, поскольку у нас должна быть возможность добавлять новые правила, не нарушая их общую классификацию. Очень часто правила помнят по номерам, поэтому их перенумерация может вызвать неприятие пользователей.

25.6.3. Реальные стандарты кодирования

Существует много стандартов кодирования для языка C++. Применение большинства из них ограничено стенами корпораций, и они недоступны для широкой публики. Во многих случаях это, вероятно, неплохо — за исключением, возможно, программистов, работающих в этих корпорациях. Перечислим стандарты, которые признаны хорошими в своих предметных областях.

1. Google C++ Style Guide: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`. Несколько старомодное и ограниченное, но развивающееся руководство.
2. Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program*. Document

Number 2RDU00001 Rev C. December 2005. В узких кругах широко известен под названием "JSF++". Это набор правил, разработанных в компании Lockheed-Martin Aero, для программного обеспечения летательных аппаратов (читай "самолетов"). Эти правила были написаны программистами и для программистов, создающих программное обеспечение, от которого зависит жизнь людей (`www.stroustrup.com/JSF-AV-rules.pdf`).

3. Programming Research. High-integrity C++ Coding Standard Manual Version 2.4. (`www.programmingresearch.com`).
4. Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.[3] Этот труд можно скорее отнести к "метастандартам кодирования"; иначе говоря, вместо того чтобы формулировать конкретные правила, авторы пишут, какие правила являются хорошими и почему.

Обратите внимание на то, что знания предметной области, языка и технологий программирования не могут быть взаимозаменяемыми. В большинстве приложений — и особенно при программировании большинства встроенных систем — необходимо знать операционную систему и/или архитектуру аппаратного обеспечения. Если вам необходимо выполнить низкоуровневое кодирование на языке C++, просмотрите отчет комитета ISO по стандартизации, посвященный проблемам производительности (ISO/IEC TR 18015; `www.stroustrup.com/performanceTR.pdf`); под производительностью авторы (и мы) понимают, в первую очередь, производительность программирования для встроенных систем.

В мире встроенных систем существует множество языков программирования и их диалектов, но где только можно вы должны использовать стандартизированные язык (например, ISO C++), инструменты и библиотеки. Это минимизирует время вашего обучения и повысит вероятность того, что вас не скоро уволят.

Задание

1. Выполните следующий фрагмент кода:

```
int v = 1;
for (int i = 0; i<sizeof(v)*8; ++i) {
    cout << v << ' '; v <<=1;
}
```

2. Выполните этот фрагмент еще раз, но теперь переменную `v` объявите как `unsigned int`.

---

[3] Имеется русский перевод: Саттер Герб, Александреску Андрей. *Стандарты программирования на C++*. — М.: "ООО И.Д. Вильямс", 2005.

3. Используя шестнадцатеричные литералы, определите, чему равны перечисленные ниже переменные типа `short unsigned int`.
   - В которой каждый бит равен единице.
   - В которой самый младший бит равен единице.
   - В которой самый старший бит равен единице.
   - В которой самый младший байт состоит из одних единиц.
   - В которой самый старший байт состоит из одних единиц.
   - В которой каждый второй бит равен единице (и самый младший бит также равен единице).
   - В которой каждый второй бит равен единице (а самый младший бит равен нулю).
4. Выведите на печать каждое из перечисленных выше значений в виде десятичного и шестнадцатеричного чисел.
5. Выполните задания 3 и 4, используя побитовые операции (`|`, `&`, `<<`) и (исключительно) литералы `1` и `0`.

Контрольные вопросы

1. Что такое встроенная система? Приведите десять примеров, не менее трех из которых не упоминались в этой главе.
2. Что особенного во встроенных системах? Приведите пять распространенных особенностей встроенных систем.
3. Определите понятие предсказуемости в контексте встроенных систем.
4. Почему встроенные системы иногда трудно модифицировать и ремонтировать?
5. Почему иногда нецелесообразно оптимизировать производительность системы?
6. Почему мы предпочитаем высокие уровни абстракции низкоуровневому коду?
7. Какие ошибки называют преходящими? Чем они особенно опасны?
8. Как разработать систему, которая восстанавливает свою работу после сбоя?
9. Почему невозможно предотвратить все сбои?
10. Что такое предметная область? Приведите примеры предметных областей.
11. Для чего необходимо знать предметную область при программировании встроенных систем?

12. Что такое подсистема? Приведите примеры.
13. Назовите три вида памяти с точки зрения языка C++.
14. Когда вы предпочитаете использовать динамическую память?
15. Почему использование динамической памяти во встроенных системах часто нецелесообразно?
16. Когда можно безопасно использовать оператор `new` во встроенной системе?
17. Какие потенциальные проблемы связаны с классом `std::vector` в контексте встроенных систем?
18. Какие потенциальные проблемы связаны с исключениями во встроенных системах?
19. Что такое рекурсивный вызов функции? Почему некоторые программисты, разрабатывающие встроенные системы, их избегают? Что они используют вместо них?
20. Что такое фрагментация памяти?
21. Что такое сборщик мусора (в контексте программирования)?
22. Что такое утечка памяти? Почему она может стать проблемой?
23. Что такое ресурс? Приведите примеры.
24. Что такое утечка ресурсов и как ее систематически предотвратить?
25. Почему мы не можем просто переместить объекты из одной области памяти в другую?
26. Что такое стек?
27. Что такое пул?
28. Почему стек и пул не приводят к фрагментации памяти?
29. Зачем нужен оператор `reinterpret_cast`? Чем он плох?
30. Чем опасна передача указателей в качестве аргументов функции? Приведите примеры.
31. Какие проблемы могут возникать при использовании указателей и массивов? Приведите примеры.
32. Перечислите альтернативы использованию указателей (на массивы) в интерфейсах.
33. Что гласит "первый закон компьютерных наук"?
34. Что такое бит?
35. Что такое байт?
36. Из скольких битов обычно состоит байт?
37. Какие операции мы можем выполнить с наборами битов?

38. Что такое "исключающее или" и чем оно полезно?
39. Как представить набор (или последовательность) битов?
40. Как мы традиционно нумеруем биты в слове?
41. Как мы традиционно нумеруем байты в слове?
42. Что такое слово?
43. Из скольких битов, как правило, состоит слово?
44. Чему равно десятичное значение числа `0xf7`?
45. Какой последовательности битов соответствует число `0xab`?
46. Что такое класс `bitset` и когда он нужен?
47. Чем тип `unsigned int` отличается от типа `signed int`?
48. В каких ситуациях мы предпочитаем использовать тип `unsigned int`, а не `signed int`?
49. Как написать цикл, если количество элементов в массиве очень велико?
50. Чему равно значение переменной типа `unsigned int` после присвоения ей числа `-3`?
51. Почему нам может потребоваться работа с битами и байтами (а не с типами более высокого уровня)?
52. Что такое битовое поле?
53. Для чего используются битовые поля?
54. Что такое кодирование (шифрование)? Для чего оно используется?
55. Можно ли зашифровать фотографию?
56. Для чего предназначен алгоритм TEA?
57. Как вывести число в шестнадцатеричной системе?
58. Для чего нужны стандарты кодирования? Назовите причины.
59. Почему не существует универсального стандарта кодирования?
60. Перечислите некоторые свойства хорошего стандарта кодирования.
61. Как стандарт кодирования может нанести вред?
62. Составьте список, содержащий не менее десяти правил кодирования (которые считаете полезными). Чем они полезны?
63. Почему мы не используем идентификаторы вида `ALL_CAPITAL`?

Термины

`bitset`	жесткие условия реального времени	ресурс
`unsigned`	исключающее или	сборщик мусора
адрес	мягкие условия реального времени	стандарт кодирования

бит	предсказуемость	устройство
битовое поле	пул	утечка
встроенная система	реальное время	шифрование

Упражнения

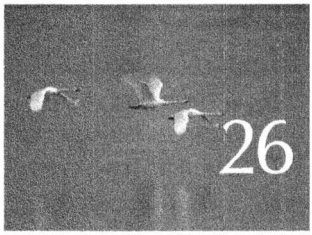
1. Выполните упражнения из разделов **Попробуйте**, если вы этого еще не сделали.
2. Составьте список слов, которые можно получить из записи чисел в шестнадцатеричной системе счисления, читая 0 как *o*, 1 как *l*, 2 как *to* и т.д. Например, *Fool* и *Beef*. Прежде чем сдать их для оценки, тщательно устраните все вульгаризмы.
3. Проинициализируйте 32-битовое целое число со знаком указанной битовой последовательностью и выведите его на экран: все биты нулевые, все биты единичные, чередующиеся нули и единицы (крайний слева бит — единичный), чередующиеся нули и единицы (крайний слева бит — нулевой), шаблон 1100110011..., 0011001100..., чередующиеся байты, состоящие только из единичных и только из нулевых битов, начиная с байта, состоящего из одних нулей. Повторите это упражнение с 32-битовым целым числом без знака.
4. Добавьте в калькулятор из главы 7 побитовые логические операторы `&`, `|`, `^` и `~`.
5. Напишите бесконечный цикл. Выполните его.
6. Напишите бесконечный цикл, который трудно распознать как бесконечный. Годится также цикл, который на самом деле не является бесконечным, потому что закончится после исчерпания ресурса.
7. Выведите шестнадцатеричные значения от 0 до 400; выведите шестнадцатеричные значения от –200 до 200.
8. Выведите числовой код каждого символа на своей клавиатуре.
9. Не используя ни стандартные заголовочные файлы (такие как `<limits>`), ни документацию, вычислите количество битов в типе `int` и определите, имеет ли знак тип `char` в вашей реализации языка C++.
10. Взгляните на пример битового поля из раздела 25.5.5. Напишите пример, в котором сначала инициализируется структура `PPN`, затем считывается и выводится на экран значение каждого ее поля, затем значение каждого поля изменяется (с помощью присваивания) и результат выводится на экран. Повторите это упражнение, сохраняя информацию из структуры `PPN` в 32-битовом целом числе без знака и используя операторы для работы с битами (см. раздел 25.5.4) для доступа к отдельным битам слова.

11. Повторите предыдущее упражнение, сохраняя биты в объекте класса `bitset<32>`.
12. Напишите понятную программу для примера из раздела 25.5.6.
13. Используйте алгоритм TEA (см. раздел 25.5.6) для "безопасной" передачи данных между двумя компьютерами. Использовать электронную почту настоятельно не рекомендуется.
14. Реализуйте простой вектор, в котором могут храниться не более `N` элементов, память для которых выделяется из пула. Протестируйте его при `N==1000` и целочисленных элементах.
15. Измерьте время (см. раздел 26.6.1), которое затрачивается на размещение 10 тысяч объектов случайного размера в диапазоне (0;1000] байтов с помощью оператора `new`; затем измерьте время, которое будет затрачено на удаление этих элементов с помощью оператора `delete`. Сделайте это дважды: один раз освобождая память в обратном порядке, второй раз — случайным образом. Затем выполните то же задание для 10 тысяч объектов размером 500 байтов, выделяя и освобождая память в пуле. После этого разместите 10 тысяч объектов случайного размера в диапазоне (0;1000] байтов, выделяя память в стеке и освобождая ее (в обратном порядке). Сравните результаты измерений. Выполните каждое измерение не менее трех раз, чтобы убедиться в согласованности результатов.
16. Сформулируйте двадцать правил, регламентирующих стиль кодирования (не копируя правила из раздела 25.6). Примените их к программе, состоящей более чем из 300 строк, которую вы недавно написали. Напишите короткий (на одной-двух страницах) комментарий о применении этих правил. Нашли ли вы ошибки в программе? Стал ли код яснее? Может быть, он стал менее понятным? Теперь модифицируйте набор правил, основываясь на своем опыте.
17. В разделах 25.4.3–25.4.4 мы описали класс `Array_ref`, делающий доступ к элементам массива более простым и безопасным. В частности, мы заявили, что теперь наследование обрабатывается корректно. Испытайте разные способы получения указателя `Rectangle*` на элемент массива `vector<Circle*>`, используя класс `Array_ref<Shape*>`, но не прибегая к приведению типов и другим операциям с непредсказуемым поведением. Это должно оказаться невозможным.

Послесловие

Так что же, программирование встроенных систем сводится, по существу, к "набивке битов"? Не совсем, особенно если вы преднамеренно стремитесь минимизировать работу с битами как потенциальный источник ошибок. Однако иногда избежать работы с битами и

байтами невозможно; вопрос только в том, где и как ее выполнять. В большинстве систем низкоуровневый код может и должен быть локализован. Многие из наиболее интересных систем, с которыми нам пришлось работать, были встроенными, и некоторые из наиболее интересных и сложных задач программирования возникают именно в этой предметной области.



26

Тестирование

Я только доказал корректность кода, но не тестировал его.

— Дональд Кнут (Donald Knuth)

В данной главе обсуждаются вопросы тестирования и проектирования для получения корректных программ. Это очень обширные темы, поэтому мы можем осветить их лишь поверхностно. Наша главная цель — описать некоторые практические идеи и методы тестирования модулей программы, таких как функции и классы. Мы обсудим использование интерфейсов и выбор тестов для проверки программ. Основной акцент будет сделан на проектировании и разработке систем, упрощающих тестирование и его применение на ранних этапах разработки. Бегло рассматриваются также вопросы доказательства корректности программ и решения проблем, связанных с производительностью.

26.1. Чего мы хотим

Проведем простой эксперимент. Напишите программу для бинарного поиска и выполните ее. Не ждите, пока дочитаете эту главу или раздел до конца. Важно, чтобы вы выполнили это задание немедленно! Бинарный поиск — это поиск в упорядоченной последовательности, который начинается с середины.

- Если средний элемент равен искомому, мы заканчиваем поиск.
- Если средний элемент меньше искомого, проводим бинарный поиск в правой части.
- Если средний элемент больше искомого, проводим бинарный поиск в левой части.
- Результат поиска является индикатором его успеха и позволяет модифицировать искомый элемент. Для этого в качестве такого индикатора используется индекс, указатель или итератор.

Используйте в качестве критерия сравнения (сортировки) оператор "меньше" (<). Можете выбрать любую структуру данных, любые способы вызова функций и способ возвращения результата, но обязательно напишите эту программу самостоятельно. Это редкий случай, когда использование функции, написанной кем-то другим, является контрпродуктивным, даже если эта функция написана хорошо. В частности, не используйте алгоритмы из стандартной библиотеки (`binary_search` или `equal_range`), которые в любой другой ситуации были бы наилучшим выбором. Можете затратить на разработку этой программы сколько угодно времени.

Итак, вы написали функцию для бинарного поиска. Если нет, то вернитесь к предыдущему абзацу! Если да — почему вы уверены, что ваша функция поиска корректна? Изложите свои аргументы, обосновывающие корректность программы. Вы уверены в своих аргументах? Нет ли слабых мест в вашей аргументации?

Это была тривиальная программа, реализующая очень простой и хорошо известный алгоритм. Исходный текст вашего компилятора занимает порядка 200 тысяч строк, исходный текст операционной системы — от 10 до 50 миллионов строк, а критичный с точки зрения безопасности код, обеспечивающий полет самолета, на котором вы отправитесь отдыхать во время ваших следующих каникул или на конференцию, составляет от 500 тысяч до 2 миллионов строк. Чувствуете ли вы себя после этого комфортно в кресле самолета? Как применить методы, которые вы использовали для проверки функции бинарного поиска, к реальному программному обеспечению, имеющему гораздо большие размеры?

Любопытно, что, несмотря на всю сложность, большую часть времени большая часть программного обеспечения работает правильно (мы не говорим сейчас о программах на переполненных игрушками персоналках). Что еще более важно, программное обеспечение с особыми требованиями к безопасности работает корректно практически всегда. Мы не можем припомнить примеры сбоев в программном обеспечении бортовых компьютеров авиалайнеров или автомобилей за последнее десятилетие. Рассказы о банковском программном обеспечении, вышедшем из строя из-за чека на нулевую сумму в настоящее время устарели; такие вещи больше не происходят. И все же программное обеспечение пишут такие же люди, как вы. Вы знаете, что делаете ошибки; но если мы можем делать ошибки, то почему следует думать, что "они" их не делают?

Чаще всего мы считаем, что знаем, как создать надежную систему из ненадежных частей. Мы тяжело работаем над каждой программой, каждым классом и каждой функцией, но, как правило, терпим неудачу при первом же испытании. Затем мы отлаживаем, тестируем и заново проектируем программу, устраняя в ней как можно больше ошибок. Однако в любой нетривиальной системе остается несколько скрытых ошибок. Мы знаем о них, но не можем найти — или, точнее, не можем найти их все за то время и с помощью тех усилий, которые можем затратить на эту проблему. После этого мы перепроектируем систему, чтобы уберечься от неожиданных и "невозможных" событий. В результате может получиться система, которая выглядит надежной. Заметим, что такие надежные системы могут по-прежнему скрывать ошибки (как правило, так и бывает) и работать меньше, чем ожидалось. Тем не менее они не выходят из строя окончательно и обеспечивают минимально приемлемый уровень обслуживания. Например, при исключительно большом количестве звонков телефонная система может не справляться с правильной обработкой каждого звонка, но никогда не отказывает окончательно.

Можно было бы пофилософствовать и подискутировать о том, следует ли считать ошибки в результате неожиданных событий действительно ошибками, но давайте не будем этого делать. Для разработчиков системы выгоднее "просто" выяснить, как сделать свои системы более надежными.

26.1.1. Предостережение

Тестирование — необъятная тема. Существует несколько точек зрения на то, как осуществлять тестирование, причем в разных прикладных областях имеются свои традиции и стандарты тестирования. И это естественно: нам не нужны одинаковые стандарты надежности для видеоигр и программного обеспечения для бортовых компьютеров авиалайнеров, но в итоге возникает путаница в терминах и избыточное разнообразие инструментов. Эту главу следует рассматривать как источник идей, касающихся тестирования как ваших персональных проектов, так и крупных систем. При тестировании больших систем используются настолько разнообразные комбинации инструментов и организационных структур, что описывать их здесь совершенно бессмысленно.

26.2. Доказательства

Постойте! Почему бы просто не доказать, что наши программы корректны, и не возиться с тестами? Как лаконично указал Эдсгер Дейкстра (Edsger Dijkstra), "Тестирование может выявить наличие ошибок, но не их отсутствие". Это приводит к очевидному желанию доказать корректность программ так, как математики доказывают теоремы.

К сожалению, доказательство корректности нетривиальных программ выходит за пределы современных возможностей (за исключением некоторых очень ограниченных прикладных областей), само доказательство может содержать ошибки (как и математические теоремы), и вся теория и практика доказательства корректности программ являются весьма сложными. Таким образом, мы пытаемся как можно сильнее структурировать свои программы с тем, чтобы к ним можно было применять рассуждения и убеждаться в их корректности. Однако мы также тестируем программы (раздел 26.3) и пытаемся организовать код так, чтобы он был устойчив к остающимся ошибкам (раздел 26.4).

26.3. Тестирование

В разделе 5.11 мы назвали тестирование "систематическим поиском ошибок". Рассмотрим методы такого поиска.

Различают *модульное тестирование* (unit testing) и *системное тестирование* (system testing). Модулем называется нечто наподобие функции или класса, являющееся частью полной программы. Если мы тестируем такие модули по отдельности, то знаем, где искать проблемы в случае обнаружения ошибок; все ошибки, которые мы можем обнаружить, находятся в проверяемом модуле (или в коде, который мы

используем для проведения тестирования). Этим модульное тестирование отличается от системного, в ходе которого тестируется полная система, и мы знаем лишь, что ошибка находится "где-то в системе". Как правило, ошибки, найденные при тестировании систем — при условии, что мы хорошо протестировали отдельные модули, — связаны с нежелательными взаимодействиями модулей. Найти ошибки в системе часто труднее, чем в модуле, причем на это затрачивается больше сил и времени.

Очевидно, что модуль (скажем, класс) может состоять из других модулей (например, функций или других классов), а системы (например, электронные коммерческие системы) могут состоять из других систем (например, баз данных, графического пользовательского интерфейса, сетевой системы и системы проверки заказов), поэтому различия между тестированием модулей и тестированием систем не так очевидны, как хотелось бы, но общая идея заключается в том, что при правильном тестировании мы экономим силы и нервы пользователей.

Один из подходов к тестированию основан на конструировании нетривиальных систем из модулей, которые, в свою очередь, сами состоят из более мелких модулей. Итак, мы начинаем тестирование с самых маленьких модулей, а затем тестируем модули, которые состоят из этих модулей, и так до тех пор, пока не приступим к тестированию всей системы. Иначе говоря, система при таком подходе рассматривается как самый большой модуль (если она не используется как часть более крупной системы).

Прежде всего рассмотрим, как тестируется модуль (например, функция, класс, иерархия классов или шаблон). Тестировщики различают тестирование белого (прозрачного) ящика (когда мы можем видеть детали реализации тестируемого модуля) и тестирование черного ящика (когда мы видим только интерфейс тестируемого модуля). Мы не будем глубоко вникать в различия между этими методами; в обоих вариантах читается тестируемый исходный код. Но помните, что позже кто-то может изменить эту реализацию, а потому не пытайтесь использовать информацию, которая не гарантируется интерфейсом. По существу, при любом виде тестирования основная идея заключается в исследовании реакции интерфейса на ввод информации.

Говоря, что кто-то (может быть, вы сами) может изменить код после того, как он будет протестирован, приводит нас к идее регрессивного тестирования. Как только вы вносите некоторое изменение, сразу же следует повторить тестирование, чтобы убедиться, что ничего не нарушилось. Так что при любых усовершенствованиях модуля следует повторить его тестирование и, перед тем как передать законченную систему кому-то еще (или перед тем, как использовать ее самому), необходимо выполнить тестирование полной системы. Выполнение такого полного тестирования системы часто называют *регрессивным тестированием* (regression testing), поскольку оно обычно включает выполнение

тестов, которые ранее уже выявили ошибки, чтобы убедиться, что они не возникли вновь. Если они возникли вновь, то программа регрессировала и ошибки следует устранять заново.

26.3.1. Регрессивные тесты

Создание крупной коллекции тестов, которые в прошлом оказались полезными для поиска ошибок, является основным способом конструирования эффективного тестового набора для системы. Предположим, у вас есть пользователи, которые будут сообщать вам о выявленных недостатках. Никогда не игнорируйте их отчеты об ошибках! В любом случае они свидетельствуют либо о наличии реальной ошибки в системе, либо о том, что пользователи имеют неправильное представление о системе. Об этом всегда полезно знать.

Как правило, отчет об ошибках содержит слишком много посторонней информации, и первой задачей при его обработке является создание как можно более короткой программы, которая выявляла бы указанную проблему. Для этого часто приходится отбрасывать большую часть представленного кода: в частности, мы обычно пытаемся исключить использование библиотек и прикладной код, который не влияет на ошибку. Поиск такой минимальной программы часто помогает локализовать ошибку в системном коде, и эту минимальную программу имеет смысл добавить в тестовый набор. Для того чтобы получить минимальную программу, следует удалять код до тех пор, пока не исчезнет сама ошибка, — в этот момент следует вернуть в программу последнюю исключенную часть кода. Эту процедуру следует продолжать до тех пор, пока не будут удалены все возможные фрагменты кода, не имеющие отношения к ошибке.

Простое выполнение сотен (или десятков тысяч) тестов, созданных на основе прошлых отчетов об ошибках, может выглядеть не очень систематизированным, но на самом деле в этом случае мы действительно целенаправленно используем коллективный опыт пользователей и разработчиков. Набор регрессивных тестов представляет собой главную часть коллективной памяти группы разработчиков. При разработке крупных систем мы просто не можем рассчитывать на постоянный контакт с авторами исходных кодов, чтобы они поясняли нам детали проектирования и реализации. Именно регрессивные тесты не позволяют системе отклоняться от линии поведения, согласованной с разработчиками и пользователями.

26.3.2. Модульные тесты

Однако довольно слов! Рассмотрим конкретный пример: протестируем программу бинарного поиска. Вот ее спецификация из стандарта ISO (раздел 25.3.3.4):

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first,
                   ForwardIterator last, const T& value );

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

Требования. Элементы `e` из диапазона [`first`,`last`) разделены в соответствии с отношением `e<value` и `!(value<e)` или `comp(e,value)` и `!comp(value,e)`. Кроме того, для всех элементов `e` диапазона [`first`, `last`) из условия `e<value` следует `!(value<e)`, а из условия `comp(e, value)` следует `!comp(value,e)`.

Возврат. Значение `true`, если в диапазоне [`first`,`last`) существует итератор `i`, удовлетворяющий условию `!(*i<value)&&!(value<*i)` или `comp(*i,value)==false && comp(value,*i)==false`.

Сложность. Не более `log(last-first)+2` сравнения.

Нельзя сказать, что непосвященному человеку легко читать эту формальную (ну хорошо, полуформальную) спецификацию. Однако, если вы действительно выполнили упражнение, посвященное проектированию и реализации бинарного поиска, которое мы настоятельно рекомендовали сделать в начале главы, то уже должны хорошо понимать, что происходит при бинарном поиске и как его тестировать. Данная (стандартная) версия функции для бинарного поиска получает в качестве аргументов пару однонаправленных итераторов (см. раздел 20.10.1) и определенное значение и возвращает значение `true`, если в диапазоне, определенном указанными итераторами, имеется это значение. Итераторы должны определять упорядоченную последовательность. Критерием сравнения (упорядочения) является оператор `<`. Вторую версию функции `binary_search`, в которой критерий сравнения задается как дополнительный аргумент, мы оставляем читателям в качестве упражнения.

Здесь мы столкнемся только с ошибками, которые не перехватывает компилятор, поэтому примеры, подобные этому, станут проблемой для кого-то другого.

```
// Ошибка: int — не однонаправленный итератор:
binary_search(1,4,5);

// Ошибка: невозможно найти строку в векторе целых чисел:
vector<int> v(10);
binary_search(v.begin(),v.end(),"7");

// Ошибка: забыли значение:
binary_search(v.begin(),v.end());
```

Как же *систематически* протестировать функцию `binary_search()`?

Очевидно, мы не можем просто перебрать все возможные аргументы, так как этими аргументами являются любые мыслимые последовательности значений любого возможного типа — количество таких тестов станет бесконечным! Итак, мы должны выбрать тесты и определить некие принципы этого выбора.

- Тест на *вероятные ошибки* (находит большинство ошибок).
- Тест на *опасные ошибки* (находит ошибки, имеющие наихудшие возможные последствия).

Под опасными мы подразумеваем ошибки, которые могут иметь самые ужасные последствия. В целом это понятие носит неопределенный характер, но для конкретных программ его можно уточнить. Например, если рассматривать бинарный поиск изолированно от других задач, то все ошибки могут быть одинаково опасными. Но, например, если мы используем функцию `binary_search` в программе, в которой ответы проверяются дважды, то получить неправильный ответ от функции `binary_search` может быть более приемлемым вариантом, чем не получить никакого из-за бесконечного цикла в функции. В таком случае, чтобы найти трюк с данными, провоцирующий бесконечный (или очень длинный) цикл в функции `binary_search`, нам пришлось бы затратить куда больше усилий по сравнению с исследованием вариантов, в которых она дает неправильный ответ. Отметьте в данном контексте слово "трюк". Помимо всего прочего, тестирование — это занятие, требующее изобретательного подхода к задаче "как заставить код работать неправильно". Наилучшие тестировщики не только методичные, но и изворотливые (конечно, в хорошем смысле этого слова) люди.

26.3.2.1. Стратегия тестирования

С чего же мы начинаем испытание функции `binary_search`? Мы смотрим на ее требования, т.е. на предположения о ее входных данных. К сожалению для тестировщиков, в требованиях явно указано, что диапазон [`first`,`last`) должен быть упорядоченной последовательностью. Другими словами, именно вызывающий модуль должен это гарантировать, поэтому мы не имеем права испытывать функцию `binary_search`, подавая на ее вход неупорядоченную последовательность или диапазон [`first`,`last`), в котором выполняется условие `last<first`. Обратите внимание на то, что в требованиях функции `binary_search` не указано, что она должна делать, если мы нарушим эти условия. В другом месте стандарта говорится, что в этих случаях функция может генерировать исключение, но не обязана это делать. Такие вещи следует твердо помнить во время тестирования функции `binary_search`, потому что, если вызывающий модуль нарушает требования функции, такой как `binary_search`, ошибки возникнут не по вине тестируемой функции.

Для функции `binary_search` можно себе представить следующие виды ошибок.

- Функция ничего не возвращает (например, из-за бесконечного цикла).
- Крах (например, неправильное разыменование, бесконечная рекурсия).
- Значение не найдено, несмотря на то что оно находится в указанной последовательности.
- Значение найдено, несмотря на то что оно не находится в указанной последовательности.

Кроме того, необходимо помнить о следующих "возможностях" для пользовательских ошибок.

- Последовательность не упорядочена (например, `{2,1,5,-7,2,10}`).
- Последовательность не корректна (например, `binary_search(&a[100], &a[50],77)`).

Какую ошибку (которую должны найти тестировщики) может допустить программист при простом вызове функции `binary_search(p1,p2,v)`? В основном ошибки возникают в частных случаях. Например, в ходе анализа последовательностей (любого вида) мы всегда ищем их начало и конец. Кроме того, всегда следует проверять, не пуста ли последовательность. Рассмотрим несколько массивов целых чисел, которые упорядочены так, как требуется.

```
{ 1,2,3,5,8,13,21 }              // "Обычная последовательность"
{ }                              // Пустая последовательность
{ 1 }                            // Только один элемент
{ 1,2,3,4 }                      // Четное количество элементов
{ 1,2,3,4,5 }                    // Нечетное количество элементов
{ 1, 1, 1, 1, 1, 1, 1 }          // Все элементы равны друг другу
{ 0,1,1,1,1,1,1,1,1,1,1,1,1 }    // Иной элемент в начале
{ 0,0,0,0,0,0,0,0,0,0,0,0,1 } // Иной элемент в конце
```

Некоторые тестовые последовательности лучше генерировать программно.

- Очень длинная последовательность:

```
vector<int> v1;
    for(int i=0; i<100000000; ++i) v.push_back(i);
```

- Последовательность со случайным количеством элементов.
- Последовательность со случайными (но упорядоченными) элементами.

И все же этот тест не настолько систематический, насколько нам бы хотелось. В конце концов, мы "просто выбрали" несколько последовательностей. Однако мы следовали некоторым общим правилам, которые часто полезны при работе с множествами значений; перечислим их.

- Пустое множество.
- Небольшие множества.

- Большие множества.
- Множества с экстремальными распределениями.
- Множества, в концах которых происходит "нечто интересное".
- Множества с дубликатами элементов.
- Множества с четным и нечетным количеством элементов.
- Множества, сгенерированные с помощью случайных чисел.

Мы используем случайные последовательности просто для того, чтобы увидеть, а не повезет ли нам найти неожиданную ошибку. Этот подход носит слишком прямолинейный характер, зато он относительно экономичен с точки зрения времени.

Почему мы рассматриваем четное и нечетное количество элементов? Дело в том, что многие алгоритмы разделяют входные последовательности на части, например на две половины, а программист может учесть только нечетное или только четное количество элементов. В общем случае, когда последовательность разделяется на части, то точка, в которой это происходит, становится концом подпоследовательности, а как известно, многие ошибки возникают в конце последовательностей.

В общем случае мы ищем следующее:

- экстремальные ситуации (большие или маленькие последовательности, странные распределения входных данных и т.п.);
- граничные условия (все, что происходит в окрестностях границ).

Реальный смысл этих понятий зависит от конкретной тестируемой программы.

26.3.2.2. Схема простого теста

Существуют две категории тестов: тесты, которые должны пройти успешно (например, поиск значения, которое есть в последовательности), и тесты, которые должны завершиться неудачей (например, поиск значения в пустой последовательности). Создадим для каждой из приведенных выше последовательностей несколько успешных и неудачных тестов. Начнем с простейшего и наиболее очевидного теста, а затем станем его постепенно уточнять, пока не дойдем до уровня, приемлемого для нашей функции `binary_search`.

```
vector<int> v { 1,2,3,5,8,13,21 };
if (binary_search(v.begin(),v.end(),1) == false) cout<<"отказ";
if (binary_search(v.begin(),v.end(),5) == false) cout<<"отказ";
if (binary_search(v.begin(),v.end(),8) == false) cout<<"отказ";
if (binary_search(v.begin(),v.end(),21)== false) cout<<"отказ";
if (binary_search(v.begin(),v.end(),-7)== true ) cout<<"отказ";
if (binary_search(v.begin(),v.end(),4) == true ) cout<<"отказ";
if (binary_search(v.begin(),v.end(),22)== true ) cout<<"отказ";
```

Это скучно и утомительно, но это всего лишь начало. На самом деле многие простые тесты — это не более чем длинные списки похожих вызовов. Положительной стороной такого наивного подхода является его чрезвычайная простота. Даже новичок в команде тестировщиков может добавить в этот набор свой вклад. Однако обычно мы поступаем лучше. Например, если в каком-то месте приведенного выше кода произойдет сбой, мы не сможем понять, где именно. Это неприемлемо. Написание тестов не прощает программирование методом "скопировать и вставить". Тестирующий код следует проектировать, как и любой другой. Поэтому перепишем наш код иначе:

```
vector<int> v { 1,2,3,5,8,13,21 };
for (int x : {1,5,8,21,-7,2,44})
    if (binary_search(v.begin(),v.end(),x) == false)
        cout << x << ": отказ";
```

Представьте себе десятки тестов и почувствуйте огромную разницу. При тестировании реальных систем часто приходится выполнять многие тысячи тестов, поэтому знать, какой из них закончился неудачей, очень важно.

Прежде чем идти дальше, отметим еще один пример (полуформальный) методики тестирования: мы тестировали правильные значения, иногда выбирая их из конца последовательности, а иногда из середины. Для данной последовательности мы можем перебрать все ее значения, но на практике сделать это нереально. Для тестов, ориентированных на провал, выбираем одно значение в каждом из концов последовательности и одно в середине. И снова следует отметить, что этот подход не является идеально систематическим, хотя и демонстрирует широко распространенный шаблон, которому можно следовать при работе с последовательностями или диапазонами значений.

Какими недостатками обладают указанные тесты?

- (Изначально) мы писали один и тот же код многократно.
- (Изначально) мы нумеровали тесты вручную.
- Вывод тестов минимален (не слишком полезен).

Поразмыслив немного, мы решили записать тесты в файл. Каждый тест должен иметь идентифицирующую метку, искомое значение, последовательность и ожидаемый результат, например такие:

```
{ 27 7 { 1 2 3 5 8 13 21} 0 }
```

Это тест под номером `27`. Он ищет число `7` в последовательности `{1,2,3,5,8,13,21}`, ожидая, что результатом будет `0` (т.е. `false`). Почему мы записали этот тест в файл, а не в текст программы? Ведь мы вполне могли бы написать этот тест прямо в исходном коде, но большое количество данных в тексте программы может ее запутать. Кроме того, тесты часто генерируются другими программами. Как правило, тесты, сгенерированные

программами, записываются в файлы данных. Кроме того, теперь мы можем написать тестовую программу, которую можно запускать с самыми разными тестовыми файлами.

```
struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

// Оператор >> использует описанный выше формат:
istream& operator>>(istream& is, Test& t);

int test_all(istream& is)
{
    int error_count = 0;
    for (Test t; is>>t; ) {
        bool r = binary_search(t.seq.begin(),
                               t.seq.end(), t.val);
        if (r !=t.res) {
            cout << "отказ: тест " << t.label
                 << " binary_search: "
                 << t.seq.size() << " элементов, val=="
                 << t.val << " -> " << t.res << '\n';
            ++error_count;
        }
    }
    return error_count;
}

int main()
{
    int errors = test_all(ifstream("my_tests.txt"));
    cout << "количество ошибок: " << errors << "\n";
}
```

Вот как выглядят некоторые тестовые данные.

```
{  1.1   1  {  1  2  3  5  8  13  21  }  1  }
{  1.2   5  {  1  2  3  5  8  13  21  }  1  }
{  1.3   8  {  1  2  3  5  8  13  21  }  1  }
{  1.4  21  {  1  2  3  5  8  13  21  }  1  }
{  1.5  -7  {  1  2  3  5  8  13  21  }  0  }
{  1.6   4  {  1  2  3  5  8  13  21  }  0  }
{  1.7  22  {  1  2  3  5  8  13  21  }  0  }
{  2     1  {  }  0  }
{  3.1   1  {  1  }  1  }
{  3.2   0  {  1  }  0  }
{  3.3   2  {  1  }  0  }
```

Здесь видно, почему мы использовали строковую метку, а не число: это позволяет более гибко нумеровать тесты с помощью десятичной точки,

обозначающей разные тесты для одной и той же последовательности. Более сложный формат записи тестов позволяет исключить необходимость повторения одной и той же тестовой последовательности в файле данных.

26.3.2.3. Случайные последовательности

Выбирая значения для тестирования, мы пытаемся перехитрить программистов, создавших реализацию функции (причем ими часто являемся мы сами), и использовать значения, которые могут выявить слабые места, скрывающие ошибки (например, сложные последовательности условий, концы последовательностей, циклы и т.п.). Однако то же самое мы делаем,

когда пытаемся написать и отладить свой код. Так что, проектируя тест, мы можем повторить логическую ошибку, сделанную при создании программы, и полностью пропустить проблему. Это одна из причин, по которым желательно, чтобы тесты проектировал не автор программы, а кто-то другой.

Существует один прием, который иногда помогает решить эту проблему: просто сгенерировать много случайных значений. Например, ниже приведена функция, которая записывает описание теста в поток **cout** с помощью функции **randint()** из раздела 24.7 и заголовочного файла **std_lib.facilities.h**.

```
void make_test(const string& lab, int n, int base, int spread)
    // Записывает описание теста с меткой lab в поток cout
    // Генерирует последовательность из n элементов, начиная
    // со значения base; среднее расстояние между элементами
    // имеет равномерное распределение на отрезке [0, spread]
{
    cout << "{ " << lab << " " << n << " { ";
    vector<int> v;
    int elem = base;
    for (int i = 0; i<n; ++i) {          // Создаем элементы
        elem += randint(spread);
        v.push_back(elem);
    }

    int val = base + randint(elem-base); // Искомое значение
    bool found = false;
    for(int i=0; i<n; ++i) {             // Вывод элементов
          if (v[i]==val) found = true;   // Есть ли элемент val
          cout << v[i] << " ";
    }
    cout << "} " << found << " }\n";
}
```

Заметим, что для проверки, имеется ли элемент **val** в случайной последовательности, мы не использовали функцию **binary_search**. Для того чтобы обеспечить корректность теста, мы не должны использовать функцию, которую проверяем!

На самом деле функция **binary_search** — не самый удобный пример для тестирования с помощью наивного подхода на основе случайных чисел. Мы сомневаемся, что таким образом сможем найти какие-то новые ошибки, пропущенные на ранних этапах с помощью тестов, разработанных "вручную", тем не менее этот метод довольно часто оказывается полезным. В любом случае следует выполнить несколько случайных тестов.

```
int no_of_tests = randint(100);  // Создаем порядка 50 тестов
for (int i = 0; i<no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab+to_string(i),  // to_string из раздела 23.2
              randint(500),      // Количество элементов
              0,                 // base
              randint(50));      // spread
}
```

Сгенерированные на основе случайных чисел тесты особенно полезны в ситуациях, когда необходимо протестировать кумулятивные эффекты многих операций, результат которых зависит от того, как были обработаны более ранние операции, т.е. от состояния системы (см. раздел 5.2).

Причина, по которой случайные числа не являются панацеей для тестирования функции **binary_search**, заключается в том, что результат любого поиска в последовательности не зависит от результатов других попыток поиска в этой последовательности (разумеется, в предположении что функция **binary_search** не содержит совершенно глупый код, например не модифицирует последовательность). Для этого случая у нас есть более хороший тест (упр. 5).

26.3.3. Алгоритмы и не алгоритмы

Мы рассмотрели функцию **binary_search()** в качестве примера. Это истинный алгоритм, обладающий следующими свойствами.

- Имеет точно определенные требования к входным данным.
- Есть точно определенные указания, что он может и чего не может делать с входными данными (в данном случае — не изменяет эти данные).
- Не зависит от объектов, которые не являются явными входными данными.
- На окружение алгоритма не наложено никаких серьезных ограничений (например, не указаны предельное время работы и объем памяти или иных ресурсов, имеющихся в его распоряжении).

У алгоритма бинарного поиска есть очевидные и открыто сформулированные пред- и постусловия (см. раздел 5.10). Другими словами, этот алгоритм — просто мечта тестировщика. Часто нам не так сильно везет и приходится тестировать плохой код, сопровождаемый (в лучшем случае) небрежным описанием и парой диаграмм.

Погодите! А не впадаем ли мы в заблуждение? Как можно говорить о корректности и тестировании, если у нас нет точного описания, что именно должен делать сам код? Проблема заключается в том, что многое из того, что должно делать программное обеспечение, нелегко выразить с помощью точных математических терминов. Кроме того, во многих случаях, даже когда это теоретически возможно, необходимая математика лежит за пределами возможностей программиста, который пишет и тестирует данный код. Поэтому мы должны расстаться с идеальными представлениями о совершенно точных спецификациях и смириться с реальностью, в которой существуют не зависящие от нас условия и спешка.

А теперь представим себе плохую функцию, которую нам требуется протестировать. Под плохой функцией мы понимаем следующее.

- *Входные данные.* Требования к входным данным (явные или неявные) сформулированы не так четко, как нам хотелось бы.
- *Выходные данные.* Результаты (явные или неявные) сформулированы не так четко, как нам хотелось бы.
- *Ресурсы.* Условия использования ресурсов (время, память, файлы и пр.) сформулированы не так четко, как нам хотелось бы.

Под явным или неявным мы подразумеваем, что следует проверять не только формальные параметры и возвращаемое значение, но и влияние глобальных переменных, потоков ввода-вывода, файлы, распределения динамической памяти и т.д. Что же мы можем сделать? Прежде всего, такая функция практически всегда очень длинная, иначе ее требования и результаты можно было бы описать более точно. Возможно, речь идет о функции длиной около пяти страниц или функции, использующей вспомогательные функции сложным и неочевидным способом. Для функции пять страниц — это много; тем не менее мы видели функции гораздо, гораздо более длинные. К сожалению, они не так уж редки.

Если вы проверяете свой код и у вас есть время, прежде всего попробуйте разделить такую плохую функцию на функции меньшего размера, каждая из которых будет ближе к идеалу функции с точной спецификацией, и в первую очередь протестируйте их. Однако в данный момент мы будем предполагать, что наша цель — тестирование программного обеспечения, т.е. систематический поиск как можно большего количества ошибок, а не простое исправление выявленных дефектов.

Итак, что же мы ищем? Наша задача как тестировщиков — искать ошибки. Где они обычно скрываются? Чем отличаются программы, которые чаще всего содержат ошибки?

- Тонкие зависимости от другого кода. Ищите использование глобальных переменных, аргументы, которые передаются не с помощью константных ссылок, указатели и т.п.

- Управление ресурсами. Обратите внимание на управление памятью (операторы **new** и **delete**), использование файлов, блокировки и т.п.
- Поищите циклы. Проверьте условия выхода из них (как в функции **binary_search()**).
- Инструкции **if** и **switch** (которые часто называют инструкциями ветвления). Ищите ошибки в их логике.

Рассмотрим примеры, иллюстрирующие каждый из перечисленных пунктов.

26.3.3.1. Зависимости

Рассмотрим следующую бессмысленную функцию.

```
int do_dependent(int a, int& b) // Плохая функция
{                               // Неорганизованные зависимости
    int val ;
    cin>>val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

Для тестирования функции **do_dependent()** мы должны не просто синтезировать набор аргументов и посмотреть, что она с ними будет делать. Мы должны учесть, что эта функция использует глобальные переменные **cin**, **cout** и **vec**. Это обстоятельство вполне очевидно в данной небольшой и бессмысленной программе, но в более крупном коде оно может быть скрыто. К счастью, существует программное обеспечение, помогающее находить такие зависимости. К сожалению, оно не всегда легкодоступно и довольно редко используется. Допустим, у нас нет программного обеспечения для анализа кода, и мы вынуждены строка за строкой просматривать функцию в поисках ее зависимостей.

Для того чтобы протестировать функцию **do_dependent()**, мы должны рассмотреть

- входные данные функции
  - значение переменной **a**
  - значения переменной **b** и переменной типа **int**, на которую ссылается переменная **b**
  - ввод из потока **cin** (в переменную **val**) и состояние потока **cin**
  - состояние потока **cout**
  - значение переменной **vec**, в частности — значение **vec[val]**
- выходные данные функции

- возвращаемое значение
- значение переменной типа **int**, на которую ссылается переменная **b** (выполнен ее инкремент)
- состояние объекта **cin** (проверьте состояния потока и форматирования)
- состояние объекта **cout** (проверьте состояния потока и форматирования)
- состояние массива **vec** (мы присвоили значение элементу **vec[val]**)
- Любые исключения, которые мог сгенерировать массив **vec** (ячейка **vec[val]** может находиться за пределами допустимого диапазона).

Это длинный список. Фактически он длиннее, чем сама функция. Он отражает наше неприятие глобальных переменных и беспокойство о неконстантных ссылках (и указателях). Все-таки в функциях, которые просто считывают свои аргументы и выводят возвращаемое значение, есть своя прелесть: их легко понять и протестировать.

Как только мы идентифицировали входные и выходные данные, мы тут же оказываемся в ситуации, в которой уже побывали, тестируя **binary_search()**. Мы просто генерируем тесты с входными значениями (для явного и неявного ввода), чтобы увидеть, приводят ли они к желаемым результатам (явным и неявным). Тестируя функцию **do_dependent()**, мы могли бы начать с очень большого значения переменной **val** и отрицательного значения переменной **val**, чтобы увидеть, что произойдет. Было бы лучше, если бы массив **vec** оказался вектором, предусматривающим проверку диапазона (иначе мы можем очень просто сгенерировать действительно опасные ошибки). Конечно, мы могли бы поинтересоваться, что сказано об этом в документации, но плохие функции, подобные рассматриваемой, редко сопровождаются полной и точной спецификацией, поэтому мы просто "сломаем" эту функцию (т.е. найдем ошибки) и начнем задавать вопросы о ее корректности. Часто такое сочетание тестирования и вопросов приводит к полному перепроектированию функции.

26.3.3.2. Управление ресурсами

Рассмотрим следующую бессмысленную функцию.

```
void do_resources1(int a,int b, // Плохая функция
                   const char* s) // Неверное использование
{                                 // ресурсов
    FILE* f = fopen(s,"r");       // Открываем файл (стиль C)
    int* p = new int[a];          // Выделяем память
    if (b<=0) throw Bad_arg();    // Может генерировать исключение
    int* q = new int[b];          // Выделяем еще немного памяти
    delete[] p;                   // освобождаем память, на которую указывает p
}
```

Для того чтобы протестировать функцию `do_resources1()`, мы должны рассмотреть, правильно ли распределены ресурсы, т.е. освобожден ли выделенный ресурс или передан другой функции.

Перечислим очевидные недостатки.

- Файл с именем `s` не закрыт.
- Память, выделенная для указателя `p`, не освобождается, если `b <= 0` или если второй оператор `new` генерирует исключение.
- Утечка памяти, выделенной для `q`, при `b > 0`.

Кроме того, мы всегда должны рассматривать возможность того, что попытка открыть файл закончится неудачей. Для того чтобы получить этот грустный результат, мы намеренно использовали устаревший стиль программирования (функция `fopen()` — это стандартный способ открытия файла в языке C). Мы могли бы упростить работу тестировщиков, если бы просто написали следующий код:

```
void do_resources2(int a,int b, // Менее плохой код
                   const char* s)
{
    ifstream is(s);              // Открываем файл
    vector<int>v1(a);            // Создаем вектор (владение памятью)
    if (b<=0) throw Bad_arg();   // МОЖЕТ генерировать исключение
    vector<int> v2(b);           // Создаем другой вектор
                                 // (владение памятью)
}
```

Теперь каждый ресурс принадлежит объекту с деструктором, который его и освобождает. Иногда, чтобы выработать идеи для тестирования, полезно попытаться сделать функцию более простой и ясной. Общую стратегию решения задач управления ресурсами обеспечивает идиома RAII, описанная в разделе 19.5.2.

Обратите внимание, что управление ресурсами не сводится к простой проверке, освобожден ли каждый выделенный фрагмент памяти. Иногда мы получаем ресурсы извне (например, как аргумент), а иногда сами передаем их какой-нибудь функции (как возвращаемое значение). В этих ситуациях довольно трудно понять, правильно ли выполняется работа с ресурсами. Рассмотрим пример.

```
FILE* do_resources3(int a, int* p, // Плохая функция
                    const char* s)
    // Неправильная передача ресурса
{
    FILE* f = fopen(s,"r");
    delete p;
    delete var;
    var = new int[27];
    return f;
}
```

Правильно ли, что функция `do_resources3()` передает (предположительно) открытый файл обратно как возвращаемое значение? Правильно ли, что функция `do_resources3()` освобождает память, передаваемую ей как аргумент `p`? Мы также добавили действительно коварный вариант использования глобальной переменной `var` (очевидно, указателя). В принципе передача ресурсов в функцию и из нее является довольно распространенной и полезной практикой, но для того чтобы понять, все ли корректно, необходимо знать стратегию управления ресурсами. Кто владеет ресурсом? Кто должен его удалять/освобождать? Документация должна ясно и четко отвечать на эти вопросы. (Размечтались...) В любом случае передача ресурсов изобилует возможностями для ошибок и представляет сложность для тестирования.

Обратите внимание на то, что мы (преднамеренно) усложнили пример управления ресурсами, использовав глобальную переменную. Если в программе перемешано несколько источников ошибок, ситуация может резко ухудшиться. Как программисты мы стараемся избегать таких ситуаций, как тестировщики — стремимся найти их.

26.3.3.3. Циклы

Мы уже рассматривали циклы, когда обсуждали функцию `binary_search()`. Большинство ошибок возникает в конце циклов.

- Правильно ли проинициализированы переменные в начале цикла?
- Правильно ли заканчивается цикл (часто на последнем элементе)?

Приведем пример, который содержит ошибку.

```
int do_loop(const vector<int>& v) // Плохая функция
    // Неправильный цикл
{
     int i;
     int sum;
     while(i<=vec.size()) sum+=v[i];
     return sum;
}
```

Здесь содержатся три очевидные ошибки. (Какие именно?) Кроме того, хороший тестировщик немедленно выявит возможности для переполнения при добавлении чисел к переменной `sum`.

- Многие циклы связаны с данными и могут вызвать переполнение при работе с большими числами.

Широко известная и особенно опасная ошибка, связанная с циклами и заключающаяся в переполнении буфера, относится к категории ошибок, которые можно систематически отлавливать, задавая два ключевых вопроса о циклах.

```
char buf[MAX];      // Буфер фиксированного размера
char* read_line()   // Опасная функция
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch!='\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}
```

Разумеется, уж вы бы не написали ничего подобного! (А почему нет? Что плохого в функции `read_line()`?) Однако эта ошибка, к сожалению, является довольно распространенной и имеет разные варианты.

```
// Опасный фрагмент
gets(buf);          // Считываем строку в переменную buf
scanf("%s",buf);    // Считываем строку в переменную buf
```

Поищите описание функций `gets()` и `scanf()` в своей документации и избегайте их как чумы. Под словом "опасная" мы подразумеваем, что переполнение буфера является инструментом для взлома компьютеров. В настоящее время многие реализации C++ выдают предупреждение об опасности использования функции `gets()` и ее аналогов именно по этой причине.

26.3.3.4. Ветвление

Очевидно, что, делая выбор, мы можем принять неправильное решение. Из-за этого инструкции `if` и `switch` являются одними из основных целей для тестировщиков. Существуют две проблемы, которые необходимо исследовать.

- Все ли возможные варианты предусмотрены?
- Правильные ли действия связаны с правильными вариантами выбора?

Рассмотрим следующую бессмысленную функцию:

```
void do_branch1(int x, int y) // Плохая функция
    // Неправильное использование инструкции if
{
    if (x<0) {
        if (y<0)
            cout << "очень отрицательное\n";
        else
            cout << "немного отрицательное\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "очень положительное\n";
        else
```

```
            cout << "немного положительное\n";
    }
}
```

Наиболее очевидная ошибка в этом фрагменте заключается в том, что мы забыли о варианте, в котором переменная **x** равна нулю. Сравнивая числа (положительные или отрицательные) с нулем, программисты часто забывают о нем или приписывают неправильной ветви (например, относят его к отрицательным числам). Кроме того, существует более тонкая (хотя и распространенная) ошибка, скрытая в этом фрагменте: действия при условиях **(x>0 && y<0)** и **(x>0 && y>=0)** каким-то образом поменялись местами. Это часто случается, когда программисты используют метод "копировать и вставить".

Чем более сложными являются варианты использования инструкций **if**, тем вероятнее становятся ошибки. Тестировщики анализируют такие коды и стараются не пропустить ни одной ветви. Для функции **do_branch1()** очевидный набор тестов выглядит следующим образом.

```
do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1( 1,-1);
do_branch1( 1, 1);
do_branch1(-1, 0);
do_branch1( 0,-1);
do_branch1( 1, 0);
do_branch1( 0, 1);
do_branch1( 0, 0);
```

По сути, это прямолинейный подход "перебора всех альтернатив", которой мы применили, заметив, что функция **do_branch1()** сравнивает значения с нулем с помощью операторов **<** и **>**. Для того чтобы выявить неправильные действия при положительных значениях переменной **x**, мы должны объединить вызовы функции с желаемыми результатами.

Обработка инструкций **switch** по сути идентична обработке инструкций **if**.

```
void do_branch1(int x, int y)     // Плохая функция
    // Неправильное использование инструкции switch
{
    if (y<0 && y<=3)
        switch (x) {
        case 1:
            cout << "один\n";
            break;
        case 2:
            cout << "два\n";
        case 3:
            cout << "три\n";
    }
}
```

Здесь сделаны четыре классические ошибки.

- Мы проверяем диапазон значений неправильной переменной (`y`, а не `x`).
- Мы забыли об инструкции `break`, что приводит к неправильному действию при `x==2`.
- Мы забыли о разделе `default` (считая, что он предусмотрен инструкцией `if`).
- Мы написали `y<0`, хотя имели в виду `0<y`.

Как тестировщики мы всегда ищем непредвиденные варианты. Пожалуйста, помните, что просто устранить проблему недостаточно. Она может возникнуть снова, когда мы ее не ожидаем. Мы хотим писать тесты, которые выявляют ошибки систематически. Если мы просто исправим этот простой код, то можем либо неправильно решить задачу, либо внести новые и отличные от исправленных ошибки. Цель анализа кода заключается не только в выявлении ошибок (хотя это всегда полезно), но и в разработке подходящего набора тестов, позволяющих выявить все ошибки (или, говоря более реалистично, большинство из них).

Заметим, что циклы всегда содержат неявные инструкции `if`: они выполняют проверку условия выхода из цикла. Таким образом, циклы также являются инструкциями ветвления. Когда мы анализируем программы, содержащие инструкции ветвления, первым возникает следующий вопрос: "Все ли ветви мы проверили?" Удивительно, но в реальной программе это не всегда возможно (потому что в реальном коде функции вызываются так, как нужно другим функциям, и не всегда всеми возможными способами). В результате возникает следующий вопрос: "Какую часть кода мы проверили?" И в лучшем случае мы можем ответить "Мы проверили большинство ветвей", поясняя, почему достичь остальных ветвей оказалось слишком трудно. 100%-ный охват — это идеал.

26.3.4. Системные тесты

Тестирование любой более или менее значимой системы — сложная работа для опытных тестировщиков. Например, тестирование компьютеров, управляющих телефонной системой, проводится в специально оборудованных комнатах с полками, заполненными компьютерами, имитирующими трафик десятков тысяч людей. Такие системы стоят миллионы долларов и являются результатом работы коллективов очень опытных инженеров. Предполагается, что после их развертывания основные телефонные коммутаторы будут непрерывно работать двадцать лет, а общее время их простоя составит не более двадцати минут (по любым причинам, включая исчезновение энергопитания, наводнения и землетрясения). Мы не будем углубляться в детали — легче научить новичка, не знающего физики, вычислить поправки к курсу космического аппарата, спускающегося на

поверхность Марса, — но попытаемся изложить идеи, которые могут оказаться полезными при тестировании менее крупных проектов или для понимания принципов тестирования более крупных систем.

Прежде всего следует вспомнить, что целью тестирования является выявление ошибок, особенно часто встречающихся и потенциально опасных. Написать и выполнить большое количество тестов непросто. Отсюда следует, что для тестировщика крайне желательно понимать сущность тестируемой системы. Для эффективного тестирования систем знание прикладной области еще важнее, чем для тестирования отдельных модулей. Для разработки системы необходимо знать не только язык программирования и компьютерные науки, но и прикладную область, а также людей, которые будут использовать приложение. Это является одной из наших мотиваций для работы с программами: вы увидите много интересных приложений и встретите много интересных людей.

Для того чтобы протестировать полную систему, необходимо создать все ее составные части (модули). Это может потребовать значительного времени, поэтому многие системные тесты выполняются только один раз в сутки (часто ночью, когда предполагается, что разработчики спят) после тестирования всех модулей по отдельности. В этом процессе ключевую роль играют регрессивные тесты. Самой подозрительной частью программы, в которой, вероятнее всего, кроются ошибки, является новый код и те области кода, в которых ранее обнаруживались ошибки. По этой причине существенным является выполнение набора старых тестов (регрессивное тестирование); без этого крупная система никогда не будет стабильной. Мы можем вносить новые ошибки с той же скоростью, с которой удаляются старые.

Обратите внимание на то, что мы полагаем неизбежным случайное внесение новых ошибок при исправлении старых. Мы надеемся, что вновь добавленных ошибок меньше, чем удаленных старых, причем последствия новых ошибок менее серьезны. Однако, по крайней мере пока не будут выполнены регрессивные тесты, а для нового кода не будут добавлены новые тесты, мы должны исходить из предположения, что наша система неработоспособна (из-за новых ошибок, внесенных в ходе исправления старых).

26.3.5. Поиск предположений, которые не выполняются

Спецификация `binary_search` ясно указывает на то, что последовательность, в которой выполняется поиск, должна быть упорядочена. Это лишает нас возможности создавать множество изощренных модульных тестов. Но очевидно, что существуют возможности написать неправильный код, для которого мы не сможем написать находящий ошибки тест (за

исключением системных тестов). Можем ли мы использовать свое понимание системных модулей (функций, классов и т.п.) для разработки более надежных тестов?

К сожалению, простейший ответ — нет. Поскольку мы являемся только тестировщиками, мы не можем изменять код, а для того, чтобы выявить нарушение требований интерфейса (предусловий), надо либо проверять их перед каждым вызовом, либо сделать частью реализации каждого вызова (см. раздел 5.5). Однако, тестируя собственный

код, мы можем добавлять такие тесты. Если мы тестировщики, но люди, создающие код, к нам прислушиваются (что бывает не всегда), то мы можем сообщить им о непроверяемых требованиях и убедить их вставить в код такие проверки.

Рассмотрим функцию **binary_search** еще раз: мы не можем проверить, что входная последовательность [**first**,**last**) действительно является последовательностью и что она упорядочена (см. раздел 26.3.2.2). Однако можно написать функцию, которая выполняет такую проверку.

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // Проверяем, является ли диапазон [first,last)
    // последовательностью:
    if (last<first) throw Bad_sequence();

    // Проверяем упорядоченность последовательности:
    if (2<=last- first)
        for (Iter p = first+1; p<last; ++p)
            if (*p<*(p-1)) throw Not_ordered();

     // Все в порядке, вызываем binary_search:
     return binary_search(first,last,value);
}
```

Перечислим причины, по которым функция **binary_search** не содержала таких проверок.

- Условие **last < first** нельзя проверить для однонаправленного итератора; например, итератор контейнера **std::list** не имеет оператора **<** (раздел Б.3.2). В общем случае хорошего способа проверки того, что пара итераторов определяет последовательность, не существует (начинать перемещение с итератора **first**, надеясь достигнуть итератора **last**, — не самая хорошая идея).
- Просмотр последовательности для проверки того, что ее значения упорядочены, является более затратным, чем выполнение самой функции **binary_search** (действительная цель выполнения функции **binary_search** заключается в том, чтобы избежать слепого прохода по последовательности в поисках значения, как это делает функция **std::find**).

Что же мы можем сделать? Мы могли бы при тестировании заменить функцию **binary_search** функцией **b2** (впрочем, только для вызовов функции **binary_search** с помощью итераторов произвольного доступа). В качестве альтернативы можно заставить разработчика функции **binary_search** изменить ее код для возможного тестирования.

```
template<class Iter, class T>
// Предупреждение: содержит псевдокод
bool binary_search (Iter first, Iter last, const T& value)
{
    if (тест_включен) {
        if (Iter является итератором произвольного доступа) {
            // Проверяем, является ли [first,last)
            // последовательностью:
            if (last<first) throw Bad_sequence();
        }

        // Проверяем, является ли последовательность
        // упорядоченной:
        if (first!=last) {
            Iter prev = first;
            for (Iter p = ++first; p!=last; ++p, ++ prev)
                if (*p<*prev) throw Not_ordered();
        }
    }

    // Выполняем функцию binary_search
}
```

Поскольку смысл условия **тест_включен** зависит от способа организации тестирования (для конкретной системы в конкретной организации), мы оставим его в виде псевдокода: тестируя собственный код, вы можете, например, просто использовать переменную **test_enabled**. Мы также оставили условие **Iter является итератором произвольного доступа** псевдокодом, поскольку не хотели пояснять здесь свойства итераторов. Если вам действительно необходима такая проверка, обратитесь к теме *свойства итераторов* (iterator traits) в более подробном учебнике по языку C++.

26.4. Проектирование с учетом тестирования

Приступая к написанию программы, мы знаем, что в итоге она должна быть полной и правильной. Мы также знаем, что для этого ее будет необходимо тестировать. Поэтому, разрабатывая программу, мы должны учитывать возможности ее тестирования с первого дня. Многие хорошие программисты руководствуются девизом "Тестируй заблаговременно и часто" и не пишут программу, если не представляют себе, как ее тестировать. Размышление о тестировании на ранних этапах раз-

работки программы позволяет избежать ошибок (и помогает найти их позднее). Мы разделяем эту точку зрения. Некоторые программисты даже пишут тесты для модулей еще до реализации самих модулей.

Примеры из разделов 26.3.2.1 и 26.3.3 иллюстрируют эти важные положения.

- Пишите точно определенные интерфейсы с тем, чтобы вы могли написать для них тесты.
- Придумайте способ описать операции в виде текста, чтобы их можно было хранить, анализировать и воспроизводить. Это относится также к операциям вывода.
- Встраивайте тесты для непроверяемых предположений (assertions) в вызывающий код, чтобы перехватить неправильные аргументы до системного тестирования.
- Минимизируйте зависимости и делайте их явными.
- Придерживайтесь ясной стратегии управления ресурсами.

С философской точки зрения это можно рассматривать как применение методов модульного тестирования для проверки подсистем и полных систем.

Если производительность работы программы не имеет большого значения, то в ней проверку предположений (требований, предусловий), которые в противном случае остались бы непроверяемыми, можно оставить навсегда. Однако существуют причины, по которым это не делается систематически. Например, мы уже говорили, что проверка упорядоченности последовательности сложна и связана с гораздо большими затратами, чем сама функция `binary_sort`. Следовательно, целесообразно разработать систему, позволяющую избирательно включать и выключать такие проверки. Для многих систем удобно оставить определенное количество простых и недорогостоящих (т.е. мало влияющих на производительность) проверок в окончательной версии, поставляемой пользователям: иногда происходят даже невероятные события, и лучше узнать об этом из конкретного сообщения об ошибке, чем в результате сбоя программы.

26.5. Отладка

Отладка — это вопрос методики и вкуса, причем вкус играет не последнюю роль. Пожалуйста, перечитайте еще раз главу 5. Обратите внимание на то, чем отладка отличается от тестирования. В ходе обоих процессов выявляются ошибки, но при отладке это происходит не систематически и, как правило, связано с удалением известных ошибок и реализацией определенных свойств. Должно быть сделано все возможное, чтобы отладка больше походила на тестирование. С сильной натяжкой можно сказать, что мы неплохо относимся к тестированию, но что

касается отладки — ее мы ненавидим. Хорошее тестирование модулей на ранних этапах их разработки и проектирование с учетом тестирования помогает минимизировать отладку.

26.6. Производительность

Для того чтобы программа оказалась полезной, мало, чтобы она была правильной. Даже если предположить, что она имеет достаточную функциональность, чтобы быть полезной, она должна еще обеспечивать приемлемый уровень производительности. Хорошая программа является достаточно эффективной; иначе говоря, она выполняется за приемлемое время при данных ресурсах. Абсолютная эффективность никого не интересует, и стремление сделать программу как можно более быстродействующей за счет усложнения кода может серьезно повредить всей системе (из-за большего количества ошибок и большего объема отладки), повысив к тому же сложность и дороговизну ее эксплуатации и сопровождения (включая перенос на другие компьютеры и настройку производительности ее работы).

Как же узнать, что программа (или ее модуль) является "достаточно эффективной"? Абстрактно на этот вопрос ответить невозможно. Современное аппаратное обеспечение работает настолько быстро, что для многих программ этот вопрос вообще не возникает. Нам встречались программы, намеренно скомпилированные в режиме отладки (т.е. работающие раз в 25 медленнее, чем требуется), чтобы обеспечить лучшие возможности диагностики ошибок, которые могут возникнуть после развертывания (это может произойти даже с самым лучшим кодом, который вынужден сосуществовать с другими программами, разработанными "где-то еще").

Следовательно, ответ на вопрос "Достаточно ли эффективной является программа?" звучит так: "Измерьте время, за которое выполняется интересующий вас тест". Очевидно, что для этого необходимо очень хорошо знать своих конечных пользователей и иметь представление о том, что именно их интересует и какую продолжительность работы они считают приемлемой. Логически мы просто засекаем время выполнения тестов с помощью секундомера и проверяем, не работали ли они дольше разумного предела. Определить продолжительность выполнения тестов можно с помощью функции `system_clock()` (раздел 26.6.1), так что можно автоматически сравнивать продолжительность выполнения тестов с оценками, которые признаны разумными. В качестве альтернативы (или в дополнение) можно записывать продолжительность выполнения тестов и сравнивать их с ранее полученными результатами. Таким образом мы получаем разновидность регрессивного тестирования производительности.

Обычно плохие показатели производительности обусловлены неудачным выбором алгоритмов и могут быть обнаружены при тестировании. Одна из причин применения больших наборов данных для тестирования программ заключается в выявлении неэффективных алгоритмов. В качестве примера предположим, что приложение должно суммировать элементы, находящиеся в строках матрицы (используя класс **Matrix** из главы 24). Некто предложил для этого функцию

```
// Суммирование элементов из m[n]
double row_sum(Matrix<double,2> m, int n);
```

Затем этот некто стал использовать эту функцию для того, чтобы сгенерировать вектор сумм, где **v[n]** — сумма элементов в первых **n** строках.

```
// Сумма элементов в m[0,n)
double row_accum(Matrix<double,2> m, int n)
{
    double s = 0;
    for (int i=0; i<n; ++i) s+=row_sum(m,i);
    return s;
}


// Вычисление накопленных сумм по строкам матрицы m:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

Представьте себе, что этот код является частью модульного теста или выполняется как часть системного теста. В любом случае вы заметите нечто странное, если матрица станет действительно большой: оказывается, что время, необходимое для выполнения программы, по сути квадратично зависит от размера матрицы **m**. Почему? Дело в том, что мы просуммировали и внесли в вектор сумму всех элементов первой строки; затем, чтобы найти сумму элементов двух строк, мы снова суммируем все элементы из первой строки, а затем — все элементы из второй; после мы вновь суммируем все элементы из первых двух строк и добавляем к ним элементы третьей строки и т.д.

Если вы считаете этот пример неудачным, посмотрите, что произойдет, если функция **row_sum()** будет обращаться за значениями элементов к базе данных. Чтение данных с диска происходит во много тысяч раз медленнее, чем чтение из оперативной памяти.

Вы можете возразить: "Никто никогда не сделает нечто настолько глупое!" Извините, но мы видели вещи и похуже, и, как правило, плохой (с точки зрения производительности) алгоритм очень нелегко выявить, если он глубоко скрыт в коде приложения. Заметили ли вы проблемы с производительностью при первом взгляде на этот код? Проблему бывает трудно

выявить, если не искать ее целенаправленно. Рассмотрим простой реальный пример, найденный на одном сервере.

```
for(int i=0; i<strlen(s); ++i){/* Что-то делаем с s[i] */}
```

Зачастую `s` представляет собой строку размером около 20 Кбайт.

Не все проблемы, связанные с производительностью программы, объясняются плохим алгоритмом. Фактически (как мы указывали в разделе 26.3.3) большую часть кода, который мы пишем, нельзя квалифицировать как плохой алгоритм. Такие "неалгоритмические" проблемы обычно связаны с неправильным проектированием. Перечислим некоторые из них.

- Повторяющееся вычисление информации заново (как, например, в приведенном выше примере).
- Повторяющаяся проверка одного и того же факта (например, проверка того, что индекс не выходит за пределы допустимого диапазона при каждом его использовании в цикле, или повторяющаяся проверка аргумента, который передается от функции к функции без каких-либо изменений).
- Повторяющиеся обращения к диску (или к сети).

Обратите внимание на слово "*повторяющиеся*". Очевидно, что мы имеем в виду "напрасно повторяющееся", но главное в том, что пока мы не выполним некоторое действие очень много раз, мы не увидим его влияния на производительность программы. Мы являемся горячими сторонниками строгой проверки аргументов функций и переменных циклов, но если мы миллионы раз проверяем одну и ту же переменную, то такие излишние проверки могут нанести ущерб производительности программы. Если — в результате измерений! — выяснится, что производительность упала, мы должны изыскать возможность удалить повторяющиеся действия. Но не делайте этого, пока не убедитесь, что производительность программы действительно стала неприемлемо низкой. Дело в том, что преждевременная оптимизация часто является источником многих ошибок и отнимает много времени.

26.6.1. Измерение времени

Как понять, достаточно ли быстро работает фрагмент кода? Как узнать, насколько быстро работает данная операция? Во многих ситуациях, связанных с измерением времени, можете просто посмотреть на часы (секундомер, стенные или наручные часы). Это не научно и не точно, но если это оказалось невыполнимым (программа завершилась слишком быстро), то можно считать, что программа работает с достаточной скоростью. Но

такой подход неприемлем для тех, кого беспокоят реальные вопросы производительности программ.

Если вам необходимо измерять более мелкие интервалы времени или вы не хотите сидеть с секундомером, вам следует научиться использовать возможности компьютера, знающего, как измерять время. Например, в операционной системе Unix достаточно просто поставить перед командой слово **time**, чтобы система вывела продолжительность ее выполнения. Можете, например, использовать команду **time**, чтобы выяснить, сколько времени заняла компиляция исходного файла **x.cpp**. Обычно компиляция выполняется командой

```
g++ x.cpp
```

Для того чтобы измерить продолжительность компиляции, поставьте перед ней слово **time**.

```
time g++ x.cpp
```

После компиляции файла **x.cpp** на экран будет выведено затраченное время. Это простой и эффективный способ измерения продолжительности работы небольших программ. Не забудьте выполнить измерения несколько раз, потому что на продолжительность выполнения программы могут влиять другие действия, выполняемые на вашем компьютере. Если вы получите, скажем, три одинаковых ответа, то обычно такому результату можно доверять.

Но что если вы хотите измерить интервал времени, длящийся всего несколько миллисекунд? Что если вы хотите выполнить собственные, более подробные измерения, связанные с работой части вашей программы? Продемонстрируем использование функции **clock()** из стандартной библиотеки, позволяющей измерить продолжительность выполнения функции **do_something()**.

```
#include <chrono>
#include <iostream>
using namespace std;

int main()
{
    int n = 10000000;              // Повтор do_something() n раз
    auto t1 = system_clock::now();// Начало отсчета
    for (int i = 0; i<n; i++)
        do_something();            // Цикл измерений
    auto t2 = system_clock::now();// Конец отсчета
    cout<< "Выполнение do_something() " << n << " раз заняло "
        << duration_cast<milliseconds>(t2-t1).count() << "мс\n";
}
```

`system_clock` — один из стандартных таймеров, а `system_clock::now()` возвращает момент времени (`time_point`), в который этот метод вызван. Вычитая два значения `time_point` (в нашем случае `t2-t1`), мы получаем продолжительность (`duration`). Можно использовать ключевое слово `auto`, чтобы не вдаваться в детали типов `duration` и `time_point`, которые достаточно сложны. В действительности возможности стандартной библиотеки по работе со временем были изначально разработаны для сложных физических приложений и являются существенно более гибкими и обобщенными, чем это требуется большинству пользователей.

Для получения промежутка времени `duration`, выраженного в определенных единицах, таких как `seconds`, `milliseconds` или `nanoseconds`, мы конвертируем ("приводим") его к соответствующим единицам с помощью функции преобразования `duration_cast`. Вам потребуется что-то наподобие `duration_cast`, поскольку различные системы и различные системные часы измеряют время в разных единицах. Не забывайте о `.count()`. Этот вызов извлекает количество единиц ("тактов часов", clock ticks) из объекта `duration`, содержащего как количество тактов, так и их продолжительность.

Таймер `system_clock` предназначен для измерения интервалов от долей секунд до нескольких секунд. Не пытайтесь с его помощью измерять промежутки времени длительностью в часы.

Еще раз напомним: нельзя доверять любым измерениям времени, которые нельзя повторить, получив примерно одинаковые результаты. Что значит "примерно одинаковые результаты"? Разумным будет разброс в 10%. Как мы уже говорили, современные компьютеры являются *быстрыми*: миллиард команд в секунду является обычной скоростью. Это значит, что вы не сможете измерить продолжительность ни одной операции, если она не повторяется десятки тысяч раз или если программа не работает действительно очень медленно, например, записывая данные на диск или обращаясь в веб. В последнем случае вы должны повторить действие несколько сотен раз, но такая медленная работа программы должна вас насторожить.

26.7. Ссылки

1. Stone, Debbie, Caroline Jarrett, MarkWoodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
2. Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2003. ISBN 0321194330.

Задание

Протестируйте функцию `binary_search`.

1. Реализуйте оператор ввода для класса `Test` из раздела 26.3.2.2.
2. Заполните файл тестов для последовательностей из раздела 26.3.

```
{ 1 2 3 5 8 13 21}              // "Обычная последовательность"
{ }
{ 1 }
{ 1 2 3 4 }                     // Нечетное количество элементов
{1 2 3 4 5 }                    // Четное количество элементов
{ 1  1  1  1  1  1  1 }         // Все элементы равны
{ 0 1 1 1 1 1 1 1 1 1 1 1 1 }   // Иной элемент в начале
{ 0 0 0 0 0 0 0 0 0 0 0 0 0 1 } // Иной элемент в конце
```

3. На основе раздела 26.3.1.3 завершите программу, генерирующую
   - очень большую последовательность (что считать большой последовательностью и почему?);
   - десять последовательностей со случайным количеством элементов;
   - десять последовательностей с 0, 1, 2, ..., 9 случайных (упорядоченных) элементов.
4. Повторите эти тесты для последовательностей строк, таких как `{Bohr Darwin Einstein Lavoisier Newton Turing}`.

Контрольные вопросы

1. Создайте список приложений, сопровождая их кратким описанием наихудшего события, которое может произойти из-за ошибки; например, управление самолетом — авиакатастрофа: гибель 231 человека; потеря оборудования на 500 млн. долларов.
2. Почему мы не можем просто доказать, что программа работает правильно?
3. В чем заключается разница между модульным и системным тестированием?
4. Что такое регрессивное тестирование и почему оно является важным?
5. Какова цель тестирования?
6. Почему функция `binary_search` сама не проверяет свои требования?
7. Если мы не можем проверить все возможные ошибки, то какие ошибки следует искать в первую очередь?
8. В каких местах кода, манипулирующего последовательностью элементов, вероятнее всего обнаружить ошибки?
9. Почему целесообразно тестировать программу при больших значениях?
10. Почему часто тесты представляются в виде данных, а не в виде кода?

11. Почему и когда мы используем многочисленные тесты, основанные на случайных величинах?
12. Почему трудно тестировать программы, использующие графический пользовательский интерфейс?
13. Что необходимо тестировать при проверке отдельного модуля?
14. Как связаны между собой тестируемость и переносимость?
15. Почему классы тестировать труднее, чем функции?
16. Почему важно, чтобы тесты были воспроизводимыми?
17. Что может сделать тестировщик, обнаружив, что модуль основан на непроверяемых предположениях (предусловиях)?
18. Как проектировщик/реализатор может улучшить тестирование?
19. Чем тестирование отличается от отладки?
20. Когда производительность имеет значение?
21. Приведите два (и более) примера того, как легко возникают проблемы с производительностью.

Ключевые слова

`system_clock`	модульный тест	регрессия
ветвление	охват теста	системный тест
входные данные	постусловия	состояние
выходные данные	предположения	тестирование
доказательство	предусловия	тестирование белого ящика
измерение времени	проектирование для тестирования	тестирование черного ящика

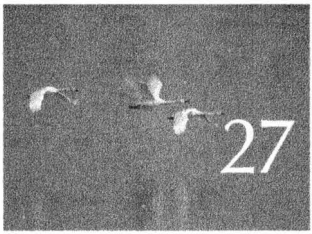
Упражнения

1. Выполните ваш алгоритм **`binary search`** из раздела 26.1 с тестами, представленными в разделе 26.3.1.
2. Модифицируйте тестирование функции **`binary_search`** для работы с элементами произвольного типа. Затем протестируйте ее на последовательности элементов типа **`string`** и чисел с плавающей точкой.
3. Повторите упр. 1 с вариантом функции **`binary_search`**, который получает в качестве аргумента критерий сравнения. Создайте список новых возможностей для появления ошибок, возникающих из-за дополнительного аргумента.
4. Разработайте формат для тестовых данных, чтобы можно было один раз задать последовательность и выполнить с ней несколько тестов.
5. Добавьте новый тест в набор тестов для функции **`binary_search`** и попытайтесь перехватить (маловероятную) ошибку модификации функцией входной последовательности.

6. Слегка модифицируйте калькулятор из главы 7, предусмотрев ввод из файла и вывод в файл (или используя возможности операционной системы для перенаправления ввода-вывода). Затем разработайте для него исчерпывающий набор тестов.
7. Протестируйте простой текстовый редактор из раздела 20.6.
8. Добавьте текстовый интерфейс к библиотеке графического пользовательского интерфейса из глав 12–15. Например, строка `Circle{Point {0,1},15}` должна генерировать вызов `Circle{Point{0,1},15}`. Используйте этот текстовый интерфейс для создания "детского рисунка": плоский домик с крышей, два окна и дверь.
9. Добавьте формат текстового вывода к библиотеке графического интерфейса. Например, при выполнении вызова `Circle{Point{0,1},15}` в поток вывода должна выводиться строка `Circle{Point{0,1},15}`.
10. Используя текстовый интерфейс из упр. 9, напишите более качественный тест для библиотеки графического пользовательского интерфейса.
11. Оцените время выполнения суммирования в примере из раздела 26.6, где `m` — квадратная матрица с размерностями, равными 100, 10 000, 1 000 000 и 10 000 000. Используйте случайные значения из диапазона [–10,10). Перепишите процедуру вычисления величины `v`, используя более эффективный (не $O(N^2)$) алгоритм, и сравните время его выполнения со временем работы исходного алгоритма.
12. Напишите программу, генерирующую случайные числа с плавающей точкой, и отсортируйте их с помощью функции `std::sort()`. Измерьте время, затраченное на сортировку 500 тысяч и 5 миллионов чисел типа `double`.
13. Повторите эксперимент из предыдущего упражнения, но со случайными строками, длина которых лежит в интервале [0, 100).
14. Повторите предыдущее упражнение, но на этот раз используйте контейнер `map`, а не `vector`, сортировать который не требуется.

Послесловие

Как программисты мы мечтаем о прекрасных программах, которые бы просто работали — и желательно с первой же попытки. Реальность иная: трудно сразу написать правильную программу и предотвратить появление в ней ошибок по мере того, как вы (и ваши коллеги) станете ее улучшать. Тестирование, включая проектирование с учетом тестирования, — это главный способ, гарантирующий, что система в итоге действительно будет работать. Живя в высокотехнологичном мире, мы должны в конце рабочего дня с благодарностью вспомнить о тестировщиках (о которых часто забывают).



Язык программирования C

C — строго типизированный язык программирования со слабой проверкой.

Деннис Ритчи (Dennis Ritchie)

Данная глава представляет собой краткий обзор языка программирования C и его стандартной библиотеки с точки зрения человека, знающего язык C++. В ней перечислены свойства языка C++, которых нет в языке C, и приведены примеры того, как программисты на языке C могут обойтись без них. Рассмотрены различия между языками C и C++, а также вопросы их одновременного использования. Приведены примеры ввода-вывода, работы со списком, управления памятью, а также операций над строками.

27.1. Языки C и C++: братья

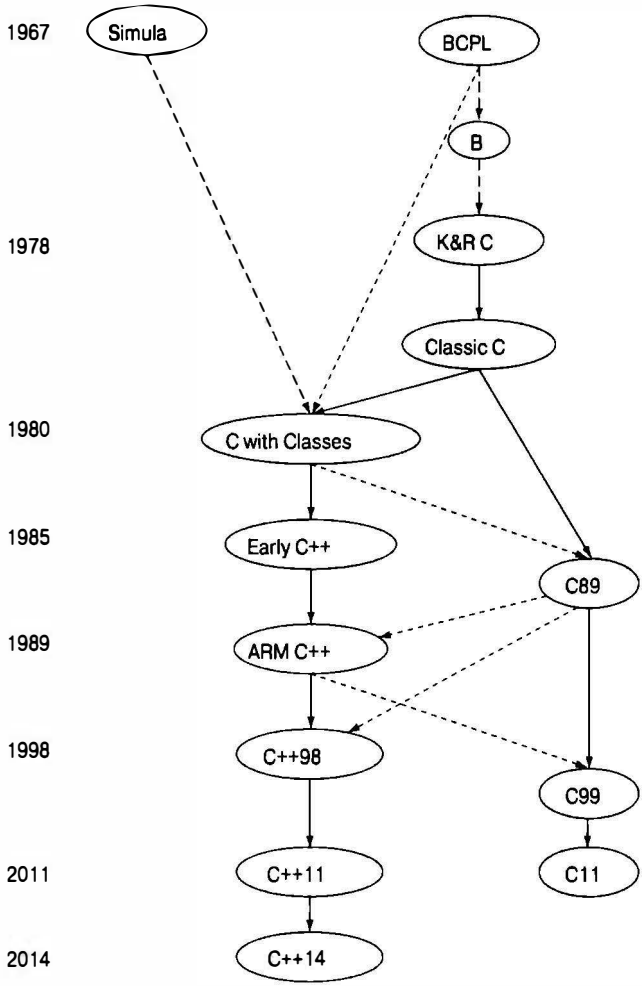
Язык программирования C был изобретен и реализован Деннисом Ритчи (Dennis Ritchie) из компании Bell Labs. Он изложен в книге *The C Programming Language* Брайана Кернигана (Brian Kernighan) и Денниса Ритчи (Dennis Ritchie) (в разговорной речи известной как "K&R"), которая, вероятно, является самым лучшим введением в язык C и одной из лучших книг по программированию (см. раздел 22.2.5). Текст исходного определения языка C++ по сути был редакцией определения языка C, написанного в 1980 году Деннисом Ритчи. После этого момента оба языка стали развиваться самостоятельно. Как и язык C++, язык C в настоящее время определен стандартом ISO.

Мы рассматриваем язык C в основном как подмножество языка C++. Следовательно, с точки зрения языка C++ проблемы описания языка C сводятся к двум вопросам.

- Описать те моменты, в которых язык C не является подмножеством языка C++.

- Описать те свойства языка C++, которых нет в языке C, и те возможности и приемы, с помощью которых этот недостаток можно компенсировать.

Исторически современный язык C++ и современный язык C являются "братьями". Они оба являются наследниками "классического C", диалекта языка C, описанного в первом издании книги Кернигана и Ритчи *The C Programming Language*, в который были добавлены присваивание структур и перечисления.

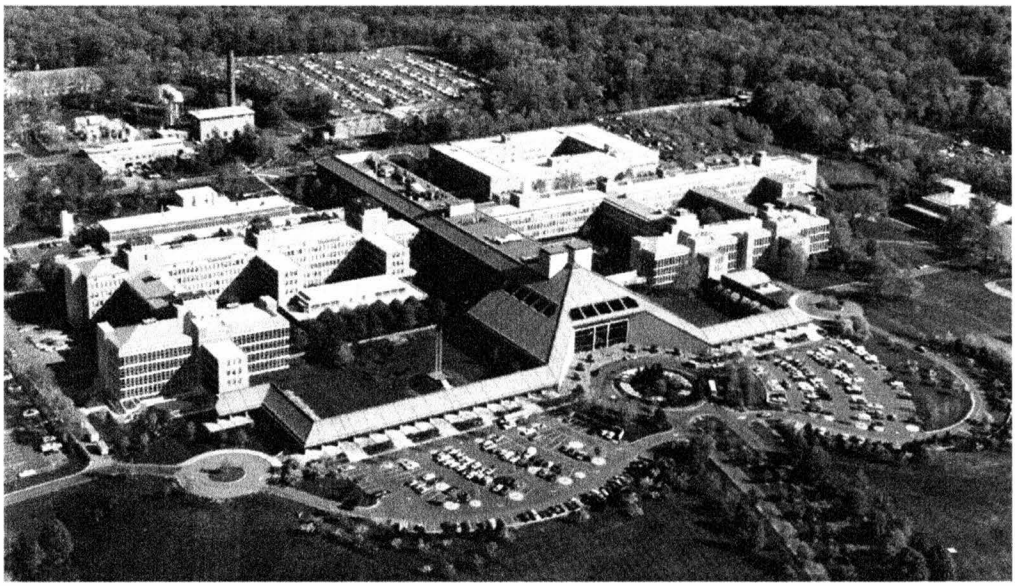


В настоящее время практически повсеместно все еще используется версия C89 (описанная во втором издании книги K&R[1]). Именно эта версия взята за основу в данной книге. Помимо этой версии, кое-где все еще

[1] Имеется русский перевод: Керниган Б., Ритчи Д. *Язык программирования C, 2-е изд.* — М.: ООО "И.Д. Вильямс", 2006. — *Примеч. ред.*

по-прежнему используется классический C, и есть несколько примеров использования версии C99, но это не должно стать проблемой для читателей, если они знают языки C++ и C89.

Языки C и C++ являются "детьми" Исследовательского центра компьютерных наук компании Bell Labs (Computer Science Research Center of Bell Labs), Мюррей-Хилл, штат Нью-Джерси (Murray Hill, New Jersey) (кстати, мой кабинет находился на противоположной стороне коридора в паре комнат от кабинетов Денниса Ритчи и Брайана Кернигана).



Оба языка в настоящее время определены и контролируются комитетами по стандартизации ISO. Для каждого языка разработано множество реализаций. Часто эти реализации поддерживают оба языка, причем желаемый язык устанавливается путем указания расширения исходного файла. По сравнению с другими языками, оба языка, C и C++, распространены на гораздо большем количестве платформ. Оба языка были разработаны и в настоящее время интенсивно используются для решения сложных программистских задач. Перечислим некоторые из них.

- Ядра операционных систем.
- Драйверы устройств.
- Встроенные системы.
- Компиляторы.
- Системы связи.

Между эквивалентными программами, написанными на языках C и C++, нет никакой разницы в производительности.

Как и язык C++, язык C очень широко используется. Взятые вместе, они образуют крупнейшее сообщество по разработке программного обеспечения на Земле.

27.1.1. Совместимость языков C и C++

Часто приходится встречать название "C/C++". Однако такого языка нет. Употребление такого названия обычно является признаком невежества. Мы используем такое название только в контексте вопросов совместимости и когда говорим о крупном сообществе программистов, использующих оба этих языка.

Язык C++ в основном, но не полностью, является надмножеством языка C. За несколькими очень редкими исключениями конструкции, общие для языков C и C++, имеют одинаковый смысл (семантику). Язык C++ был разработан так, чтобы быть "как можно ближе к языку C++, но не ближе, чем следует":

- для простоты перехода от одного языка к другому;
- для совместимости.

Многие свойства, оказавшиеся несовместимыми с языком C, объясняются более строгой проверкой типов в языке программирования C++.

Примером программы, допустимой на языке C, но не на языке C++, является программа, в которой ключевые слова из языка C++ используются в качестве идентификаторов (раздел 27.3.2).

```
int class(int new, int bool); /* C, но не C++ */
```

Примеры, в которых семантика конструкции, допустимой в обоих языках, отличается в них, найти труднее, но все же они существуют.

```
int s = sizeof('a'); /* sizeof(int), обычно 4 в языке C и 1 в языке C++ */
```

Строковый литерал, такой как `'a'`, в языке C имеет тип `int` и тип `char` — в языке C++. Однако для переменной `ch` типа `char` в обоих языках выполняется условие `sizeof(ch)==1`.

Информация, касающаяся совместимости и различий между языками, не так интересна. В языке C нет никаких изощренных методов программирования, которые стоило бы изучать специально. Вам может понравиться вывод данных с помощью функции `printf()` (раздел 27.6), но за возможным исключением этой функции (а также некоторых попыток пошутить) эта глава имеет довольно сухое и формальное содержание. Ее цель проста: дать читателям возможность читать и писать программы на языке C, если возникнет такая необходимость. Она содержит также предупреждения об опасностях, которые очевидны для опытных программистов, работающих на языке C, но, как правило, неожиданных для программистов, работающих на языке C++. Мы надеемся, что вы научитесь избегать этих опасностей с минимальными потерями.

Большинство программистов, работающих на языке C++, рано или поздно так или иначе сталкиваются с программами, написанными на языке C. Аналогично программисты, создающие программы на языке C, часто вынуждены работать с программами, написанными на языке C++. Большинство из того, что мы описываем в этой главе, уже знакомо программистам, работающим на языке C, но некоторые из этих сведений могут быть отнесены к уровню экспертов. Причина проста: не все имеют одинаковое представление об уровне экспертов, поэтому мы просто описываем то, что часто встречается в реальных программах. Рассуждения о вопросах совместимости может быть дешевым способом добиться незаслуженной репутации "эксперта по языку C". Однако следует помнить: реальный опыт достигается благодаря практическому использованию языка (в данном случае языка C), а не изучению эзотерических правил языка (как это излагается в разделах, посвященных совместимости).

Ссылки

1. ISO/IEC 9899:1999. *Programming Languages — C*. В этой книге описан язык C99; большинство компиляторов реализует язык C89 (часто с некоторыми расширениями).
2. ISO/IEC 9899:2011. *Programming Languages — C*. Определяет язык C11.
3. ISO/IEC 14882:2011. *Programming Languages — C++*.
4. Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Addison-Wesley, 1988. ISBN 0131103628.[2]
5. Stroustrup, Bjarne. "Learning Standard C++ as a New Language". *C/C++ Users Journal*, May 1999.
6. Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility". *The C/C++ Users Journal*, July, Aug., and Sept. 2002.
7. Статьи Страуструпа легко найти среди публикаций на его начальной странице.

27.1.2. Возможности C++, отсутствующие в C

С точки зрения языка C++ в языке C (т.е. в версии C89) нет многих возможностей.

- Классы и функции-члены
  - Используйте структуры и глобальные функции.

[2] Имеется русский перевод: Керниган Б., Ритчи Д. *Язык программирования C, 2-е изд.* — М.: ООО "И.Д. Вильямс", 2006. — Примеч. ред.

- Производные классы и виртуальные функции
  - Используйте структуры, глобальные функции и указатели на функции (раздел 27.2.3).
- Шаблоны и встраиваемые функции
  - Используйте макросы (раздел 27.8).
- Исключения
  - Используйте коды ошибок, возвращаемые значения, указывающие на ошибку, и т.п.
- Перегрузка функций
  - Давайте каждой функции отдельное имя.
- Операторы `new/delete`
  - Используйте функции `malloc()/free()` и отдельный код для инициализации и очистки.
- Ссылки
  - Используйте указатели.
- Ключевые слова `const`, `constexpr` и функции в константных выражениях
  - Используйте макросы.
- Тип `bool`
  - Используйте тип `int`.
- `static_cast`, `reinterpret_cast` и `const_cast`
  - Используйте приведения в стиле C, т.е. `(int)a` вместо `static_cast<int>(a)`.

На языке C написано много полезных программ, поэтому этот список должен служить напоминанием о том, что ни одно свойство языка не является абсолютно необходимым. Большинство языковых возможностей — и даже большинство свойств языка C — разработано просто для удобства программистов. В конце концов, при достаточном запасе времени, мастерстве и терпении любую программу можно написать на ассемблере. Обратите внимание на то, что благодаря близости моделей языков C и C++ к реальным компьютерам они позволяют имитировать многие стили программирования.

Остальная часть этой главы посвящена объяснению того, как писать полезные программы без помощи этих свойств. Наши основные советы по использованию языка C++ сводятся к следующему.

- Имитируйте методы программирования, для которых разработаны возможности языка C++, с помощью возможностей языка C.

- Когда вы пишете программу на языке C, пишите программы на языке, являющемся подмножеством языка C++.
- Используйте предупреждения компилятора для проверки аргументов функций.
- Для больших программ используйте инструменты проверки (см. раздел 27.2.2).

Многие детали, касающиеся несовместимости языков C и C++, устарели и носят скорее технический характер. Однако, для того чтобы читать и писать на языке C, вы не обязаны помнить о них.

- Компилятор сам напомнит вам, если вы станете использовать средства языка C, которых нет в языке C.
- Если вы следуете правилам, перечисленным выше, то вряд ли столкнетесь с чем-либо таким, что в языке C имеет другой смысл по сравнению с языком C++.

При отсутствии возможностей языка C++ некоторые средства в языке C приобретают особое значение.

- Массивы и указатели.
- Макросы.
- Оператор `typedef` (эквивалент C и C++98 для объявления `using`; см. разделы 20.5 и A.16).
- Оператор `sizeof`.
- Операторы приведения типов.

В этой главе будет приведено несколько примеров использования таких средств.

Я ввел в язык C++ комментарии `//`, унаследованные от его предшественника, языка BCPL, когда мне надоело печатать комментарии `/*...*/`. Комментарии `//` приняты в большинстве диалектов языка, включая версию C99, поэтому их можно использовать совершенно безопасно. В наших примерах мы будем использовать комментарии вида `/*...*/` исключительно для того, чтобы показать, что мы пишем программу на языке C.

В языках C99 и C11 реализованы некоторые иные возможности языка C++ (а также некоторые возможности, несовместимые с языком C++), но мы будем придерживаться версии C89, поскольку она используется наиболее широко.

27.1.3. Стандартная библиотека языка C

Естественно, возможности библиотек языка C++, зависящие от классов и шаблонов, в языке C недоступны. Перечислим некоторые из них.

- Класс `vector`.
- Класс `map`.
- Класс `set`.
- Класс `string`.
- Алгоритмы STL, такие как `sort()`, `find()` или `copy()`.
- Потоки ввода-вывода `iostream`.
- Класс `regex`.

Из-за этого библиотеки языка C часто основаны на массивах, указателях и функциях. К основной части стандартной библиотеки языка C относятся следующие заголовочные файлы.

- `<stdlib.h>`. Общие утилиты (например, `malloc()` и `free()`); см. раздел 27.4.
- `<stdio.h>`. Стандартный механизм ввода-вывода; см. раздел 27.6.
- `<string.h>`. Манипуляции строками и памятью в стиле языка C; см. раздел 27.5.
- `<math.h>`. Стандартные математические функции для операций над числами с плавающей точкой; см. раздел 24.8.
- `<errno.h>`. Коды ошибок математических функций из заголовочного файла `<math.h>`; см. раздел 24.8.
- `<limits.h>`. Размеры целочисленных типов; см. раздел 24.2.
- `<time.h>`. Функции даты и времени; см. раздел 26.6.1.
- `<assert.h>`. Утверждения (для отладки); см. раздел 27.9.
- `<ctype.h>`. Классификация символов; см. раздел 11.6.
- `<stdbool.h>`. Булевы макросы.

Полное описание стандартной библиотеки языка C можно найти в соответствующем учебнике, например в книге K&R. Все эти библиотеки (и заголовочные файлы) доступны также и в языке C++.

27.2. Функции

В языке C есть несколько особенностей при работе с функциями.

- Может существовать только одна функция с заданным именем.
- Проверка типов аргументов функции является необязательной.
- Ссылок нет (а значит, нет и механизма передачи аргументов по ссылке).
- Нет функций-членов.
- Нет встраиваемых функций (за исключением версии C99).
- Существует альтернативный синтаксис определения функций.

Помимо этого, все остальное мало отличается от языка C++. Изучим указанные отличия по отдельности.

27.2.1. Отсутствие перегрузки имен функций

Рассмотрим следующий пример:

```
void print(int);            /* Вывод целого числа */
void print(const char*);    /* Вывод строки */ /* Ошибка! */
```

Второе объявление является ошибкой, потому что в программе, написанной на языке C, не может быть двух функций с одним и тем же именем. Итак, нам необходимо придумать подходящую пару имен.

```
void print_int(int);            /* Вывод целого числа int */
void print_string(const char*); /* Вывод строки */
```

Иногда это оказывается преимуществом: теперь вы не сможете случайно использовать неправильную функцию для вывода целого числа! Очевидно, что нас такой аргумент убедить не сможет, а отсутствие перегруженных функций усложняет реализацию идей обобщенного программирования, поскольку они основаны на семантически похожих функциях, имеющих одинаковые имена.

27.2.2. Проверка типов аргументов функций

Рассмотрим следующий пример:

```
int main()
{
    f(2);
}
```

Компилятор языка C допускает такой код: вы не обязаны объявлять функции до их использования (хотя можете и должны). Определение функции `f()` может находиться где-то в другом месте. Кроме того, функция `f()` может находиться в другом модуле компиляции, но если это не так, редактор связей сообщит об ошибке.

К сожалению, это определение в другом исходном файле может выглядеть следующим образом:

```
/* other_file.c: */
int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

Редактор связей не сообщит об этой ошибке. Вместо этого вы получите ошибку на этапе выполнения программы или случайный результат.

Как же решить эту проблему? На практике программисты придерживаются согласованного использования заголовочных файлов. Если все функции, которые вы вызываете или определяете, объявлены в заголовочном файле, включенном в соответствующем месте программы с помощью директивы **#include**, будет включен механизм проверки типов. Однако в больших программах на это трудно рассчитывать. Вследствие этого в большинстве компиляторов языка C существуют ключи, предусматривающие выдачу предупреждений о вызовах необъявленных функций: воспользуйтесь ими. Кроме того, с первых дней существования языка C появились программы, с помощью которых можно выявлять все возможные проблемы, связанные непротиворечивостью типов. Обычно они называются *lint*. Используйте их для любой нетривиальной программы на языке C. Вы обнаружите, что программы lint подталкивают вас использовать язык C как подмножество языка C++. Одно из наблюдений, приведших к разработке языка C++, состояло в том, что компилятор мог легко проверять многое (хотя и не все), что могли проверять программы lint.

Вы можете запросить проверку аргументов функций в языке C. Для этого достаточно объявить функцию с заданными типами аргументов (точно так же, как в языке C++). Такое объявление называется *прототипом функции* (function prototype). Тем не менее следует избегать объявлений, не указывающих аргументы; они *не являются* прототипами функций и не включают механизм проверки типов.

```
int g(double); /* Прототип - как в языке C++ */
int h();       /* Не прототип - типы аргументов не указаны */

void my_fct()
{
    g();       /* Ошибка: пропущен аргумент */
    g("asdf"); /* Ошибка: неправильный тип аргумента */
    g(2);      /* OK: 2 преобразуется в 2.0 */
    g(2,3);    /* Ошибка: один аргумент лишний */

    h();       /* Допустимо! Результат непредсказуем */
    h("asdf"); /* Допустимо! Результат непредсказуем */
    h(2);      /* Допустимо! Результат непредсказуем */
    h(2,3);    /* Допустимо! Результат непредсказуем */
}
```

В объявлении функции **h()** не указан тип аргумента. Это не означает, что функция **h()** не получает ни одного аргумента; это значит, что она принимает любой набор аргументов и надеется, что он при вызове окажется правильным. Еще раз заметим, что хороший компилятор предупредит об этой проблеме, а программа lint перехватит ее.

C++	Эквивалент в языке C
`void f();  // Предпочтительно`	`void f(void);`
`void f(void);`	`void f(void);`
`void f(...); // Любые аргументы`	`void f();  // Любые аргументы`

Существует специальный набор правил, регламентирующих преобразование аргументов, если в области видимости нет прототипа функции. Например, переменные типов **char** и **short** преобразуются в переменные типа **int**, а переменные типа **float** — в переменные типа **double**. Если вы хотите знать, скажем, что произойдет с переменной типа **long**, загляните в хороший учебник по языку C. Наша рекомендация проста: не вызывайте функций, не имеющих прототипов.

Обратите внимание на то, что, хотя компилятор допускает передачу аргументов неправильного типа, например типа **char*** вместо параметра типа **int**, использование таких аргументов приводит к ошибкам. Как сказал Деннис Ритчи, "C — строго типизированный язык программирования со слабой проверкой".

27.2.3. Определения функций

Функции можно определять точно так же, как в языке C++. Эти определения являются прототипами функций.

```
double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square(2);        /* OK: преобразование 2 в 2.0 */
    double y = square();         /* Пропущен аргумент */
    double y = square("Hello"); /* Ошибка: неправильный тип */
    double y = square(2,3);      /* Ошибка: много аргументов */
}
```

Определение функции без аргументов не является прототипом функции.

```
void f() {    /* Что-то делает */ }

void g()
{
    f(2);     /* OK в языке C; ошибка в языке C++ */
}
```

Код

```
void f();     /* Тип аргумента не указан */
```

означающий, что функция **f()** может принять любое количество аргументов любого типа, выглядит действительно странно. В ответ на это я

изобрел новое обозначение, в котором понятие "ничего" указывалось явным образом с помощью ключевого слова **void** (*void* — слово из четырех букв, означающее "ничего"):

```
void f(void); /* Не принимает никаких аргументов */
```

Впрочем, вскоре я об этом пожалел, потому что эта конструкция выглядит странно и при последовательной проверке типов аргументов является излишней. Что еще хуже, Деннис Ритчи (автор языка C) и Дуг Макилрой (Doug McIlroy, законодатель мод в Исследовательском центре компьютерных наук в компании Bell Labs (Bell Labs Computer Science Research Center; см. раздел 22.2.5)) назвали это решение "отвратительным". К сожалению, оно стало очень популярным среди программистов, работающих на языке C. Тем не менее не используйте его в программах на языке C++, в которых оно не только выглядит уродливо, но и является совершенно излишним.

В языке C есть возможность альтернативного определения функции в стиле языка Algol60, в котором типы параметров (не обязательно) указываются отдельно от их имен.

```
int old_style(p,b,x) char* p; char b;
{
    /* ... */
}
```

Такое "определение в старом стиле" предвосхищает конструкции языка C++ и не является прототипом. По умолчанию аргумент без объявленного типа считается аргументом типа **int**. Итак, параметр **x** является аргументом функции **old_style()**, имеющим тип **int**. Мы можем вызвать функцию **old_style()** следующим образом:

```
old_style();                /* OK: пропущены все аргументы */
old_style("hello",'a',17); /* OK: аргументы правильного типа */
old_style(12, 13, 14);      /* OK: 12  - неправильный тип, но */
                            /* ведь p может не использоваться */
```

Компилятор должен пропустить эти вызовы (но мы надеемся, что он предупредит о первом и третьем аргументах).

Мы рекомендуем придерживаться следующих правил проверки типов аргументов функций.

- Последовательно используйте прототипы функций (используйте заголовочные файлы).
- Установите уровень предупреждений компилятора так, чтобы отлавливать ошибки, связанные с типами аргументов.
- Используйте (какую-нибудь) программу lint.

В результате вы получите код, который одновременно будет кодом на языке C++.

27.2.4. Вызов C-функций из C++-программы и наоборот

Вы можете установить связи между файлами, скомпилированными с помощью компилятора языка C, и файлами, скомпилированными с помощью компилятора языка C++, только если компиляторы предусматривают такую возможность. Например, можете связать объектные файлы, сгенерированные из кода на языке C и C++, используя компиляторы GNU C и GCC. Можно также связать объектные файлы, сгенерированные из кода на языках C и C++, используя компиляторы Microsoft C и C++ (MSC++). Это обычная и полезная практика, позволяющая использовать больше библиотек, чем это возможно при использовании только одного из этих языков.

В языке C++ предусмотрена более строгая проверка типов, чем в языке C. В частности, компилятор и редактор связей для языка C++ проверяют, согласованно ли определены и используются функции **f(int)** и **f(double)**, даже если они находятся в разных исходных файлах. Редактор связей для языка C не проводит такой проверки. Для того чтобы вызвать функцию, определенную в языке C, в программе, написанной на языке C++, и наоборот, необходимо сообщить компилятору о том, что вы собираетесь сделать.

```
// Вызов функции на языке C из кода на языке C++:

extern "C" double sqrt(double); // Связь с C-функцией

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

По существу, выражение **extern "C"** сообщает компилятору о том, что вы используете соглашения, принятые компилятором языка C. Кроме этого, с точки зрения языка C++ в этой программе все нормально. Фактически стандартная функция **sqrt(double)** из языка C++ обычно входит и в стандартную библиотеку языка C. Для того чтобы вызвать функцию из библиотеки языка C в программе, написанной на языке C++, ничего не требуется. Язык C++ просто адаптирован к соглашениям, принятым в редакторе связей языка C.

Мы можем также использовать выражение **extern "C"**, чтобы сделать функцию языка C++ вызываемой из программы, написанной на языке C.

```
// C++-функция, которую можно вызвать из кода на языке C:
extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```

Теперь в программе на языке C можно косвенно вызвать функцию-член **f()**.

```
/* Вызов C++-функции из функции на языке C: */
int call_f(S* p, int i);
struct S* make_S(int,const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p,i);
    /* ... */
}
```

Больше о языке C++ упоминать необязательно, данная конструкция будет работать и так.

Выгоды такого взаимодействия очевидны: код можно писать на смеси языков C и C++. В частности, программы на языке C++ могут использовать библиотеки, написанные на языке C, а программы на языке C могут использовать библиотеки, написанные на языке C++. Более того, большинство языков (например, Fortran) имеют интерфейс вызова функций, написанных на языке C, и допускают вызов своих функций в программах, написанных на языке C.

В приведенных выше примерах мы предполагали, что программы, написанные на языках C и C++, совместно используют объект, на который указывает указатель `p`. Это условие выполняется для большинства объектов классов. В частности, допустим, что у нас есть следующий класс:

```
// В языке C++:
class complex {
    double re, im;
public:
    // Все обычные операции
};
```

Тогда можно обойтись без передачи указателя на объект в программу, написанную на языке C, и наоборот. Можно даже получить доступ к членам `re` и `im` в программе, написанной на языке C, с помощью объявления

```
/* В языке C: */
struct complex {
    double re, im;
    /* Никаких операций */
};
```

Правила компоновки в любом языке могут быть сложными, а правила компоновки модулей, написанных на нескольких языках, иногда даже трудно описать. Тем не менее функции, написанные на языках C и C++, могут обмениваться объектами встроенных типов и классами (структурами) без виртуальных функций. Если класс содержит виртуальные функции, можете просто передать указатели на его объекты

и предоставить работу с ними коду, написанному на языке C++. Примером этого правила является функция `call_f()`: функция-член `f()` может быть виртуальной. Следовательно, этот пример иллюстрирует вызов виртуальной функции из программы, написанной на языке C.

Кроме встроенных типов, простейшим и наиболее безопасным способом совместного использования типов является конструкция `struct`, определенная в общем заголовочном файле языков C и C++. Однако эта стратегия серьезно ограничивает возможности использования языка C++, поэтому мы ее не рекомендуем.

27.2.5. Указатели на функции

Что можно сделать на языке C, если мы хотим использовать объектно-ориентированную технологию (см. разделы 14.2–14.4)? По существу, нам нужна какая-то альтернатива виртуальным функциям. Большинству людей в голову, в первую очередь, приходит мысль использовать структуру с "полем типа" ("type field"), описывающим, какой вид фигуры представляет данный объект. Рассмотрим пример.

```
struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* ... */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
    case circle:
        /* Рисуем окружность */
        break;
    case rectangle:
        /* Рисуем прямоугольник */
        break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* ... */
}
```

Этот прием срабатывает. Однако есть две загвоздки.

- Для каждой "псевдовиртуальной" функции (такой, как функция `draw()`) мы должны написать новую инструкцию `switch`.
- Каждый раз, добавляя новую фигуру, мы должны модифицировать каждую псевдовиртуальную функцию (такую, как функция `draw()`), добавляя новый раздел `case` в инструкцию `switch`.

Вторая проблема носит довольно неприятный характер, поскольку она означает, что мы не можем включать псевдовиртуальные функции ни в какие библиотеки — поскольку наши пользователи должны будут довольно часто модифицировать эти функции. Наиболее эффективной альтернативой является использование указателей на функции.

```
typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* ... */
};

void draw(struct Shape2* p)
{
    (p->draw)(p);
}

void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}
```

Структуру `Shape2` можно использовать точно так же, как структуру `Shape1`.

```
int f(struct Shape2* pp)
{
    draw(pp);
    /* ... */
}
```

Проделав небольшую дополнительную работу, мы можем добиться, чтобы объекту было не обязательно содержать по одному указателю на каждую псевдовиртуальную функцию. Вместо этого можно хранить указатель на массив указателей на функции (это очень похоже на то, как реализованы виртуальные функции в языке C++). Основная проблема при использовании таких схем в реальных программах заключается в том, чтобы правильно инициализировать все эти указатели на функции.

27.3. Второстепенные языковые различия

В этом разделе приводятся примеры незначительных различий между языками C и C++, которые могут вызвать затруднения у читателей, если они впервые о них слышат. Некоторые из них оказывают серьезное влияние на программирование, поскольку их надо учитывать явным образом.

27.3.1. Дескриптор пространства имен `struct`

В языке C имена структур (в нем нет ключевого слова **class**, а есть только слово **struct**) находятся в отдельном от остальных идентификаторов пространстве имен. Следовательно, имени каждой структуры (называемому *дескриптором структуры* (structure tag)) должно предшествовать ключевое слово **struct**. Рассмотрим пример.

```
struct pair { int x,y; };
pair p1;        /* Ошибка: идентификатора pair нет  */
                /* в области видимости              */
struct pair p2; /* OK                               */
int pair = 7;   /* OK: дескриптор структуры pair не */
                /* в области видимости              */
struct pair p3; /* OK: дескриптор структуры pair не */
                /* маскируется типом int            */
pair = 8;       /* OK: идентификатор pair ссылается */
                /* на число типа int                */
```

Довольно интересно, что, применив обходной маневр, этот прием можно заставить работать и в языке C++. Присваивание переменным (и функциям) тех же имен, что и структурам, — весьма распространенный трюк в программах на языке C, хотя мы его не рекомендуем.

Если вы не хотите писать ключевое слово **struct** перед именем каждой структуры, используйте оператор **typedef** (см. раздел 20.5). Широко распространена следующая идиома:

```
typedef struct { int x,y; } pair;
pair p1 = { 1, 2 };
```

В общем случае оператор **typedef** используется чаще и является более полезным в программах на языке C, в которых у программиста нет возможности определять новые типы и связанные с ними операции.

В языке C имена вложенных структур находятся в той же области видимости, что и имя структуры, в которую они вложены. Рассмотрим пример.

```
struct S {
    struct T { /* ... */ };
    / * ... */
};

struct T x; /* OK в языке C (но не в C++) */
```

В программе на языке C++ этот фрагмент следовало бы написать так:

```
S::T x;     // OK в языке C++ (но не в C)
```

Если это возможно, не используйте вложенные структуры в программах на языке C: их правила разрешения области видимости отличаются от наивных (и вполне разумных) предположений большинства людей.

27.3.2. Ключевые слова

Многие ключевые слова в языке C++ не являются ключевыми словами в языке C (поскольку язык C не обеспечивает соответствующие функциональные возможности) и поэтому могут использоваться как идентификаторы в программах на языке C.

Ключевые слова C++, которые не являются ключевыми словами C				
alignas	class	inline	private	true
alignof	compl	mutable	protected	try
and	concept	namespace	public	typeid
and_eq	const_cast	new	reinterpret_cast	typename
asm	constexpr	noexcept	requires	using
bitand	delete	not	static_assert	virtual
bitor	dynamic_cast	not_eq	static_cast	wchar_t
bool	explicit	nullptr	template	xor
catch	export	operator	this	xor_eq
char16_t	false	or	thread_local	
char32_t	friend	or_eq	throw	

Не используйте эти имена в качестве идентификаторов в программах на языке C, иначе ваш код станет несовместимым с языком C++. Если вы используете одно из этих имен в заголовочном файле, то не сможете использовать этот файл в программе на языке C++.

Некоторые ключевые слова в языке C++ являются макросами в языке C.

Ключевые слова C++, которые являются макросами в C				
and	bitor	false	or	wchar_t
and_eq	bool	not	or_eq	xor
bitand	compl	not_eq	true	xor_eq

В языке C они определены в заголовочных файлах `<iso646.h>` и `<stdbool.h>` (`bool`, `true`, `false`). Не пользуйтесь тем фактом, что они являются макросами в языке C.

27.3.3. Определения

Язык C++ допускает определения в большем количестве мест программы по сравнению с языком C89. Рассмотрим пример.

```
for(int i = 0; i<max; ++i)        // Определение переменной i,
    x[i] = y[i];                  // недопустимое в языке C
while (struct S* p = next(q)) {   // Определение указателя p,
    /* ... */                     // недопустимое в языке C
}
```

```
void f(int i)
{
    if (i< 0 || max<=i) error("ошибка диапазона");
    int a[max];      // Ошибка: объявление после инструкции
                     // в языке C не разрешено
    /* ... */
}
```

Язык C (C89) не допускает объявлений в разделе инициализации цикла **for**, в условиях и после инструкций в блоке. Мы должны переписать предыдущий фрагмент примерно следующим образом:

```
int i;
for (i = 0; i<max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /* ... */
}

void f(int i)
{
    if (i< 0 || max<=i) error("ошибка диапазона");
    {
        int a[max];
        /* ... */
    }
}
```

В языке C++ неинициализированное объявление считается определением; в языке C оно считается простым объявлением, поэтому его можно дублировать.

```
int x;
int x; /* Определяет или объявляет одну целочисленную переменную
          с именем x в программе на языке C; ошибка в C++ */
```

В языке C++ сущность должна быть определена только один раз. Ситуация становится интереснее, если эти две переменные типа **int** с одинаковыми именами находятся в разных модулях компиляции.

```
/* В файле x.c: */
int x;

/* В файле y.c: */
int x;
```

Ни компилятор языка C, ни компилятор языка C++ не найдет никаких ошибок в файлах **x.c** or **y.c**. Но если файлы **x.c** и **y.c** скомпилировать как файлы на языке C++, то редактор связей выдаст сообщение об ошибке, связанной с двойным определением. Если же файлы **x.c** и **y.c** скомпилировать на языке C, то редактор связей не выдаст сообщений об ошибке и

(в полном соответствии с правилами языка C) будет считать, что речь идет об одной и той же переменной **x**, совместно используемой в файлах **x.c** и **y.c**. Если вы хотите, чтобы в программе всеми модулями совместно использовалась одна глобальная переменная **x**, то сделайте это явно, как показано ниже.

```
/* В файле x.c: */
int x = 0;          /* Определение */

/* В файле y.c: */
extern int x;       /* Объявление, но не определение */
```

Впрочем, лучше используйте заголовочный файл.

```
/* В файле x.h: */
extern int x;       /* Объявление, но не определение */

/* В файле x.c: */
#include "x.h"
int x = 0;          /* Определение */

/* В файле y.c: */
#include "x.h"
/* Объявление переменной x находится в заголовочном файле */
```

Но лучше всего избегать глобальных переменных вовсе.

27.3.4. Приведение типов в стиле языка C

В языке C (и в языке C++) можно явно привести переменную **v** к типу **T**, используя минимальные обозначения.

```
(T)v
```

Это так называемое "приведение в стиле языка C", или "приведение в старом стиле". Его любят люди, не умеющие набирать тексты (за лаконичность) и ленивые (потому что они не обязаны знать, что нужно для того, чтобы из переменной **v** получилась переменная типа **T**). С другой стороны, этот стиль яростно отвергают программисты, занимающиеся сопровождением программ, поскольку такие преобразования остаются практически незаметными и никак не привлекают к себе внимания. Приведения в языке C++ (*приведения в новом стиле* (new-style casts), или *приведения в шаблонном стиле* (template-style casts); см. раздел A.5.7) осуществляют явное преобразование типов, которое легко заметить. В языке C у вас нет выбора.

```
int* p = (int*)7; /* Интерпретирует битовую комбинацию:
                     reinterpret_cast<int*>(7) */
int x = (int)7.5; /* Усечение: static_cast<int>(7.5) */
```

```
typedef struct S1 { /* ... */ } S1;
typedef struct S2 { /* ... */ } S2;
S1 a;
const S2 b;         /* В языке C допускаются неинициализированные
                       константы */

S1* p = (S2*)&a;    /* Интерпретация битовой комбинации:
                       reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b;    /* Отбрасывание модификатора const:
                       const_cast<S2*>(&b) */
S1* r = (S1*)&b;    /* Удаление const и изменение типа;
                       вероятно, ошибка */
```

Мы не рекомендуем использовать макросы даже в программах на языке C (раздел 27.8), но, возможно, описанные выше идеи можно было бы выразить следующим образом:

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST(S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

Это не обеспечит проверку типов при выполнении операторов **reinterpret_cast** и **const_cast**, но хотя бы сделает эти ужасные операции заметными и привлечет внимание программиста.

27.3.5. Преобразование указателей типа void*

В языке C указатель типа **void*** можно использовать как в правой части оператора присваивания, так и для инициализации указателей любого типа; в языке C++ это невозможно. Рассмотрим пример.

```
void* alloc(size_t x);              /* Выделяет x байтов */

void f (int n)
{
    int* p = alloc(n*sizeof(int)); /* OK в языке C;
                                      ошибка в языке C++ */
    /* ... */
}
```

Здесь указатель типа **void*** возвращается как результат функции **alloc()** и неявно преобразовывается в указатель типа **int***. В языке C++ мы могли бы переписать эту строку следующим образом:

```
int* p = (int*)alloc(n*sizeof(int)); /* OK и в C, и в C++ */
```

Мы использовали приведение в стиле языка C (раздел 27.3.4), чтобы оно оказалось допустимым как в программах на языке C, так и в программах на языке C++.

Почему неявное преобразование `void*` в `T*` является недопустимым в языке C++? Потому, что такие преобразования могут быть небезопасными.

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q; /* Небезопасно; разрешено в C, ошибка в C++ */
    *pp = -1;    /* Перезапись памяти, начиная с адреса &i   */
}
```

В данном случае мы даже не знаем, какой фрагмент памяти будет перезаписан. Возможно, переменная `j` и часть указателя `p`? А может быть, память, использованная для управления вызовом функции `f()` (стек функции `f`)? Какие бы данные ни были перезаписаны, вызов функции `f()` приведет к печальным последствиям.

Обратите внимание на то, что (обратное) преобразование указателя типа `T*` в указатель типа `void*` является совершенно безопасным, — вы не сможете придумать ужасные примеры, подобные предыдущему, — и они допускаются как в языке C, так и в языке C++.

К сожалению, неявное преобразование `void*` в `T*` широко распространено в языке C и, вероятно, является основной проблемой совместимости языков C и C++ в реальных программах (см. раздел 27.4).

27.3.6. Перечисление

В языке C можно присваивать целое число перечислению без приведения `int` к `enum`:

```
enum color { red, blue, green };
int x = green;       /* OK в языках C и C++ */
enum color col = 7;  /* OK в языке C; ошибка в языке C++ */
```

Одним из следствий этого факта является то, что в программах на языке C мы можем применять операции инкремента (++) и декремента (--) к переменным, являющимся перечислениями. Это может быть удобным, но одновременно небезопасным.

```
enum color x = blue;
++x; /* x получает значение green; ошибка в языке C++ */
++x; /* x получает значение 3;     ошибка в языке C++ */
```

Выход за пределы перечисления может входить в наши планы, но может быть и совершенно неожиданным.

Обратите внимание на то, что, подобно дескрипторам структур, имена перечислений находятся в собственном пространстве имен, поэтому

каждый раз при указании имени перечисления перед ним следует ставить ключевое слово **enum**.

```
color c2 = blue;       /* Ошибка в языке C: переменная color не
                          в области видимости; ОК в языке C++ */
enum color c3 = red;   /* ОК */
```

27.3.7. Пространства имен

В языке C нет пространств имен (в том смысле, как это принято в языке C++). Так что же можно сделать, чтобы избежать коллизий имен в больших программах, написанных на языке C? Как правило, для этого используются префиксы и суффиксы.

```
/* В bs.h: */
typedef struct bs_string
    { /* ... */ } bs_string; /* Строка Бьярне    */
typedef int bs_bool ;         /* Булев тип Бьярне */

/* В pete.h: */
typedef char* pete_string;    /* Строка Пита      */
typedef char pete_bool ;      /* Булев тип Пита   */
```

Этот прием настолько широко используется, что использовать одно- и двухбуквенные префиксы обычно уже недостаточно.

27.4. Динамическая память

В языке C нет операторов **new** и **delete**, работающих с объектами. Для использования динамической памяти в нем используются функции, работающие с памятью. Наиболее важные функции определены в стандартном заголовочном файле общих утилит **<stdlib.h>**.

```
void* malloc(size_t sz);  /* Выделить sz байтов */
void free(void* p);       /* Освободить область памяти, на
                             которую указывает указатель p */
void* calloc(size_t n,    /* Выделить n*sz байтов,
             size_t sz);     инициализировав их нулями*/
void* realloc(void* p,    /* Перенести память, на которую
              size_t sz);    указывает p, в новый блок
                             размером sz */
```

Тип **size_t** — это беззнаковый тип, определенный с помощью **typedef** в заголовочном файле **<stdlib.h>**.

Почему функция **malloc()** возвращает указатель **void***? Потому что она не имеет информации о том, объект какого типа вы хотите разместить в памяти. Инициализация — это ваша проблема. Рассмотрим пример.

```
struct Pair {
    const char* p;
    int val;
};

struct Pair  p2 = {"apple",78};
/* Выделение памяти: */
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair));

pp->p = "pear";   /* Инициализация */
pp->val = 42;
```

Теперь мы не можем написать инструкцию

```
*pp = {"pear", 42}; /* Ошибка: не C и не C++98 */
```

ни в программе на языке C, ни в программе на языке C++. Однако в языке C++ мы могли бы определить конструктор для структуры `Pair` и записать

```
Pair* pp = new Pair("pear", 42);
```

В языке C (но не в языке C++; см. раздел 27.3.4) перед вызовом функции `malloc()` можно не указывать приведение типа, но мы не рекомендуем это делать.

```
int* p = malloc(sizeof(int)*n); /* Избегайте этого */
```

Игнорирование приведения довольно часто встречается в программах, потому что это экономит время и позволяет выявить редкую ошибку, когда программист забывает включить в текст программы заголовочный файл `<stdlib.h>` перед использованием функции `malloc()`. Однако при этом исчезает и визуальный маркер, свидетельствующий о том, что размер памяти подсчитан неправильно.

```
p = malloc(sizeof(char)*m); /* Вероятно, ошибка — нет
                               места для m чисел типа int */
```

Не используйте функции `malloc()/free()` в программах, написанных на языке C++; операторы `new/delete` не требуют приведения типа, выполняют инициализацию (вызывая конструкторы) и очистку (вызывая деструкторы), сообщают об ошибках, связанных с распределением памяти (с помощью исключений), и просто работают быстрее. Не удаляйте объект, размещенный в памяти с помощью функции `malloc()`, выполняя оператор `delete`, и не удаляйте объект, созданный с помощью оператора `new`, вызывая функцию `free()`:

```
int* p = new int[200];
// ...
free(p);     // Ошибка

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q;    // Ошибка
```

Этот код может оказаться вполне работоспособным, но он не является переносимым. Более того, для объектов, имеющих конструкторы и деструкторы, смешение стилей языков C и C++ при управлении динамической памятью может привести к катастрофе.

Для расширения буферов обычно используется функция **realloc()**.

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) { /* Чтение до конца файла       */
      if (count==max-1) {    /* Необходимо расширить буфер */
            max += max;      /* Удвоение размера буфера    */
            p = (char*)realloc(p,max);
            if (p==0) quit();
      }
      p[count++] = c;
}
```

Операции ввода в языке C описаны в разделах 27.6.2 и Б.11.2.

Функция **realloc()** может выделить память на прежнем участке, а может и перенести его содержимое во вновь выделенную область памяти. Даже не думайте применять функцию **realloc()** к области памяти, выделенной с помощью оператора **new**.

Используя стандартную библиотеку языка C++, этот код (грубо) эквивалентен следующему:

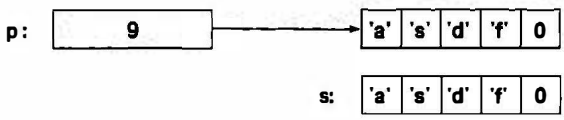
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

Более подробное обсуждение стратегий ввода и распределения памяти можно найти в статье "*Learning Standard C++ as a New Language*" (см. список библиографических ссылок в конце раздела 27.1).

27.5. Строки в стиле C

В языке C строка (в литературе, посвященной языку C++, ее часто называют *C-строкой* (*C-string*) или *строкой в стиле языка C* (*C-style*)) — это массив символов, завершающийся нулевым символом:

```
char* p = "asdf";
char s[ ] = "asdf";
```



В языке C нет функций-членов, невозможно перегружать функции и нельзя определить оператор (такой, как ==) для структур. Вследствие этого для манипулирования строками в стиле языка C необходим набор специальных функций (не членов класса). В стандартных библиотеках языков C и C++ такие функции определены в заголовочном файле **<string.h>**.

```
size_t strlen(const char* s); /* Количество символов в строке */
char* strcat(char* s1,        /* Копирует s2 в конец s1       */
             const char* s2);
int strcmp(const char* s1,    /* лексикографическое сравнение */
           const char* s2);   /* строк                        */
char* strcpy(char* s1,        /* Копирует s2 в s1             */
             const char* s2);
char* strchr(const char *s,   /* Поиск символа c в строке s   */
             int c);
char* strstr(const char *s1,  /* Поиск строки s2 в строке s1  */
             const char *s2);
char* strncpy(char*,          /* Сравнение (<= n символов)    */
              const char*, size_t n);
char* strncat(char*,          /* strcat (<= n символов)       */
              const char, size_t n);
int strncmp(const char*,      /* strcmp (<= n символов)       */
            const char*, size_t n);
```

Это неполный список функций для работы со строками, но он содержит самые полезные и широко используемые функции. Кратко проиллюстрируем их применение.

Мы можем сравнивать строки. Оператор проверки равенства (==) сравнивает значения указателей; стандартная библиотечная функция **strcmp()** сравнивает значения C-строк.

```
const char* s1 = "asdf";
const char* s2 = "asdf";
if (s1==s2) { /* Указывают ли указатели s1 и s2 на один и */
              /* тот же массив? (обычно это нежелательно) */
}

if (strcmp(s1,s2)==0) { /* Хранят ли строки s1 и s2       */
                        /* одни и те же символы?          */
}
```

Функция **strcmp()** может дать три разных ответа. При заданных выше значениях **s1** и **s2** функция **strcmp(s1,s2)** вернет нуль, что означает полное совпадение. Если строка **s1** предшествует строке **s2** в лексикографическом порядке, то функция вернет отрицательное число, а если строка **s1** следует за строкой **s2** в лексикографическом порядке, то функция вернет положительное число. Термин *лексикографический* (lexicographical) означает "как в словаре." Рассмотрим пример.

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0 /* "ape" в словаре предшествует "dodo" */
strcmp("pig","cow") >0 /* "pig" в словаре следует после "cow" */
```

Результат сравнения указателей `s1==s2` не обязательно равен 0 (`false`). Механизм реализации языка может использовать для хранения всех строковых литералов одну и ту же область памяти, поэтому можем получить ответ 1 (`true`). Обычно функция `strcmp()` является верным выбором для сравнения C-строк.

Длину C-строки можно найти с помощью функции `strlen()`:

```
int lgt = strlen(s1);
```

Обратите внимание на то, что функция `strlen()` подсчитывает символы без учета завершающего нулевого символа. В данном случае `strlen(s1)==4`, а строка `"asdf"` занимает в памяти пять байтов. Эта небольшая разница является источником многих ошибок при подсчетах.

Мы можем копировать одну C-строку (включая завершающий нуль) в другую.

```
strcpy(s1,s2); /* Копируем символы из s2 в s1 */
```

Программист должен сам гарантировать, что целевая строка (массив) имеет достаточный размер, чтобы в ней поместились символы исходной строки.

Функции `strncpy()`, `strncat()` и `strncmp()` являются версиями функций `strcpy()`, `strcat()` и `strcmp()`, работающих не более чем с `n` символами, где параметр `n` задается как третий аргумент. Обратите внимание на то, что если в исходной строке окажется более `n` символов, то функция `strncpy()` не будет копировать завершающий нулевой символ и результат копирования не будет корректной C-строкой.

Функции `strchr()` и `strstr()` ищут свой второй аргумент в строке, являющейся первым аргументом и при успешном поиске возвращают указатель на первый символ совпадения (при неудаче возвращается нулевой указатель). Как и функция `find()`, они выполняют поиск в строке слева направо.

Удивительно, как много можно сделать с этими простыми функциями и как легко при этом допустить незаметные ошибки. Рассмотрим простую задачу: конкатенация имени пользователя с адресом, с размещением между ними символа `@`. С помощью класса `std::string` это можно сделать очень просто:

```
string s = id + '@' + addr;
```

С помощью стандартных функций для работы с C-строками этот код можно написать следующим образом:

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
```

```
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz-1]=0;
    return res;
}
```

Все ли мы сделали правильно? И кто вызовет функцию `free()` для строки, которую вернула функция `cat()`?

ПОПРОБУЙТЕ

Протестируйте функцию `cat()`. Почему в первой инструкции мы добавляем число 2? В функции `cat()` мы сделали ошибку, связанную с производительностью и характерную для новичка. Найдите и устраните ее. Мы также "забыли" прокомментировать код. Добавьте соответствующие комментарии, предполагая, что читатель знает стандартные функции для работы с С-строками.

27.5.1. Строки в стиле C и ключевое слово `const`

Рассмотрим следующий пример:

```
char* p = "asdf";
p[2] = 'x';
```

В языке С так писать можно, а в языке C++ — нет. В языке C++ строковый литерал является константой, т.е. неизменяемой величиной, поэтому оператор `p[2]='x'` (который пытается превратить исходную строку в строку `"asxf"`) недопустим. К сожалению, некоторые компиляторы пропускают присваивание указателю `p`, что приводит к проблемам. Если вам повезет, то произойдет ошибка на этапе выполнения программы, но рассчитывать на это не стоит. Вместо этого следует писать так:

```
const char* p = "asdf"; // Теперь вы не сможете записать символ
                        // в строку "asdf" с помощью указателя p
```

Эта рекомендация относится как к языку С, так и к языку C++.

Функция `strchr()` из языка С порождает аналогичную, но даже более трудноуловимую проблему. Рассмотрим пример.

```
/* Ищет в константной строке s символ c (не C++): */
char* strchr(const char* s, int c);

const char aa[] = "asdf";  /* aa  - массив констант          */
char* q = strchr(aa, 'd'); /* Поиск символа 'd'              */
*q = 'x';                  /* Замена 'd' в строке aa на 'x' */
```

Этот код некорректен как в языке C, так и в языке C++, но компиляторы языка C не в состоянии поймать ошибку. Иногда это явление называют *трансмутацией* (transmutation): функция превращает константы в не константы, нарушая разумные предположения о коде.

В языке C++ эта проблема решается с помощью немного измененного объявления стандартной библиотечной функции **strchr()**.

```
// Ищет в константной строке s символ c
char const* strchr(const char* s, int c);

// Ищет в строке s символ c
char* strchr(char* s, int c);
```

То же самое относится и к функции **strstr()**.

27.5.2. Операции над байтами

В далекие темные века (где-то в начале 1980-х годов), еще до изобретения указателя **void***, программисты, работавшие на языках C и C++, для работы с байтами использовали строки. В настоящее время основные стандартные библиотечные функции для работы с памятью имеют параметры типа **void*** и возвращают указатели типа **void***, чтобы предупредить пользователей о непосредственной работе с нетипизированной памятью.

```
/* Копирует n байтов из строки s2 в s1 (подобно strcpy): */
void* memcpy(void* s1, const void* s2, size_t n);

/* Копирует n байтов из строки s2 в строку s1            */
/* Диапазон [s1,s1+n) может перекрываться с [s2,s2+n) ): */
void* memmove(void* s1, const void* s2, size_t n);

/* Сравнивает n байтов из строк s2 и s1 (подобно strcmp):*/
int memcmp(const void* s1, const void* s2, size_t n);

/* Находит символ c (преобразованный в unsigned char)    */
/* среди первых n байтов строки s:                       */
void* memchr(const void* s, int c, size_t n);

/* Копирует символ c (преобразованный в unsigned char) в */
/* каждый из n байтов строки, на которую указывает s:    */
void* memset(void* s, int c, size_t n);
```

Не используйте эти функции в программах на языке C++. В частности, функция **memset()** обычно влияет на гарантии, предоставляемые конструкторами.

27.5.3. Пример: функция strcpy()

Определение функции **strcpy()** представляет собой печально известный пример лаконичного стиля, который допускает язык C (и C++):

```
char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}
```

Объяснение, почему этот код на самом деле копирует С-строку `q` в С-строку `p`, мы оставляем читателям в качестве упражнения.

ПОПРОБУЙТЕ

Является ли корректной реализация функции `strcpy()`? Объясните, почему.

Если вы не можете объяснить, почему, вы не вправе считать себя программистом, работающим на языке С (при этом вы можете быть компетентным в других языках программирования). Каждый язык имеет собственные идиомы, и это одна из идиом языка С.

27.5.4. Вопросы стиля

Мы потихоньку втягиваемся в длинные и часто яростно оспариваемые вопросы стиля, которые, впрочем, часто не имеют большого значения. Мы объявляем указатель следующим образом:

```
char* p;   // p  –  указатель на переменную типа char
```

Мы не приемлем стиль, продемонстрированный ниже:

```
char *p;   /* p  –  нечто, что можно разыменовать, чтобы получить символ */
```

Пробел совершенно игнорируется компилятором, но для программиста он имеет значение. Наш стиль (общепринятый среди программистов на языке C++) подчеркивает тип объявляемой переменной, в то время как альтернативный стиль (общепринятый среди программистов на языке С) делает упор на ее использовании. Мы не рекомендуем объявлять несколько переменных в одной строке.

```
char c, *p, a[177], *f(); /* Разрешено, но может ввести в заблуждение */
```

Такие объявления можно часто встретить в старых программах. Вместо этого объявления следует размещать в нескольких строках, используя свободное место для комментариев и инициализации.

```
char c = 'a'; /* Символ завершения ввода для функции f()  */
char* p = 0;  /* Последний символ, считанный функцией f() */
char a[177];  /* Буфер ввода                              */
char* f();    /* Считывает данные в буфер a; возвращает   */
              /* указатель на первый считанный символ     */
```

Кроме того, не забывайте выбирать осмысленные имена.

27.6. Ввод-вывод: заголовочный файл stdio.h

В языке C нет потоков ввода-вывода **iostream**, поэтому мы используем стандартный механизм ввода-вывода языка C, определенный в заголовочном файле **<stdio.h>**. Эквивалентами потоков ввода и вывода **cin** и **cout** из языка C++ в языке C являются потоки **stdin** и **stdout**. Стандартные средства ввода-вывода языка C и потоки **iostream** могут использоваться в одной и той же программе одновременно (для одних и тех же потоков ввода-вывода), но мы не рекомендуем это делать. Если вам необходимо совместно использовать эти механизмы, хорошенько разберитесь в них (обратите особое внимание на функцию **ios_base::sync_with_stdio()**), используя хороший учебник. (См. также раздел Б.11.)

27.6.1. Вывод

Наиболее популярной и полезной функцией библиотеки **stdio** является функция **printf()**. Основным предназначением функции **printf()** является вывод C-строки.

```
#include<stdio.h>
void f(const char* p)
{
    printf("Hello, World!\n");
    printf(p);
}
```

Это не очень интересно. Намного интереснее то, что функция **printf()** может получать любое количество аргументов и начальную управляющую строку, которая определяет, как вывести дополнительные аргументы. Объявление функции **printf()** в языке C выглядит следующим образом:

```
int printf(const char* format, ... );
```

Многоточие (...) означает "и, возможно, другие аргументы". Мы можем вызвать функцию **printf()**, например, так:

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

Здесь символы **%g** означают "вывести число с плавающей точкой, используя универсальный формат", символы **%s** означают "вывести C-строку", символы **%d** означают "вывести целое число, используя десятичные цифры," а символы **%c** означают "вывести символ". Каждый такой спецификатор формата связан с очередным до поры до времени не используемым аргументом, так что спецификатор **%g** выводит на экран значение переменной **d**; **%s** — значение переменной **s**, **%d** — значение переменной **i**, а

%c — значение переменной **ch**. Полный список форматов функции **printf()** приведен в разделе Б.11.2.

К сожалению, функция **printf()** не является безопасной с точки зрения типов. Рассмотрим пример.

```
        char a[] = { 'a', 'b' };          /* Нет завершающего нуля */
void f2(char* s, int i)
{
    printf("goof %s\n", i);       /* Неперехваченная ошибка */
    printf("goof %d: %s\n", i);   /* Неперехваченная ошибка */
    printf("goof %s\n", a);       /* Неперехваченная ошибка */
}
```

Интересен эффект последнего вызова функции **printf()**: после символов из массива **a[]** она выводит на экран байты участка памяти, следующего за элементом **a[1]**, пока не встретит нулевой символ. Такой вывод может состоять из довольно большого количества символов.

Отсутствие проверки типов является одной из причин, по которым мы предпочитаем потоки **iostream**, несмотря на то что стандартный механизм ввода-вывода, описанный в библиотеке **stdio** языков C и C++, работает одинаково. Другой причиной является то, что функции из библиотеки **stdio** не допускают расширения: мы не можем расширить функцию **printf()** так, чтобы она выводила на экран значения переменных вашего собственного типа. Для этого можно использовать потоки **iostream**. Например, нет никакого способа, который позволил бы вам определить свой спецификатор формата **%Y** для вывода некоторой структуры **struct Y**.

Существует полезная версия функции **printf()**, принимающая в качестве первого аргумента дескриптор файла.

```
int fprintf(FILE* stream, const char* format, ... );
```

Рассмотрим пример.

```
fprintf(stdout,"Hello, World!\n"); // Идентично printf("Hello,World!\n");
FILE* ff = fopen("My_file","w");   // Открывает файл My_file для записи
fprintf(ff,"Hello, World!\n");     // Запись "Hello,World!\n" в файл My_file
```

Дескрипторы файлов описаны в разделе 27.6.3.

27.6.2. Ввод

Наиболее популярные функции из библиотеки **stdio** включают следующие.

```
int scanf(const char* format, ... );      /* Форматный ввод
                                             из потока stdin */
int getchar(void);          /* Ввод символа из потока stdin  */
int getc(FILE* stream);     /* Ввод символа из потока stream */
char* gets(char* s);        /* Ввод символов из потока stdin */
```

Простейший способ считать строку символов — это использовать функцию **gets()**:

```
char a[12];
gets(a); /* Чтение в массив символов a до символа '\n' */
```

Никогда так не поступайте! Считайте, что функция **gets()** отравлена. Вместе со своей ближайшей "родственницей" — функцией **scanf("%s")** — функция **gets()** является мишенью для примерно четверти успешных хакерских атак. Она порождает много проблем, связанных с безопасностью. Откуда в тривиальном примере, приведенном выше, вы можете знать, что до следующего символа новой строки будет введено не более 11 символов? Вы не можете этого знать. Следовательно, функция **gets()** почти наверняка приведет к повреждению памяти (байтов, находящихся за буфером), а повреждение памяти является основным инструментом для хакерских атак. Не считайте, что можете угадать максимальный размер буфера, достаточный на все случаи жизни. Возможно, что "субъект" на другом конце потока ввода — программа, не соответствующая вашим критериям разумности.

Функция **scanf()** считывает данные с помощью формата точно так же, как и функция **printf()**. Как и функция **printf()**, она может быть очень удобной.

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* Считываем данные в переменные, передаваемые в виде указателей: */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* Спецификатор %s пропускает начальные пробельные символы
       и завершает ввод на очередном пробельном символе */
}
```

Как и функция **printf()**, функция **scanf()** является небезопасной с точки зрения типов. Символы формата и аргументы (все являющиеся указателями) должны точно соответствовать друг другу, иначе во время выполнения программы будут происходить странные вещи. Обратите также внимание на то, что считывание данных в строку **s** с помощью спецификатора **%s** может привести к переполнению. Никогда не используйте вызовы **gets()** и **scanf("%s")**!

Но как же безопасно ввести символы? Мы можем использовать вид формата **%s**, устанавливающий предел количества считываемых символов:

```
char buf[20];
scanf("%19s",buf);
```

Нам требуется участок памяти, заканчивающийся нулевым символом (он вводится функцией **scanf()**), поэтому 19 — это максимальное количество символов, которое можно считать в массив **buf**. Однако этот способ не отвечает на вопрос, что делать, если некто введет больше 19 символов. Лишние символы останутся в потоке ввода и будут обнаружены при следующей попытке ввода.

Проблема с функцией **scanf()** означает, что часто благоразумнее и легче использовать функцию **getchar()**. Типичный ввод символов с помощью функции **getchar()** выглядит следующим образом:

```
while((x=getchar())!=EOF) {
    /* ... */
}
```

Макрос **EOF**, описанный в **stdio.h**, означает "конец файла"; см. также раздел 27.4.

Альтернативы функций **scanf("%s")** и **gets()** в стандартной библиотеке языка C++ от этих проблем не страдают:

```
string s;
cin >> s;        // Считываем слово
getline(cin,s); // Считываем строку
```

27.6.3. Файлы

В языке C (как и в C++) файлы можно открыть с помощью функции **fopen()**, а закрыть — с помощью функции **fclose()**. Эти функции, вместе с представлением дескриптора файлов **FILE** и макросом **EOF** (конец файла), описаны в заголовочном файле **<stdio.h>**.

```
FILE *fopen(const char* filename, const char* mode);
int   fclose(FILE *stream);
```

В основном мы используем файлы примерно так:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r"); /* Открываем файл fn для чтения */
    FILE* fo = fopen(fn2,"w"); /* Открываем файл fn для записи */

    if (fi == 0) error("Невозможно открыть файл для ввода");
    if (fo == 0) error("Невозможно открыть файл для вывода");

    /* Чтение из файла с помощью функций ввода из
       библиотеки stdio, например getc() */
    /* Запись в файл с помощью функций вывода из
       библиотеки stdio, например fprintf() */

    fclose(fo);
    fclose(fi);
}
```

Учтите: в языке C нет исключений, потому вы не можете гарантировать закрытие файлов при обнаружении ошибок.

27.7. Константы и макросы

В языке C **const** не является константой времени компиляции.

```
const int max = 30;
const int x;      /* Неинициализированная константа:          */
                  /* OK в C (ошибка в C++)                    */

void f(int v)
{
    int a1[max]; /* Ошибка: граница массива не является      */
                 /* константой (OK в языке C++). (max не     */
                 /* допускается в константном выражении!)    */
    int a2[x];   /* Ошибка: граница массива не является      */
                 /* константой */

    switch (v) {
    case 1:
        /* ... */
        break;
    case max:    /* Ошибка: метка раздела case не является */
                 /* константой (OK в языке C++)            */
        /* ... */
        break;
    }
}
```

Технически причина этого в том, что в языке C (но не в языке C++) **const** неявно доступна из других единиц трансляции:

```
/* Файл x.c: */
const int x;        /* Инициализирована в другом месте */

/* Файл xx.c: */
const int x = 7;    /* Настоящее определение           */
```

В языке C++ такой исходный текст приведет к тому, что в разных файлах будут существовать два разных объекта с одним и тем же именем **x**. Вместо ключевого слова **const** для представления символических констант программисты на языке C обычно используют макросы:

```
#define MAX 30

void f(int v)
{
    int a1[MAX]; /* OK */

    switch (v) {
    case 1:
```

```
        /* ... */
        break;
    case MAX:     /* OK */
        /* ... */
        break;
    }
}
```

Имя макроса **MAX** заменяется символами 30, представляющими собой значение этого макроса; иначе говоря, количество элементов массива **a1** равно 30, а меткой второго раздела **case** является число 30. По общепринятому соглашению имя макроса **MAX** состоит только из прописных букв. Это позволяет минимизировать ошибки, вызываемые макросами.

27.8. Макросы

Берегитесь макросов: в языке C нет по-настоящему эффективных способов избежать макросов, но их использование имеет серьезные побочные эффекты, поскольку они не подчиняются обычным правилам разрешения области видимости и типов, принятым в языках C и C++. Макросы — это вид текстуальной подстановки. (См. также раздел A.17.2.)

Как защититься от потенциальных проблем, связанных с макросами, не отказываясь от них навсегда (и не прибегая к альтернативам, предусмотренным в языке C++?)

- Присваивайте всем макросам имена, состоящие только из прописных букв: **ALL_CAPS**.
- Не присваивайте имена, состоящие только из прописных букв, объектам, которые не являются макросами.
- Никогда не давайте макросам короткие или "изящные" имена, такие как **max** или **min**.
- Надейтесь, что остальные программисты следуют этим простым и общеизвестным правилам.

В основном макросы применяются в следующих случаях:

- определение "констант";
- определение "функциеподобных" конструкций;
- улучшение синтаксиса;
- управление условной компиляцией.

Кроме того, существует большое количество менее известных ситуаций, в которых могут использоваться макросы.

Мы считаем, что макросы используются слишком часто, но в программах на языке C у них нет разумных и полноценных альтернатив. Их трудно избежать даже в программах на языке C++ (особенно если вам необходимо написать программу, которая должна подходить для очень старых компиляторов или выполняться на платформах с необычными ограничениями).

Мы приносим извинения читателям, считающим, что приемы, которые будут описаны ниже, являются "грязными трюками", и полагают, что о них лучше не говорить в приличном обществе. Однако мы думаем, что программирование должно учитывать реалии и что эти (очень простые) примеры правильного и неправильного использования макросов сэкономят часы страданий для новичков. Незнание макросов не приносит счастья.

27.8.1. Макросы, похожие на функции

Рассмотрим типичный макрос, напоминающий функцию.

```
#define MAX(x, y)  ((x)>=(y)?(x):(y))
```

Мы используем прописные буквы в имени **MAX**, чтобы отличить его от множества функций с именем **max** (в разных программах). Очевидно, что этот макрос сильно отличается от функции: у него нет типов аргументов, нет тела, нет инструкции **return** и так далее, и вообще, зачем здесь так много скобок? Проанализируем следующий код:

```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

Он разворачивается в такой фрагмент программы:

```
int aa = ((1)>=(2)?(1):(2));
double dd = ((aa++)>=(2)?(aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

Если бы всех этих скобок не было, то последняя строка выглядела бы следующим образом.

```
char cc = dd>=aa?dd:aa+2;
```

Иначе говоря, переменная **cc** могла бы легко получить другое значение, которого вы не ожидали, исходя из определения макроса. Определяя макрос, не забывайте заключать в скобки каждый аргумент, входящий в выражение.

С другой стороны, скобки не всегда могут спасти нас от второго варианта развертывания. Параметр макроса **x** получил значение **aa++**, а поскольку переменная **x** в макросе **MAX** используется дважды, инкремент переменной **a** также может быть выполнен дважды. Не передавайте макросу аргументы, имеющие побочные действия.

Какой-то "гений" определил приведенный ниже макрос и поместил его в широко используемый заголовочный файл. К сожалению, он назвал его **max**, а не **MAX**, поэтому, когда в стандартном заголовочном файле языка C++ объявляется функция

```
template<class T> inline T max(T a, T b) { return a<b?b:a; }
```

имя **max** разворачивается с аргументами **T a** и **T b**, и компилятор видит строку

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b))
          { return a<b?b:a; }
```

Сообщения об ошибке, выдаваемые компилятором, достаточно "интересны", но не слишком информативны. В случае опасности определение макроса можно отменить:

```
#undef max
```

К счастью, этот макрос не привел к большим неприятностям. Однако в широко используемых заголовочных файлах имеются десятки тысяч макросов; вы не можете отменить их все, не вызвав хаоса.

Не все параметры макросов используются как выражения. Рассмотрим следующий пример:

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

Это реальный пример, который может оказаться очень полезным для предотвращения ошибок, возникающих из-за несоответствия желаемого типа выделяемой памяти и использования оператора **sizeof**.

```
double* p = malloc(sizeof(int)*10); /* Похоже на ошибку */
```

К сожалению, написать макрос, который заодно позволял бы выявить исчерпание памяти, — задача нетривиальная. Это можно сделать, определив в соответствующем месте программы переменную **error_var** и функцию **error()**.

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
                    (error_var==0)                       \
                  ?(error("ошибка выделения памяти"),0)  \
                  :error_var)
```

Строки, завершающиеся символом \, не являются опечатками; это просто способ разбить определение макроса на несколько строк. Когда мы пишем программы на языке C++, то предпочитаем использовать оператор **new**.

27.8.2. Синтаксические макросы

Можно определить макрос, который приводит текст исходного кода в приятный для вас вид. Рассмотрим пример.

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```

Мы ***категорически*** против этого. *Многие* программисты пытались делать такие вещи. Они (и те, кому пришлось поддерживать такие программы) пришли к следующим выводам.

- Многие программисты не разделяют ваших взглядов на то, что считать лучшим синтаксисом.
- Улучшенный синтаксис является нестандартным и неожиданным; остальные программисты будут сбиты с толку.
- Использование улучшенного синтаксиса может вызвать непонятные ошибки компиляции.
- Текст программы, который вы видите перед собой, не совпадает с текстом, который видит компилятор, и компилятор сообщает об ошибках, используя свой словарный запас, а не ваш.

Не пишите синтаксические макросы для того, чтобы улучшить внешний вид своего кода. Вы и ваши лучшие друзья могут считать его превосходным, но опыт показывает, что вы окажетесь в крошечном меньшинстве среди более крупного сообщества программистов, поэтому кому-то придется переписать ваш код (если он сможет просуществовать до этого момента).

27.8.3. Условная компиляция

Представьте себе, что у вас есть два варианта заголовочного файла, например один — для операционной системы Linux, а другой — для операционной системы Windows. Как выбрать правильный вариант в вашей программе? Вот как выглядит общепринятое решение этой задачи:

```
#ifdef WINDOWS
     #include "my_windows_header.h"
#else
     #include "my_linux_header.h"
#endif
```

Теперь, если кто-нибудь уже определил `WINDOWS` до того, как компилятор увидел этот код, произойдет следующее:

```
#include "my_windows_header.h"
```

В противном случае будет включен другой заголовочный файл.

```
#include "my_linux_header.h"
```

Директива `#ifdef WINDOWS` не интересуется, что собой представляет макрос `WINDOWS`; она просто проверяет, был ли он определен раньше.

Большинство крупных систем (включая все варианты операционных систем) определяют макросы, которые вы можете проверить. Например, можно выяснить, как компилируется ваша программа: как программа на языке C++ или программа на языке C:

```
#ifdef __cplusplus
    // Компилятор C++
#else
    /* Компилятор C */
#endif
```

Аналогичная конструкция, которую часто называют *защитой включения* (include guard), обычно используется для предотвращения повторного включения заголовочного файла.

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* Информация заголовочного файла */
#endif
```

Директива **#ifndef** проверяет, не было ли указанное имя определено раньше; т.е. директива **#ifndef** противоположна директиве **#ifdef**. С логической точки зрения макросы, использующиеся для контроля файлов с исходными текстами, существенно отличаются от макросов, используемых для модификации исходного кода. Они просто используют одинаковый базовый механизм для выполнения своих функций.

27.9. Пример: интрузивные контейнеры

Контейнеры из стандартной библиотеки языка C++, такие как **vector** и **map**, являются неинтрузивными; иначе говоря, им не требуются данные типов, используемых в качестве элементов. Это позволяет обобщить их для практически всех типов (как встроенных, так и пользовательских), лишь бы эти типы могли быть скопированы. Существует и другая разновидность контейнеров — *интрузивные контейнеры* (intrusive container), популярные как в языке C, так и в C++. Для того чтобы проиллюстрировать использование структур, указателей и динамической памяти с стиле C, мы воспользуемся неинтрузивным списком.

Определим двухсвязный список с девятью операциями.

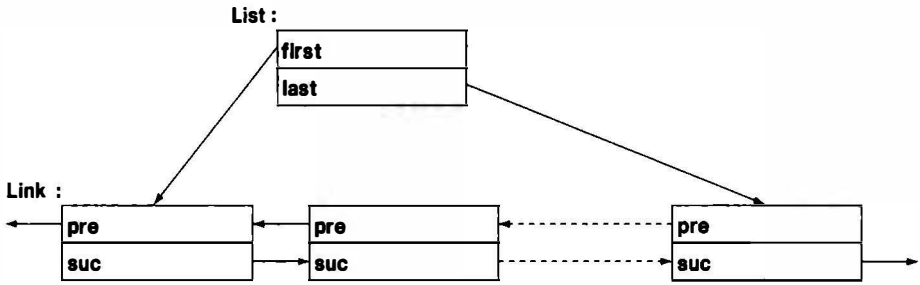
```
void init(struct List* lst);      /* Инициализирует lst пустым */
struct List* create();            /* Создает пустой список     */
                                  /* в динамической памяти     */
void clear(struct List* lst);     /* Удаляет все элементы lst  */
void destroy(struct List* lst);   /* Удаляет все элементы lst, */
                                  /* а затем удаляет сам lst   */
```

```
void push_back(struct List* lst, /* Добавляет элемент p       */
               struct Link* p);  /* в конец списка lst        */
void push_front(struct List*,    /* Добавляет элемент p       */
                struct Link* p); /* в начало списка lst       */

void insert(struct List* lst,    /* Вставляет элемент q       */
            struct Link* p,      /* перед элементом p         */
            struct Link* q);     /* в список lst              */
struct Link* erase(
            struct List* lst,    /* Удаляет элемент p         */
            struct Link* p);     /* из списка lst             */

struct Link* advance(            /* Возвращает элемент, нахо- */
            struct Link* p,      /* дящийся за n элементов до */
            int n);              /* или после элемента p      */
```

Идея заключается в том, чтобы определить эти операции так, чтобы их пользователям было достаточно использовать только указатели **List*** и **Link***. Это значит, что реализации этих функций можно кардинально изменять, не влияя на работу их пользователей. Очевидно, что выбор имен был сделан под влиянием библиотеки STL. Структуры **List** и **Link** можно определить очевидным и тривиальным образом.

```
struct List {
    struct Link* first;
    struct Link* last;
};

struct Link {       /* Узел двусвязного списка */
    struct Link* pre;
    struct Link* suc;
};
```

Приведем графическое представление контейнера **List**:



В наши намерения не входит демонстрация изощренных методов представления или алгоритмов, поэтому ни один из них на рисунке не показан. Тем не менее обратите внимание на то, что мы не упоминаем о данных, которые хранятся в узлах (элементах списков). Оглядываясь на функции-члены этой структуры, мы видим, что сделали что-то подобное определению пары абстрактных классов **Link** и **List**. Данные для хранения в узлах

будут предоставлены позднее. Указатели **Link*** и **List*** иногда называют непрозрачными типами (opaque types); т.е., передавая указатели **Link*** и **List*** функциям, мы получаем возможность манипулировать элементами контейнера **List**, ничего не зная о внутреннем устройстве структур **Link** и **List**.

Для реализации наших функций для работы с **List** сначала включаем некоторые заголовочные файлы стандартной библиотеки.

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

В языке C нет пространств имен, поэтому можно не беспокоиться о декларациях или директивах **using**. С другой стороны, следовало бы побеспокоиться о слишком коротких и слишком популярных именах (**Link**, **insert**, **init** и т.д.), поэтому в реальных программах такой набор функций использовать нельзя.

Инициализация тривиальна, обратите только внимание на использование **assert()**.

```
void init(struct List* lst) /* Инициализируем *lst   */
{                           /* пустым списком        */
    assert(lst);
    lst->first = lst->last = 0;
}
```

Мы решили не связываться с обработкой ошибок, связанных с некорректными указателями на списки во время выполнения программы. Используя макрос **assert()**, мы во время выполнения программы просто получим сообщение о системной ошибке, если указатель на список окажется нулевым. Эта "системная ошибка" даст нам имя файла и номер строки, в которой находится макрос **assert()** с нарушенным условием, переданным ему в качестве аргумента. **assert()** — это макрос, определенный в заголовочном файле **<assert.h>**, а проверка доступна только в режиме отладки. В отсутствие исключений нелегко понять, что следует делать с некорректными указателями.

Функция **create()** просто создает список **List** в динамической памяти. Она напоминает комбинацию конструктора (функция **init()** выполняет инициализацию) и оператора **new** (функция **malloc()** выделяет память).

```
struct List* create() /* Создает пустой список */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

Функция **clear()** предполагает, что все узлы созданы в динамической памяти, и освобождает выделенную для них память с помощью функции **free()**.

```
void clear(struct List* lst) /* Освобождает все элементы lst */
{
    assert(lst);
    {
        struct Link* curr = lst->first;
        while(curr) {
            struct Link* next = curr->suc;
            free(curr);
            curr = next;
        }
        lst->first = lst->last = 0;
    }
}
```

Обратите внимание на способ, с помощью которого мы обходим список, используя член **suc** класса **Link**. Мы не можем получить безопасный доступ к члену объекта после его удаления с помощью функции **free()**, поэтому ввели переменную **next**, в которой храним информацию о своей позиции в контейнере **List**, одновременно удаляя объекты класса **Link** с помощью функции **free()**.

Если не все объекты структуры **Link** находятся в динамической памяти, функцию **clear()** лучше не вызывать, иначе она вызовет разрушение памяти.

Функция **destroy()**, по существу, противоположна функции **create()**, т.е. она представляет собой сочетание деструктора и оператора **delete**.

```
void destroy(struct List* lst) /* Удаляет все элементы lst;    */
{                              /* затем удаляет сам список lst */
    assert(lst);
    clear(lst);
    free(lst);
}
```

Обратите внимание на то, что перед вызовом функции очистки памяти (деструктора) мы не делаем никаких предположений об элементах, представленных в виде узлов списка. Эта схема не является полноценной имитацией методов или обобщенности языка C++, но она и не может, и, вероятно, и не должна быть таковой.

Функция **push_back()**— добавление узла **Link** в конец списка — вполне очевидна.

```
void push_back(struct List* lst, /* Добавляет элемент p    */
               struct Link* p)   /* в конец списка lst     */
{
    assert(lst);
    {
```

```
        struct Link* last = lst->last;
        if (last) {
            last->suc = p;          /* Добавляет p после last */
            p->pre = last;
        }
        else {
            lst->first = p;         /* p — первый элемент     */
            p->pre = 0;
        }
        lst->last = p;              /* p — новый последний    */
        p->suc = 0;                 /* элемент                */
    }
}
```

Весь этот код было бы трудно написать, не нарисовав схему из нескольких прямоугольников и стрелок. Обратите внимание, что мы забыли рассмотреть вариант, когда аргумент `p` равен нулю. Передайте нуль вместо указателя на узел, и ваша программа даст сбой. Этот код нельзя назвать совершенно неправильным, но он не соответствует промышленным стандартам. Его цель — проиллюстрировать общепринятые и полезные методы (а также распространенные недостатки и ошибки).

Функцию `erase()` можно было бы написать следующим образом:

```
struct Link* erase(struct List* lst, struct Link* p)
/*
    Удаляет узел p из списка lst; возвращает
    указатель на узел, расположенный после узла p
*/
{
    assert(lst);
    if (p==0) return 0;          /* Можно вызывать erase(0)    */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc;/* Преемник становится первым */
            p->suc->pre = 0;
            return p->suc;
        }
        else {
            lst->first =
                lst->last = 0;  /* Список становится пустым   */
            return 0;
        }
     }
     else if (p == lst->last) {
        if (p->pre) {
            lst->last = p->pre; /* Предшественник становится  */
            p->pre->suc = 0;    /* последним                  */
        }
        else {
            lst->first =
```

```
                lst->last = 0;  /* Список становится пустым   */
            return 0;
        }
    }
    else {
        p->suc->pre = p->pre;
        p->pre->suc = p->suc;
        return p->suc;
    }
}
```

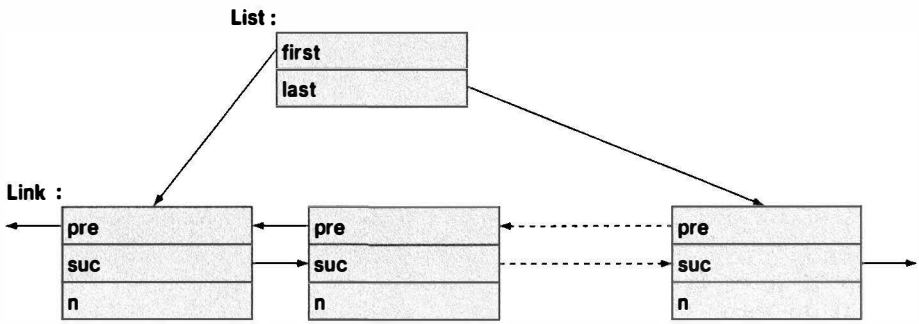
Остальные функции читатели могут написать в качестве упражнения, поскольку для нашего (очень простого) теста они не нужны. Однако теперь мы должны разрешить основную загадку этого проекта: где находятся данные в элементах списка? Как реализовать простой список имен, представленных в виде C-строк? Рассмотрим следующий пример:

```
struct Name {
    struct Link lnk; /* Требуется для операций List */
    char* n;         /* Строка имени                */
};
```

До сих пор все хорошо, хотя остается загадкой, как мы можем использовать этот член **Link**? Но поскольку нам известно, что структура **List** хранит узлы **Link** в динамической памяти, мы написали функцию, создающую объекты структуры **Name** в динамической памяти.

```
struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->n = n;
    return p;
}
```

Эту ситуацию можно проиллюстрировать следующим образом.



Попробуем использовать эту структуру.

```
int main()
{
    int count = 0;
```

```
    struct List names;          /* Создаем список     */
    struct List* curr;
    init(&names);

    /* Создаем несколько Name и добавляем их в список: */
    push_back(&names,(struct Link*)make_name("Norah"));
    push_back(&names,(struct Link*)make_name("Annemarie"));
    push_back(&names,(struct Link*)make_name("Kris"));
    /* Удаляем второе имя (с индексом 1): */
    erase(&names,advance(names.first,1));
    curr = names.first;         /* Выводим все имена */
    for (; curr!=0; curr=curr->suc) {
        count++;
        printf("element %d: %s\n", count,
               ((struct Name*)curr)->n);
    }
}
```

Вообще-то, мы "схимичили". Мы использовали приведение типа, чтобы работать с указателем типа **Name*** как с указателем типа **Link***. Таким образом, пользователь знает о "библиотечном типе" **Link**. Однако "библиотека" не знает о "прикладном типе" **Name**. Это допустимо? Да, допустимо: в языке C (и C++) можно интерпретировать указатель на структуру **struct** как указатель на ее первый элемент, и наоборот.

Очевидно, что этот пример без каких-либо изменений можно скомпилировать с помощью компилятора языка C++.

ПОПРОБУЙТЕ

Программисты, работающие на языке C++, разговаривая с программистами, работающими на языке C, рефреном повторяют: "Все, что делаешь ты, я могу сделать лучше!" Перепишите пример с контейнером **List** на языке C++, продемонстрировав, что это можно сделать короче и проще без замедления программы или увеличения объектов.

Задание

1. Напишите программу "Hello World!" на языке C, скомпилируйте ее и выполните.
2. Определите две переменные, хранящие строки "Hello" и "World!" соответственно; конкатенируйте их с пробелом между ними и выведите в виде строки **Hello World!**.
3. Определите функцию на языке C, получающую параметр **p** типа **char*** и параметр **x** типа **int**, и выведите на печать их значения в следующем формате: **p is "foo" and x is 7**. Вызовите эту функцию для нескольких пар аргументов.

Контрольные вопросы

В следующих вопросах под языком C подразумевается язык стандарта ISO C89.

1. Является ли язык C++ подмножеством языка C?
2. Кто изобрел язык C?
3. Назовите один из лучших учебников по языку C.
4. В какой организации были изобретены языки C и C++?
5. Почему язык C++ (почти) совместим с языком C?
6. Почему язык C++ только *почти* совместим с языком C?
7. Перечислите десять особенностей языка C++, отсутствующих в языке C.
8. Какой организации "принадлежат" языки C и C++?
9. Перечислите шесть компонентов стандартной библиотеки языка C++, которые не могут использоваться в языке C.
10. Какие компоненты стандартной библиотеки языка C можно использовать в языке C++?
11. Как обеспечить проверку типов аргументов функций в языке C?
12. Какие свойства языка C++, связанные с функциями, отсутствуют в языке C? Назовите по крайней мере три из них. Приведите примеры.
13. Как вызвать функцию, написанную на языке C, в программе, написанной на языке C++?
14. Как вызвать функцию, написанную на языке C++, в программе, написанной на языке C?
15. Какие типы в языках C и C++ совместимы в смысле размещения в памяти? Приведите примеры.
16. Что такое дескриптор структуры?
17. Перечислите двадцать ключевых слов языка C++, которые не являются ключевыми словами языка C.
18. Является ли инструкция `int x;` определением в языке C++? А в языке C?
19. В чем заключается приведение в стиле языка C и чем оно опасно?
20. Что собой представляет тип `void*` и чем он отличается в языках C и C++?
21. Чем отличаются перечисления в языках C и C++?
22. Что надо сделать в программе на языке C, чтобы избежать проблем, связанных с совпадением широко распространенных имен?

23. Назовите три наиболее широко используемые функции для работы с динамической памятью в языке C.
24. Как выглядит определение C-строки?
25. Чем отличаются оператор == и функция `strcmp()` для C-строк?
26. Как скопировать C-строки?
27. Как определить длину C-строки?
28. Как скопировать большой массив целых чисел типа `int`?
29. Назовите преимущества и недостатки функции `printf()`.
30. Почему никогда не следует использовать функцию `gets()`? Что следует использовать вместо нее?
31. Как открыть файл для чтения в программе на языке C?
32. В чем заключается разница между константами (`const`) в языках C и C++?
33. Почему мы не любим макросы?
34. Каковы распространенные применения макросов?
35. Что такое "защита включения"?

Термины

`#define`	`malloc()`	интрузивный контейнер
`#ifdef`	`printf()`	лексикографический порядок
`#ifndef`	`scanf()`	макрос
Bell Labs	`strcpy()`	непрозрачный тип
C/C++	`void`	перегрузка
C-строки	`void*`	приведения в стиле C
`FILE`	Брайан Керниган	связь
`fopen()`	Деннис Ритчи	совместимость
K&R	дескриптор структуры	форматная строка

Упражнения

Для этих упражнений может оказаться полезным скомпилировать все программы как с помощью компилятора языка C, так и компилятора языка C++. Если использовать только компилятор языка C++, можно случайно использовать свойства, которых нет в языке C. Если вы используете только компилятор языка C, то ошибки, связанные с типами, могут остаться незамеченными

1. Реализуйте варианты функций `strlen()`, `strcmp()` и `strcpy()`.
2. Завершите пример с интрузивным контейнером `List` из раздела 27.9 и протестируйте каждую его функцию.

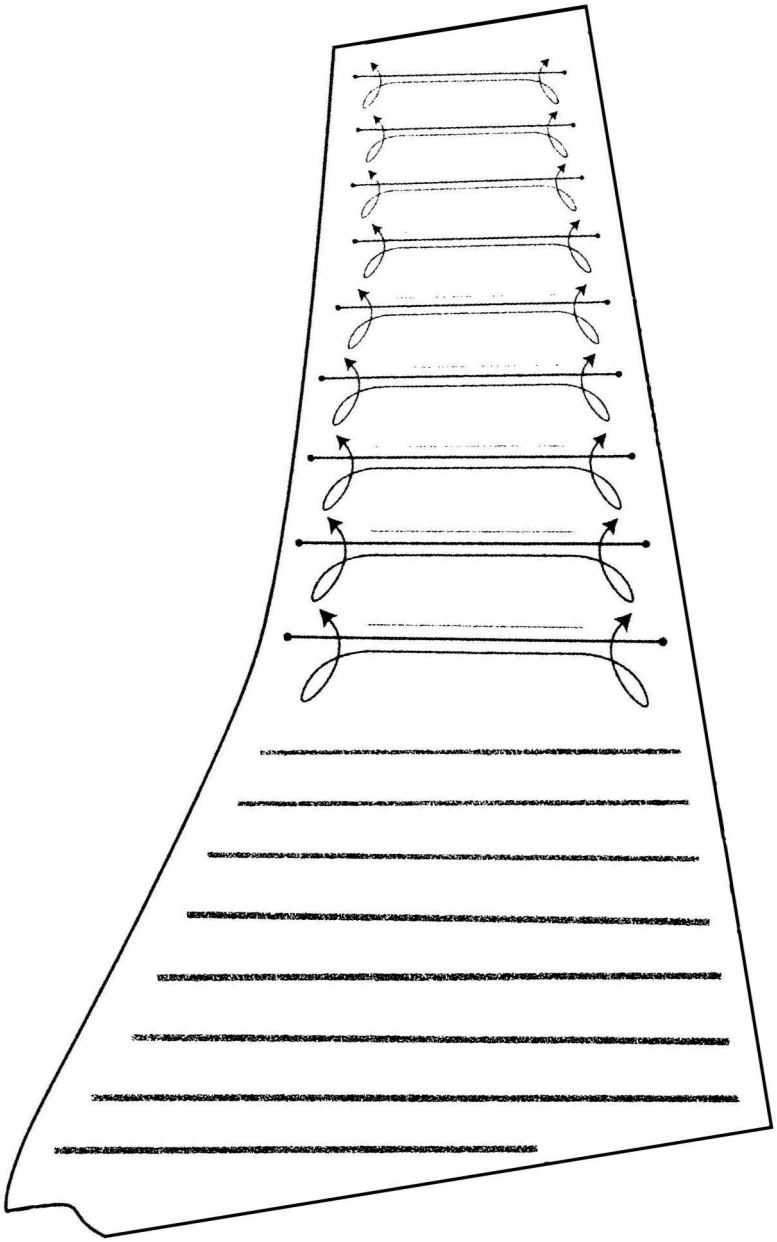
3. Усовершенствуйте пример с интрузивным контейнером `List` из раздела 27.9 по своему усмотрению. Предусмотрите перехват и обработку как можно большего количества ошибок. При этом можно изменять детали определений структур, использовать макросы и т.д.
4. Если вы еще на переписали пример с интрузивным контейнером `List` из раздела 27.9 на языке C++, сделайте это и протестируйте каждую функцию.
5. Сравните результаты упр. 3 и 4.
6. Измените представление структур `Link` и `List` из раздела 27.9 без изменения пользовательского интерфейса, предоставляемого функциями. Разместите узлы в массиве и предусмотрите члены `first`, `last`, `pre` и `suc` типа `int` (индексы массива).
7. Назовите преимущества и недостатки интрузивных контейнеров по сравнению с неинтрузивными контейнерами из стандартной библиотеки языка C++. Составьте списки аргументов "за" и "против" этих контейнеров.
8. Какой лексикографический порядок принят на вашем компьютере? Выведите на печать каждый символ вашей клавиатуры и ее целочисленный код; затем выведите на печать символы в порядке, определенном их целочисленными кодами.
9. Используя только средства языка C, включая его стандартную библиотеку, прочитайте последовательность слов из потока `stdin` и выведите ее в поток `stdout` в лексикографическом порядке. *Указание*: функция сортировки в языке C называется `qsort()`; найдите ее описание. В качестве альтернативы вставляйте слова в упорядоченный список по мере его считывания. В стандартной библиотеке языка C списка нет.
10. Составьте список свойств языка C, заимствованных у языка C++ или C с классами (раздел 27.1).
11. Составьте список свойств языка C, не заимствованных у языка C++.
12. Реализуйте таблицу поиска (C-строка, `int`) с операциями `find(struct table*, const char*)`, `insert(struct table*, const char*, int)` и `remove(struct table*, const char*)`. Эту таблицу можно представить в виде массива структур или пары массивов (с элементами `const char*[]` и `int*`); выбор за вами. Выберите типы возвращаемых значений для ваших функций. Документируйте ваши проектные решения.
13. Напишите программу на языке C, которая является эквивалентом инструкций `string s; cin>>s;`. Иначе говоря, определите операцию ввода, которая считывала бы произвольно длинную последовательность символов, завершающуюся пробельным символом, в массив символов, завершающийся нулевым символом;

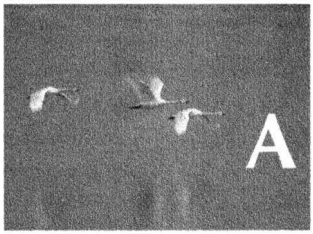
14. Напишите функцию, принимающую в качестве входных данных массив целых чисел типа `int` и находящую наименьший и наибольший его элементы. Она также должна вычислять медиану и среднее значение. Используйте в качестве возвращаемого значения структуру, хранящую результаты работы.
15. Сымитируйте одиночное наследование в языке С. Пусть каждый "базовый класс" содержит указатель на массив указателей на функции (для моделирования виртуальных функций как обычных функций, получающих указатель на объект "базового класса" в качестве своего первого аргумента) (см. раздел 27.2.3). Реализуйте создание производного класса из базового, делая базовый класс типом первого члена производного класса. Для каждого класса соответствующим образом инициализируйте массив виртуальных функций. Для проверки реализуйте вариант старого примера с классом `Shape`, функция `draw()` в котором в базовом и производном классах просто выводит имя своего класса. Используйте только средства и библиотеку, доступные в стандартном С.
16. Воспользуйтесь макросами для упрощения обозначений из предыдущего упражнения.

Послесловие

Мы уже упоминали, что не все вопросы совместимости решены наилучшим образом. Тем не менее существует множество программ на языке С (миллиарды строк кода), написанных кем-то, где-то и когда-то. Если вам придется читать и писать такие программы, данная глава подготовит вас к этому. Лично мы предпочитаем язык С++ и в этой главе частично объяснили, почему. Пожалуйста, отнеситесь серьезно к примеру с интрузивным списком `List` — и интрузивные списки, и непрозрачные типы являются важной и мощной технологией (как в языке С, так и в языке С++).

Часть V
Приложения





Обзор языка

Будьте осторожны со своими желаниями — они могут исполниться.

Народная мудрость

В этом приложении кратко изложены основные сведения о ключевых элементах языка C++. Оно имеет очень избирательный характер и предназначено для новичков, желающих узнать немного больше, чем написано в книге. Цель этого приложения — краткость, а не полнота.

A.1. Общие сведения

Это приложение является справочником, не предназначенным для последовательного чтения от начала до конца, как обычная глава. В нем (более или менее) систематично описаны ключевые элементы языка C++. Впрочем, это не полный справочник, а всего лишь его конспект. Приложение посвящено тем вопросам, которые чаще всего задают студенты. Как правило, для того чтобы получить более полный ответ, читателям придется прочесть соответствующие главы. Настоящее приложение нельзя считать

эквивалентом стандарта по точности изложения и терминологии. Вместо этого мы сделали акцент на доступность изложения. Более полную информацию читатели смогут найти в книге Страуструпа *The C++ Programming Language*. Определение языка C++ изложено в стандарте ISO C++, но этот документ не подходит для новичков. Впрочем, он для них и не был предназначен. Не забудьте о возможности использовать документацию, имеющуюся в сети. Если вы будете заглядывать в приложение, читая первые главы, то многое вам покажется непонятным. Читая остальные главы, вы постепенно во всем разберетесь.

Возможности стандартной библиотеки описаны в приложении Б.

Стандарт языка C++ определен комитетом, работающим под эгидой ISO (International Organization for Standardization — Международная организация по стандартизации) в сотрудничестве с национальными стандартными комитетами, такими как INCITS (США), BSI (Великобритания) и AFNOR (Франция). Действующим стандартом считается документ ISO/IEC 14882:2011 *Standard for Programming Language C++*.

А.1.1. Терминология

Стандарт языка C++ определяет, что такое программа на языке C++ и что означают разные конструкции.

- *Соответствие стандарту*. Программа, написанная на языке C++ в соответствии со стандартом, называется *соответствующей стандарту* (conforming), *правильной* или *корректной*.
- *Зависимость от реализации*. Программа может зависеть (и обычно зависит) от некоторых свойств (таких, как размер типа `int` или числовое значение символа `'a'`), которые точно определены только для заданного компилятора, операционной системы, машинной архитектуры и т.д. Свойства языка, зависящие от реализации, перечислены в стандарте и должны быть указаны в сопроводительной документации компилятора, а также в стандартных заголовках, таких как `<limits>` (см. раздел Б.1.1). Таким образом, соответствие стандарту не эквивалентно переносимости программы на все реализации языка C++.
- *Неопределенность*. Смысл некоторых конструкций является *не установленным точно* (unspecified), *неопределенным* (undefined) или *не соответствующим стандарту, но не диагностируемым* (not conforming but not requiring a diagnostic). Очевидно, что такие возможности языка лучше не использовать. В этой книге их нет. Перечислим неопределенные возможности, которых следует избегать.
  - Несогласованные определения в разных исходных файлах (используйте заголовочные файлы согласованно; см. раздел 8.3).

- Многократное чтение *и* запись одной и той же переменной в выражении (основным примером является инструкция `a[i]=++i;`).
- Многочисленные явные преобразования типов (приведения), в особенности `reinterpret_cast`.

А.1.2. Запуск и завершение программы

Программа на языке C++ должна иметь единственную глобальную функцию с именем `main()`. Программа начинается с выполнения именно этой функции. Значение, возвращаемое функцией `main()`, имеет тип `int` (альтернативный тип `void` *не* соответствует стандарту). Значение, возвращаемое функцией `main()`, передается "системе". Некоторые системы игнорируют это значение, но признаком успешного завершения программы является нуль, а признаком ошибки — ненулевое значение или исключение, оставшееся не перехваченным (правда, такие не перехваченные исключения считаются признаком плохого стиля).

Аргументы функции `main()` могут зависеть от реализации, но любая реализация должна допускать два варианта (но только один для данной конкретной программы).

```
int main();                          // Без аргументов
int main(int argc, char* argv[]);    // Массив argv[] содержит argc C-строк
```

В определении функции `main()` явно указывать возвращаемое значение необязательно. В таком случае программа, дойдя до конца, вернет нуль. Вот как выглядит минимальная программа на языке C++:

```
int main() {}
```

Если вы определили в глобальной области видимости (или в области видимости пространства имен) объект, имеющий конструктор и деструктор, то вполне логично, чтобы конструктор выполнялся до функции `main()`, а деструктор — после функции `main()` (технически выполнение таких конструкторов является частью вызова функции `main()`, а выполнение деструкторов — частью возврата из функции `main()`). При малейшей возможности старайтесь избегать глобальных объектов, особенно если они требуют нетривиального создания и уничтожения.

А.1.3. Комментарии

Все, что можно сказать в исходном тексте программы, должно быть сказано. Однако в языке C++ есть два стиля комментариев, позволяющих программистам сказать то, что невозможно выразить с помощью кода.

```
// Это однострочный комментарий
/*
    Это многострочный
```

```
    блок комментариев
*/
```

Очевидно, что блоки комментариев чаще всего оформляются как многострочные комментарии, хотя некоторые люди предпочитают разделять их на несколько однострочных.

```
// Это многострочный комментарий,
// представленный в виде двух однострочных комментариев
/* Это однострочный комментарий, представленный как блочный */
```

Комментарии играют важную роль для документирования предназначения кода (см. также раздел 7.6.4).

A.2. Литералы

Литералы представляют значения разных типов. Например, литерал `12` представляет целое число "двенадцать", литерал `"Morning"` — символьную строку *Morning*, а литерал `true` — булево значение *true*.

A.2.1. Целочисленные литералы

Целочисленные литералы (integer literals) имеют три разновидности.

- Десятичные: последовательности десятичных цифр.
  Десятичные цифры: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` и `9`.
- Восьмеричные: последовательности восьмеричных цифр, начинающиеся с нуля.
  Восьмеричные цифры: `0`, `1`, `2`, `3`, `4`, `5`, `6` и `7`.
- Шестнадцатеричные: последовательности шестнадцатеричных цифр, начинающихся с `0x` или `0X`.
  Шестнадцатеричные цифры: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E` и `F`.
- Бинарные: последовательности бинарных цифр, начинающиеся с `0b` или `0B` (C++14).
  Бинарные цифры: `0`, `1`.

Суффикс `u` или `U` означает, что целочисленный литерал не имеет знака, т.е. имеет тип `unsigned` (см. раздел 25.5.3), а суффикс `l` или `L` относит их к типу `long`, например `10u` или `123456UL`.

Стандарт C++14 допускает также одинарные кавычки в качестве разделителя в числовых литералах. Например, `0b0000'0001'0010'0011` означает `0b0000000100100011`, а `1'000'000` означает `1000000`.

A.2.1.1. Числовые системы

Обычно мы записываем числа в десятичной системе. Число `123` означает `1` сотню плюс `2` десятки плюс `3` единицы или `1*100+2*10+3*1`, или (используя символ `^` для обозначения степени) `1*10^2+2*10^1+3*10^0`. Иногда вместо слова *десятичный* говорят "в системе счисления с основанием десять" (base-10). В данном случае число 10 означает, что в выражении `1*base^2+2*base^1+3*base^0` выполняется условие `base==10`. Существует много теорий, объясняющих, почему мы используем десятичную систему счисления. Одна из них апеллирует к естественным языкам: у нас на руках десять пальцев, а каждый символ, такой как 0, 1 и 2, представляющий собой цифру в позиционной системе счисления, в английском языке называется *digit*. Слово *Digit* в латинском языке означает *палец*.

Впрочем, иногда используются и другие системы счисления. Как правило, положительные целые числа в памяти компьютера представляются в двоичной системе счисления, т.е. основание системы счисления равно 2 (значения 0 и 1 относительно легко представить с помощью физических состояний). Люди, сталкивающиеся с необходимостью решать задачи на низком уровне аппаратного обеспечения, иногда используют восьмеричную систему счисления (база равна 8), но более часто при рассмотрении содержимого памяти используется шестнадцатеричная система (основание системы счисления равно 16).

Рассмотрим шестнадцатеричную систему счисления. Мы должны именовать шестнадцать значений от 0 до 15. Обычно для этого используются символы `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`, где A имеет десятичное значение 10, `B` — десятичное значение 11 и так далее:

`A==10, B==11, C==12, D==13, E==14, F==15.`

Теперь можно записать десятичное число `123` как `7B` в шестнадцатеричной системе счисления. Для того чтобы убедиться в этом, заметим, что в шестнадцатеричной системе счисления число `7B` равно `7*16+11`, что в десятичной системе счисления равно `123`. И наоборот, шестнадцатеричное число `123` означает `1*16^2+2*16+3`, т.е. `1*256+2*16+3`, что в десятичной системе счисления равно `291`. Если вы никогда не сталкивались с недесятичными представлениями целых чисел, то мы настоятельно рекомендуем вам поупражняться в преобразовании чисел из десятичной системы в шестнадцатеричную и обратно. Обратите внимание, что шестнадцатеричная цифра имеет очень простое соответствие своему двоичному значению.

Шестнадцатеричное	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Бинарное	0000	0001	0010	0011	0100	0101	0110	0111
Шестнадцатеричное	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Бинарное	1000	1001	1010	1011	1100	1101	1110	1111

Это объясняет популярность шестнадцатеричной системы. В частности, значение байта просто выражается двумя шестнадцатеричными цифрами.

В языке C++ (к счастью) числа являются десятичными, если иное не указано явно. Для того чтобы указать, что число является шестнадцатеричным, следует поставить префикс `0X` (символ `X` происходит от слова *hex*), так что `123==0X7B` и `0X123==291`. Точно так же можно использовать символ `x` в нижнем регистре, поэтому `123==0x7B` и `0x123==291`. Аналогично мы можем использовать шестнадцатеричные цифры `a`, `b`, `c`, `d`, `e` и `f` в нижнем регистре. Например, `123==0x7b`.

Восьмеричная система имеет основание, равное восьми. В этом случае мы можем использовать только восемь восьмеричных цифр: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`. В языке C++ числа в восьмеричной системе счисления начинаются с символа `0`, так что число `0123` — это не десятичное число `123`, а `1*8^2+2*8+3`, т.е. `1*64+2*8+3` или (в десятичном виде) `83`. И наоборот, восьмеричное число `83`, т.е. `083`, равно `8*8+3`, т.е. десятичному числу `67`. Используя систему обозначений языка C++, получаем равенства `0123==83` и `083==67`.

Двоичная система имеет основание, равное двум. В этой системе есть только две цифры: `0` и `1`. В языке C++ невозможно непосредственно представить двоичные числа как литералы. В качестве литералов и формата ввода-вывода в языке C++ непосредственно поддерживаются только восьмеричные, десятичные и шестнадцатеричные числа. Однако двоичные числа полезно знать, даже если мы не можем явно представить их в тексте программы. Например, десятичное число `123` равно `1*2^6+1*2^5+1*2^4+1*2^3+0*2^2+1*2+1`, т.е. `1*64+1*32+1*16+1*8+0*4+1*2+1`, т.е. (в двоичном виде) `1111011`.

A.2.2. Литералы с плавающей точкой

Литералы с плавающей точкой (floating-point-literal) содержат десятичную точку (`.`), показатель степени (например, `e3`) или суффикс, обозначающий число с плавающей точкой (`d` или `f`). Рассмотрим примеры.

```
123      // int (нет десятичной точки, суффикса
         // или показателя степени)
123.     // double: 123.0
123.0    // double
.123     // double: 0.123
0.123    // double
1.23e3   // double: 1230.0
1.23e-3  // double: 0.00123
1.23e+3  // double: 1230.0
```

Литералы с плавающей точкой имеют тип `double`, если только суффикс не указывает иное. Рассмотрим примеры.

```
1.23  // double
1.23f // float
1.23L // long double
```

A.2.3. Булевы литералы

Литералами типа `bool` являются литералы `true` и `false`. Целочисленное значение литерала `true` равно `1`, а литерала `false` — `0`.

A.2.4. Символьные литералы

Символьный литерал (character literal) — это символ, заключенный в одинарные кавычки, например `'a'` или `'@'`. Кроме того, существует несколько "специальных символов".

Имя	Имя в кодировке ASCII	Имя в языке C++
Новая строка	NL	\n
Горизонтальная табуляция	HT	\t
Вертикальная табуляция	VT	\v
Забой	BS	\b
Возврат каретки	CR	\r
Перевод страницы	FF	\f
Звуковой сигнал	BEL	\a
Обратная косая черта	\	\\
Вопросительный знак	?	\?
Одинарная кавычка	'	\'
Двойная кавычка	"	\"
Восьмеричное значение	ooo	\ooo
Шестнадцатеричное значение	hhh	\xhhh

Специальный символ представляется с помощью имени в языке C++, заключенного в одинарные кавычки, например `'\n'` (новая строка) или `'\t'` (табуляция).

Набор символов содержит следующие видимые символы:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+|~`{}[]:";'<>?,./
```

В переносимом коде нельзя рассчитывать на дополнительные видимые символы. Значение символа, например `'a'` для буквы `a`, зависит от реализации (но его легко выяснить, например, с помощью инструкции `cout << int('a')`).

А.2.5. Строковые литералы

Строковый литерал (string literal) — это последовательность символов, заключенных в двойные кавычки, например **"Knuth"** и **"King Canute"**. Строковый литерал нельзя произвольно разбивать на несколько строк; для перехода на новую строку используется специальный символ **\n**.

```
"King
Canute "          // Ошибка: новая строка в строковом литерале
"King\nCanute"    // ОК: правильный переход на новую строку
```

Два строковых литерала, разделенных только пробелами, считаются одним строковым литералом. Рассмотрим пример.

```
"King" "Canute" // Эквивалентно "KingCanute" (без пробела)
```

Обратите внимание на то, что в строковых литералах могут находиться специальные символы, такие как **\n**.

А.2.6. Указательные литералы

Существует только один *указательный литерал* (pointer literal): нулевой указатель **nullptr**. Для совместимости в качестве нулевого указателя можно также использовать любое константное выражение, равное **0**:

```
t* p1 = 0;      // ОК: нулевой указатель
int* p2 = 2-2; // ОК: нулевой указатель
int* p3 = 1;    // Ошибка: 1 — int, а не указатель
int z = 0;
int* p4 = z;    // Ошибка: z — не константа
```

В данном случае значение **0** неявно преобразуется в нулевой указатель.

В языке C++ (но не в языке C, поэтому будьте осторожны с заголовками языка C) литерал **NULL** определен как имеющий значение **0**, поэтому можно написать следующий код:

```
int* p4 = NULL; // Нулевой указатель (при правильном
                // определении литерала NULL)
```

А.3. Идентификаторы

Идентификатор (identifier) — это последовательность символов, начинающаяся с буквы или знака подчеркивания, за которыми следуют (или не следуют) буквы, цифры или знаки подчеркивания (в верхнем или нижнем регистре).

```
int foo_bar; // ОК
int FooBar;  // ОК
int foo bar; // Ошибка: пробел в идентификаторе
int foo$bar; // Ошибка: символ $ в идентификаторе
```

Идентификаторы, начинающиеся со знака подчеркивания или содержащие двойной символ подчеркивания, резервируются для использования компилятором; не используйте их:

```
int _foo;      // Не рекомендуется
int foo_bar;   // ОК
int foo__bar;  // Не рекомендуется
int foo_;      // ОК
```

A.3.1. Ключевые слова

Ключевые слова (keywords) — это идентификаторы, используемые самим языком для выражения языковых конструкций.

Ключевые слова (зарезервированные идентификаторы)

alignas	class	explicit	noexcept	signed	typename
alignof	compl	export	not	sizeof	union
and	concept	extern	not_eq	static	unsigned
and_eq	const	false	nullptr	static_assert	using
asm	const_cast	float	operator	static_cast	virtual
auto	constexpr	for	or	struct	void
bitand	continue	friend	or_eq	switch	volatile
bitor	decltype	goto	private	template	wchar_t
bool	default	if	protected	this	while
break	delete	inline	public	thread_local	xor
case	do	int	register	throw	xor_eq
catch	double	long	reinterpret_cast	true	
char	dynamic_cast	mutable	requires	try	
char16_t	else	namespace	return	typedef	
char32_t	enum	new	short	typeid	

A.4. Область видимости, класс памяти и время жизни

Каждое имя в языке C++ (за прискорбным исключением имен препроцессора; см. раздел A.17) находится в определенной *области видимости* (scope); иначе говоря, имя принадлежит области текста, в которой его можно использовать. Данные (объекты) хранятся в памяти; вид памяти, используемой для хранения объекта, называется *классом памяти* (storage class). *Время жизни* (lifetime) объекта отсчитывается от момента его инициализации до момента окончательного уничтожения.

А.4.1. Область видимости

Существует пять видов *областей видимости* (см. раздел 8.4).

- *Глобальная область видимости* (global scope). Имя находится в глобальной области видимости, если только оно не объявлено в некоторой языковой конструкции (например, в классе или функции).
- *Область видимости пространства имен* (namespace scope). Имя находится в области видимости пространства имен, если оно определено в пространстве имен, но не в какой-либо языковой конструкции (например, в классе или функции). Формально говоря, глобальная область видимости — это область видимости пространства имен с "пустым именем".
- *Локальная область видимости* (local scope). Имя находится в локальной области видимости, если оно объявлено в функции (это определение включает параметры функции).
- *Область видимости класса* (class scope). Имя находится в области видимости класса, если оно является именем члена этого класса.
- *Область видимости инструкции* (statement scope). Имя находится в области видимости инструкции, если оно объявлено в части `(...)` инструкций **`for`**, **`while`**, **`switch`** или **`if`**.

Область видимости переменной распространяется (только) до конца инструкции, в которой она объявлена. Рассмотрим пример.

```
for (int i = 0; i<v.size(); ++i) {
    // Переменная i может быть использована здесь
}
if (i < 27) // Переменная i из инструкции for
            // вышла из своей области видимости
```

Области видимости класса и пространства имен имеют свои имена, поэтому обращаться к их членам можно и извне:

```
void f();       // В глобальной области видимости

namespace N {
    void f()    // В области видимости пространства имен N
    {
        int v;  // В локальной области видимости
        ::f();  // Вызов глобальной функции f()
    }
}

void f()
{
    N::f();     // Вызов функции f(x) из пространства имен N
}
```

Подумайте, что произойдет, если мы вызовем функцию `N::f()` или `::f()` из данного примера. (См. также раздел А.15.)

А.4.2. Класс памяти

Существуют три *класса памяти* (раздел 17.4).

- *Автоматическая память* (automatic storage). Переменные, определенные в функциях (сюда входят и параметры функций), размещаются в автоматической памяти (т.е. в стеке), если они явно не объявлены с помощью ключевого слова **static**. Автоматическая память выделяется при вызове функции и освобождается при возврате из функции. Таким образом, если функция (явно или неявно) вызывает сама себя, может существовать несколько копий автоматических данных, по одной копии на каждый вызов (см. раздел 8.5.8).
- *Статическая память* (static storage). Переменные, объявленные в глобальной области видимости и в области видимости пространства имен, хранятся в статической памяти, как и переменные, явно объявленные с помощью ключевого слова **static** в функциях и классах. Редактор связей выделяет статическую память до начала выполнения программы.
- *Динамическая память* (*куча*) (free store (heap)). Объекты, созданные с помощью оператора **new**, размещаются в динамической памяти.

Рассмотрим пример.

```
vector<int> vg(10); // Создается один раз при запуске
                    // программы ("до функции main()")
vector<int>* f(int x)
{
    static vector<int> vs(x); // Создается при первом вызове f()
    vector<int> vf(x+x);      // Создается при каждом вызове f()

    for (int i=1; i<10; ++i) {
        vector<int> v1(i);    // Создается на каждой итерации
        // ...
    }   // Переменная v1 (на каждой итерации) уничтожается здесь

    return new vector<int>(vf); // Копия переменной vf создается
                                // в динамической памяти
}       // Переменная vf уничтожается здесь

void ff()
{
    vector<int>* p = f(10);  // Получает вектор от функции f()
    // ...
    delete p;                // Удаляет вектор, полученный от
}                            // функции f
```

Переменные `vg` и `vs`, размещенные в статической памяти, уничтожаются по завершении программы (после завершения функции `main()`) при условии, что они были созданы.

Память для членов класса не выделяется как таковая. Когда вы размещаете объект в каком-то месте, там же размещаются нестатические члены (в том же классе памяти, что и сам объект, которому они принадлежат).

Код хранится отдельно от данных. Например, функция-член *не* хранится в каждом объекте своего класса; одна ее копия хранится вместе с остальной частью кода программы.

(См. также разделы 14.3 и 17.4.)

А.4.3. Время жизни

Перед тем как объект будет (корректно) использован, он должен быть проинициализирован. Эту инициализацию можно осуществить явно, с помощью инициализатора, или неявно, используя конструктор или правило инициализации объектов встроенных типов по умолчанию. Время жизни объекта заканчивается в точке, определенной его областью видимости и классом памяти (например, см. разделы 17.4 и Б.4.2).

- *Локальные (автоматические) объекты* создаются, когда поток выполнения достигает их определения, и уничтожаются при выходе из области видимости.
- *Временные объекты* создаются конкретным подвыражением и уничтожаются по завершении полного выражения. Полное выражение — это выражение, которое не является подвыражением другого выражения.
- *Объекты в пространстве имен и статические члены классов* создаются при запуске программы (до функции `main()`) и уничтожаются в конце программы (после функции `main()`).
- *Локальные статические объекты* создаются, когда (если) поток выполнения достигает их определения и уничтожаются (если они были созданы) в конце программы.
- *Объекты в динамической памяти* создаются оператором `new` и (необязательно) уничтожаются с помощью оператора `delete`.

Временная переменная, связанная с локальной ссылкой или пространством имен, существует столько же, сколько и сама ссылка. Рассмотрим пример.

```
const char* string_tbl[] = {"Mozart","Grieg","Haydn","Chopin"};
const char* f(int i) { return string_tbl[i]; }
void g(string s){}

void h()
```

```
{
    const string& r = f(0);  // Связываем временную строку со ссылкой r
    g(f(1));                 // Создаем временную строку и передаем ее
    string s = f(2);         // Копируем в s временную строку
    cout << "f(3): " << f(3) // Выводим временную строку
         << " s: " << s
         << " r: " << r << '\n';
}
```

Результат выглядит следующим образом:

```
f(3): Chopin s: Haydn r: Mozart
```

Временные строки, сгенерированные вызовами `f(1)`, `f(2)` и `f(3)`, уничтожаются в конце выражения, в котором они были созданы. Однако временная строка, сгенерированная при вызове `f(0)`, связана с переменной `r` и "живет" до конца функции `h()`.

A.5. Выражения

В этом разделе описываются операторы языка C++. Мы используем обозначения, которые считаем мнемоническими, например `m` — для имени члена; `T` — для имени типа; `p` — для выражения, создающего указатель; `x` — для выражения; `v` — для выражения lvalue; `lst` — для списка аргументов. Типы результатов арифметических операций определяются "обычными арифметическими преобразованиями" (раздел A.5.2.2). Описания, приведенные в этом разделе, касаются только встроенных операторов, а не операторов, которые программист может определить самостоятельно, хотя, определяя собственные операторы, следует придерживаться семантических правил, описанных для встроенных операторов (см. раздел 9.6).

Разрешение области видимости	
`N::m`	`m` находится в пространстве имен `N`; `N` — имя пространства имен или класса
`::m`	`m` находится в глобальном пространстве имен

Обратите внимание на то, что члены могут быть вложенными, поэтому можно получить такие выражения, как `N::C::m` (см. также раздел 8.7).

Постфиксные выражения	
`x.m`	Доступ к члену класса; `x` должен быть объектом класса
`p->m`	Доступ к члену класса; `p` должен быть указателем на объект класса; эквивалентно `(*p).m`
`p[x]`	Индексирование; эквивалентно `*(p+x)`
`f(lst)`	Вызов функции: вызов функции `f` со списком аргументов `lst`
`T(lst)`	Создание: создание объекта `T` со списком аргументов `lst`
`v++`	Постфиксный инкремент; значение `v++` равно значению `v` до инкремента

Постфиксные выражения	
`v--`	Постфиксный декремент; значение `v--` равно значению `v` до декремента
`typeid(x)`	Идентификация типа объекта `x` времени выполнения программы
`typeid(T)`	Идентификация типа `T` времени выполнения программы
`dynamic_cast<T>(x)`	Преобразование времени выполнения объекта `x` в объект типа `T` с проверкой
`static_cast<T>(x)`	Преобразование времени компиляции объекта `x` в объект типа `T` с проверкой
`const_cast<T>(x)`	Непроверяемое преобразование, которое сводится к добавлению или удалению модификатора `const` у типа объекта `x`, чтобы получить объект типа `T`
`reinterpret_cast<T>(x)`	Непроверяемое преобразование объекта `x` в объект типа `T` путем иной интерпретации битовой комбинации объекта `x`

Оператор `typeid` и его применения не описаны в этой книге; детальную информацию можно найти в более серьезной литературе. Обратите внимание на то, что операторы приведения не модифицируют свой аргумент. Вместо этого они создают результат своего типа, который некоторым образом соответствует значению аргумента (раздел A.5.7).

Унарные выражения	
`sizeof(T)`	Размер типа `T` в байтах
`sizeof(x)`	Размер типа объекта `x` (в байтах)
`++v`	Префиксный инкремент; эквивалентно `v+=1`
`--v`	Префиксный декремент; эквивалентно `v-=1`
`~x`	Дополнение `x`; `~` — побитовая операция
`!x`	Отрицание `x`; возвращает `true` или `false`
`&v`	Адрес переменной `v`
`*p`	Содержание объекта, на который указывает указатель `p`
`new T`	Создает объект типа `T` в динамической памяти
`new T(lst)`	Создает объект типа `T` в динамической памяти и инициализирует его с помощью `lst`
`new(lst) T`	Создает объект типа `T` в области памяти, заданной аргументом `lst`
`new(lst) T(lst2)`	Создает объект типа `T` в области памяти, заданной аргументом `lst`, и инициализирует его с помощью `lst2`
`delete p`	Удаляет объект, на который указывает указатель `p`
`delete[] p`	Удаляет массив объектов, на который указывает указатель `p`
`(T)x`	Приведение в стиле языка C; преобразует объект `x` в объект типа `T`

Объекты, на которые указывает указатель `p` в инструкциях `delete p` и `delete[] p`, должны быть размещены в памяти с помощью оператора `new`

(раздел А.5.6). Следует подчеркнуть, что выражение `(T)x` является менее конкретным — а следовательно, более уязвимым для ошибок, — чем более конкретные операторы приведения (раздел А.5.7).

Выбор члена класса	
`x.*ptm`	Член объекта `x`, определенный указателем на член класса `ptm`
`p->*ptm`	Член `*p`, идентифицированный указателем на член класса `ptm`

Эти инструкции в книге не рассматриваются; обратитесь к более серьезной литературе.

Мультипликативные операторы	
`x*y`	Умножение `x` на `y`
`x/y`	Деление `x` на `y`
`x%y`	Деление по модулю (остаток от деления) `x` на `y` (не для типов с плавающей точкой)

Если `y==0`, то результат выражений `x/y` и `x%y` не определен. Если переменная `x` или `y` отрицательна, то результат выражения `x%y` зависит от реализации.

Аддитивные операторы	
`x+y`	Сложение `x` и `y`
`x-y`	Вычитание `y` из `x`
Операторы сдвига	
`x<<y`	Сдвигает биты `x` влево на `y` позиций
`x>>y`	Сдвигает биты `x` вправо на `y` позиций

Для встроенных типов операторы >> и << означают сдвиг битов (см. раздел 25.5.4). Если же левым операндом является объект класса `iostream`, то эти операторы используются для ввода и вывода (см. главы 10 и 11).

Операторы сравнения	
`x<y`	`x` меньше `y`; возвращает объект типа `bool`
`x<=y`	`x` меньше или равно `y`
`x>y`	`x` больше `y`
`x>=y`	`x` больше или равно `y`

Результатом оператора сравнения является значение типа `bool`.

Операторы равенства	
`x==y`	`x` равно `y`; возвращает значение типа `bool`
`x!=y`	`x` не равно `y`

Заметим, что `x!=y` эквивалентно `!(x==y)`. Результат оператора равенства имеет тип `bool`.

	Побитовое "и"
`x&y`	Побитовое "и" значений `x` и `y`

Оператор `&` (как и операторы `^`, `|`, `~`, `>>` и `<<`) возвращает комбинацию битов. Например, если переменные `a` и `b` имеют тип `unsigned char`, то результат выражения `a&b` имеет тип `unsigned char`, в котором каждый бит является результатом применения оператора `&` к соответствующим битам переменных `a` и `b` (раздел A.5.5).

	Побитовое "исключающее или" (xor)
`x^y`	Побитовое "исключающее или" значений `x` и `y`
	Побитовое "или"
`x|y`	Побитовое "или" значений `x` и `y`
	Логическое "и"
`x&&y`	Логическое "и"; возвращает значения `true` и `false`; вычисляет значение `y`, только если `x` имеет значение `true`
	Логическое "или"
`x||y`	Логическое "или"; возвращает значения `true` и `false`; вычисляет значение `y`, только если `x` имеет значение `false`

(См. раздел A.5.5.)

	Условное выражение
`x?y:z`	Если `x` равно `true`, то результат равен `y`; иначе — равен `z`

Рассмотрим пример.

```
template<typename T> T& max(T& a, T& b) { return (a>b)?a:b; }
```

Оператор "`?:`" описан в разделе 8.4.

	Присваивание
`v=x`	Присваивает `x` переменной `v`; результат равен `v`
`v*=x`	Аналог `v=v*(x)`
`v/=x`	Аналог `v=v/(x)`
`v%=x`	Аналог `v=v%(x)`
`v+=x`	Аналог `v=v+(x)`
`v-=x`	Аналог `v=v-(x)`
`v>>=x`	Аналог `v=v>>(x)`
`v<<=x`	Аналог `v=v<<(x)`
`v&=x`	Аналог `v=v&(x)`
`v^=x`	Аналог `v=v^(x)`
`v|=x`	Аналог `v=v|(x)`

Фраза "Аналог `v=v*(x)`" означает, что значение выражения `v*=x` совпадает со значением выражения `v=v*(x)`, за исключением того, что значение `v` вычисляется только один раз. Например, выражение `v[++i]*=7+3` означает `(++i, v[i]=v[i]*(7+3))`, а не `(v[++i]=v[++i]*(7+3))` (которое может быть неопределенным; см. раздел 8.6.1).

Выражение throw	
`throw x`	Генерирует значение `x`

Тип выражения `throw` — `void`.

Выражение "запятая"	
`x,y`	Выполняет инструкцию `x`, а затем `y`. Результатом является результат инструкции `y`

Каждая таблица содержит операторы, имеющие одинаковый приоритет. Операторы в более высоко расположенных таблицах имеют более высокий приоритет по сравнению с операторами, расположенными ниже. Например, выражение `a+b*c` означает `a+(b*c)`, а не `(a+b)*c`, поскольку оператор `*` имеет более высокий приоритет по сравнению с оператором `+`. Аналогично выражение `*p++` означает `*(p++)`, а не `(*p)++`. Унарные операторы и операторы присваивания являются *правоассоциативными* (right-associative); все остальные — левоассоциативными. Например, выражение `a=b=c` означает `a=(b=c)`, а выражение `a+b+c` означает `(a+b)+c`.

Lvalue представляет собой выражение, идентифицирующее объект, который в принципе может быть изменен (очевидно, что объект lvalue, имеющий тип `const`, защищен от изменений системой типов) и имеет адрес. Противоположностью выражения lvalue является выражение rvalue, т.е. выражение, идентифицирующее нечто, что не может быть изменено или не имеет адреса, как, например, значение, возвращаемое функцией (`&f(x)` является ошибкой, поскольку значение, возвращаемое функцией `f(x)`, является значением rvalue).

A.5.1. Операторы, определенные пользователем

Правила, перечисленные выше, определены для встроенных типов. Если же используется оператор, определенный пользователем, то выражение просто преобразовывается в вызов соответствующей операторной функции, определенной пользователем, и происходящее определяется правилами, установленными для вызова функций. Рассмотрим пример.

```
class Mine { /* ... */ };
bool operator==(Mine, Mine);

void f(Mine a, Mine b)
{
```

```
    if (a==b) { // a==b означает operator==(a,b)
        // ...
    }
}
```

Тип, определенный пользователем, — это класс (см. главу 9, раздел A.12) или перечисление (см. разделы 9.5, A.11).

A.5.2. Неявное преобразование типа

Целочисленные типы и типы с плавающей точкой (раздел A.8) могут свободно смешиваться в операторах присваивания и в выражениях. При первой же возможности значения преобразовываются так, чтобы не потерять информацию. К сожалению, преобразования, уничтожающие значение, выполняются также неявно.

A.5.2.1. Повышение

Неявные преобразования, сохраняющие значения, обычно называют *повышениями* (promotions). Например, перед выполнением арифметической операции для создания типа `int` из более коротких целочисленных типов выполняется *целочисленное повышение* (integral promotion). Это отражает исходную цель повышений: привести операнды арифметических операций к "естественным" размерам. Кроме того, преобразование значения типа `float` в значение типа `double` также считается повышением.

Повышения используются как часть обычных арифметических преобразований (раздел A.5.2.2).

A.5.2.2. Преобразования

Значения фундаментальных типов можно преобразовывать друг в друга самыми разными способами. При написании программы следует избегать неопределенного поведения и преобразований, которые молча отбрасывают информацию (см. разделы 3.9 и 25.5.3). Компиляторы обычно способны предупредить о многих сомнительных преобразованиях.

- *Целочисленные преобразования*. Целое число может быть преобразовано в другой целочисленный тип. Значение перечисления может быть преобразовано в целый тип. Если результирующим типом является тип без знака (`unsigned`), то результирующее значение просто будет иметь столько битов источника, сколько сможет поместиться в целевой области памяти (старшие биты при необходимости будут отброшены). Если целевой тип имеет знак, то значение останется без изменения при условии, что его можно представить с помощью целевого типа; в противном случае значение зависит от реализации языка. Обратите внимание на то, что типы `bool` и `char` являются целочисленными.

- *Преобразования значений с плавающей точкой.* Значение с плавающей точкой можно преобразовать в значение с плавающей точкой другого типа. Если исходное значение можно точно представить с помощью целевого типа, то результатом будет исходное числовое значение. Если же исходное значение лежит между двумя соседними целевыми значениями, то результатом будет одно из этих значений. В противном случае поведение не определено. Заметим, что преобразование значения типа `float` в значение типа `double` считается повышением.
- *Преобразование указателей и ссылок.* Любой указатель на тип объекта можно неявно преобразовать в указатель типа `void*` (см. разделы 17.8 и 27.3.5). Указатель (ссылку) на производный класс можно неявно преобразовать в указатель (ссылку) на доступный и однозначно определенный базовый класс (см. раздел 14.3). Константное выражение (см. разделы А.5 и 4.3.1), равное нулю, можно неявно преобразовать в любой тип указателя. Указатель типа `T*` можно неявно преобразовать в указатель `const T*`. Аналогично ссылку `T&` можно неявно преобразовать в ссылку типа `const T&`.
- *Булевы преобразования.* Указатели, целые числа и числа с плавающей точкой можно неявно преобразовать в значение типа `bool`. Ненулевое значение преобразовывается в значение `true`, а нуль — в значение `false`.
- *Преобразования чисел с плавающей точкой в целые числа.* Если число с плавающей точкой преобразуется в целое число, то его дробная часть отбрасывается. Иначе говоря, преобразование из типа с плавающей точкой в целый тип является усечением. Если усеченное значение невозможно представить с помощью целевого типа, то поведение программы не определено. Преобразования целых чисел в числа с плавающей точкой являются математически корректными только в той степени, в какой это допускается аппаратным обеспечением. Если целое число невозможно точно представить как число с плавающей точкой, происходит потеря точности.
- *Обычные арифметические преобразования.* Эти преобразования выполняются над операндами бинарных операторов, чтобы привести их к общему типу, который затем используется в качестве типа результата.

  1. Если один из операндов имеет тип `long double`, то другой преобразовывается в тип `long double`. В противном случае, если один из операндов имеет тип `double`, другой преобразовывается в тип `double`. В противном случае, если один из операндов имеет тип `float`, другой преобразовывается в тип `float`. В противном случае над обоими операндами выполняется целочисленное повышение.

2. Если один из операндов имеет тип **unsigned long**, то другой преобразовывается в тип **unsigned long**. В противном случае, если один из операндов имеет тип **long int**, а другой — **unsigned int**, значение типа **unsigned int** преобразуется в значение типа **long int** при условии, что тип **long int** может представить все значения типа **unsigned int**. В противном случае оба операнда преобразовываются в тип **unsigned long int**. В противном случае, если один из операндов имеет тип **long**, другой операнд преобразовывается в тип **long**. В противном случае, если один из операндов имеет тип **unsigned**, другой преобразовывается в тип **unsigned**. В противном случае оба операнда имеют тип **int**.

Очевидно, что лучше не полагаться на слишком запутанные сочетания типов и минимизировать необходимость неявных преобразований.

A.5.2.3. Преобразования, определенные пользователем

В дополнение к стандартным преобразованиям и повышениям программист может определить преобразования для пользовательских типов. Конструктор, принимающий один аргумент, определяет преобразование из типа аргумента в значение своего типа. Если конструктор имеет спецификатор **explicit** (см. раздел 18.3.1), то преобразование происходит, только если программист явно потребует его выполнить. В противном случае преобразование может быть неявным.

A.5.3. Константные выражения

Константное выражение (constant expression) — это выражение, которое может быть вычислено на этапе компиляции и содержит только операнды типа **int**. (Это немного упрощенное определение, но для большинства целей оно вполне подходит.) Рассмотрим пример.

```
const int a = 2*3;
const int b = a+3;

constexpr int a = 2*3;
constexpr int b = a+3;
```

Константа (**const**) может быть инициализирована выражением, включающим переменные. Константное выражение (**constexpr**) может быть инициализировано константным выражением. Константные выражения требуются в немногих случаях, например в качестве границ массивов, меток разделов **case**, инициализаторов перечислений и шаблонных аргументов типа **int**. Рассмотрим пример.

```
int var = 7;
switch (x) {
    case 77:  // OK
```

```
    case a+2: // OK
    case var: // Ошибка (var - не константное выражение)
              // ...
};
```

Функция, объявленная как **constexpr**, может использоваться в константных выражениях.

A.5.4. Оператор sizeof

В выражении **sizeof(x)** аргумент **x** может быть типом или выражением. Если **x** — выражение, то значением **sizeof(x)** является размер результирующего объекта. Если **x** — тип, то значением **sizeof(x)** является размер объекта типа **x**. Размеры измеряются в байтах. По определению **sizeof(char)==1**.

A.5.5. Логические выражения

В языке C++ предусмотрены логические операторы для целочисленных типов.

Побитовые логические операции	
x&y	Побитовое "и" для **x** и **y**
x|y	Побитовое "или" для **x** и **y**
x^y	Побитовое "исключающее или" для **x** и **y**
Логические операции	
x&&y	Логическое "и"; возвращает **true** или **false**; вычисляет **y**, только если **x** равно **true**
x||y	Логическое "или"; возвращает **true** или **false**; вычисляет **y**, только если **x** равно **false**

Побитовые операторы применяются к каждому биту своих операндов, в то время как логические операторы (**&&** и **||**) трактуют число **0** как значение **false**, а все остальные — как **true**. Определения этих операций приведены ниже.

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

A.5.6. Операторы new и delete

Память из динамической памяти (или кучи) выделяется с помощью оператора **new**, а освобождается — с помощью оператора **delete** (для индивидуальных объектов) или **delete[]** (для массивов).

Если память исчерпана, оператор **new** генерирует исключение **bad_alloc**. В случае успеха операция **new** выделяет как минимум один байт и возвращает указатель на объект, размещенный в памяти. Тип этого объекта указывается после оператора **new**:

```
int* p1 = new int;        // Размещает (неинициализированное)
                          // число типа int
int* p2 = new int(7);     // Размещает число типа int,
                          // инициализированное значением 7
int* p3 = new int[100]; // Размещает 100 (неинициализированных)
                          // чисел int
// ...
delete p1;                // Удаляет индивидуальный объект
delete p2;
delete[] p3;              // Удаляет массив
```

Если с помощью оператора **new** вы размещаете в памяти объекты встроенного типа, то, если не указан инициализатор, они не будут инициализированы. Если с помощью оператора **new** вы размещаете в памяти объекты класса, имеющего конструктор, то будет вызван конструктор; если инициализатор не указан, будет вызван конструктор по умолчанию (см. раздел 17.4.4).

Оператор **delete** вызывает деструктор своего операнда (если таковой имеется). Обратите внимание на то, что деструктор может быть виртуальным (раздел A.12.3.1).

A.5.7. Операторы приведения

Существуют четыре оператора преобразования типов.

Операторы преобразования типов	
x=dynamic_cast<D*>(p)	Пытается преобразовать указатель **p** в объект типа **D*** (может вернуть **0**)
x=dynamic_cast<D&>(*p)	Пытается преобразовать значение ***p** в объект типа **D&** (может генерировать исключение **bad_cast**)
x=static_cast<T>(v)	Преобразует операнд **v** в значение типа **T**, если тип **T** можно преобразовать в тип операнда **v**
x=reinterpret_cast<T>(v)	Преобразует операнд **v** в значение типа **T**, представленное тем же набором битов
x=const_cast<T>(v)	Преобразует операнд **v** в значение типа **T**, добавляя или удаляя модификатор **const**
x=(T)v	Приведение в стиле языка C: выполняет любое старое приведение
x=T(v)	Приведение в функциональном стиле: выполняет любое старое приведение
x=T{v}	Создает объект типа **T** из **v** (без сужения)

Динамическое приведение обычно используется для навигации по иерархии классов, если указатель `p` — указатель на базовый класс, а класс `D` — производный от базового класса. Если операнд `v` не имеет тип `D*`, эта операция возвращает значение `0`. Если вы хотите, чтобы операция `dynamic_cast` в случае неудачи не возвращала `0`, а генерировала исключение `bad_cast`, то ее следует применять к ссылкам, а не к указателям. Динамическое приведение — единственное приведение, опирающееся на проверку типов времени выполнения программы.

Статическое приведение используется для "разумных преобразований", т.е. таких, когда операнд `v` может быть результатом неявного преобразования из типа `T` (см. раздел 17.8).

Оператор `reinterpret_cast` используется для изменения интерпретации комбинации битов. Его переносимость не гарантируется. Лучше всего считать, что он является вообще не переносимым. Типичным примером изменения интерпретации является преобразование целого числа в указатель, чтобы получить машинный адрес в программе (см. разделы 17.8 и 25.4.1).

Приведения в стиле языка C и функциональные приведения могут выполнить любое преобразование типа, которое можно осуществить с помощью `static_cast` или `reinterpret_cast` в сочетании с `const_cast`.

Приведений лучше избегать. В большинстве случаев их использование свидетельствует о плохом стиле программирования. Исключения из этого правила представлены в разделах 17.8 и 25.4.1. Приведения в стиле языка C и в функциональном стиле приведения обладают ужасным свойством: позволяют не вникать в то, что именно они делают (см. раздел 27.3.4). Если вы не можете избежать явного преобразования типа, лучше используйте именованные приведения.

A.6. Инструкции

Грамматическое определение инструкций языка C++ приведено ниже (*opt* означает "необязательный").

statement:
- *declaration*
- `{` *statement-list*$_{opt}$ `}`
- `try` `{` *statement-list*$_{opt}$ `}` *handler-list*
- *expression*$_{opt}$ `;`
- *selection-statement*
- *iteration-statement*
- *labeled-statement*
- *control-statement*

selection-statement:
- `if` `(` *condition* `)` *statement*

```
        if ( condition ) statement else statement
        switch ( condition ) statement
iteration-statement:
        while ( condition ) statement
        do statement while ( expression ) ;
        for ( for-init-statement condition_opt ; expression_opt ) statement
        for ( declaration : expression ) statement
labeled-statement:
        case constant-expression : statement
        default : statement
        identifier : statement
control-statement:
        break ;
        continue ;
        return expression_opt ;
        goto identifier ;
statement-list:
        statement statement-list_opt
condition:
        expression
        type-specifier declarator = expression
for-init-statement:
        expression_opt ;
        type-specifier declarator = expression ;
handler-list:
catch ( exception-declaration ) { statement-list_opt }
        handler-list handler-list_opt
```

Обратите внимание на то, что объявление — это инструкция, а присваивание и вызов функции являются выражениями. Дополнительную информацию можно найти в указанных ниже разделах книги.

- Итерация (**for** и **while**); см. раздел 4.4.2.
- Ветвление (**if**, **switch**, **case** и **break**); см. раздел 4.4.1. Инструкция **break** прекращает выполнение ближайшей вложенной инструкции **switch**, **while**, **do** или **for**. Иначе говоря, следующей будет выполнена инструкция, идущая за последней в теле одной из перечисленных выше инструкций.
- Выражения; см. разделы A.5 и 4.3.
- Объявления; см. разделы A.6 и 8.2.
- Исключения (**try** и **catch**); см. разделы 5.6 и 19.4.

Рассмотрим пример, созданный просто для того, чтобы продемонстрировать разнообразие инструкций (какую задачу они решают?).

```
int* f(int p[], int n)
{
    if (p==0) throw Bad_p(n);
    vector<int> v;
    int x;
    while (cin>>x) {
        if (x==terminator) break; // Выход из цикла while
        v.push_back(x);
    }
    for (int i = 0; i<v.size() && i<n; ++i) {
        if (v[i]==*p)
            return p;
        else
            ++p;
    }
    return 0;
}
```

A.7. Объявления

Объявление (declaration) состоит из трех частей:

- имя объявляемой сущности;
- тип объявляемой сущности;
- начальное значение объявляемой сущности (во многих случаях необязательное).

Мы можем объявлять следующие сущности:

- объекты встроенных типов и типов, определенных пользователем (раздел A.8);
- типы, определенные пользователем (классы и перечисления) (разделы A.10–A.11, глава 9);
- шаблоны (классов и функций) (раздел A.13);
- псевдонимы (раздел A.16);
- пространства имен (разделы A.15 и 8.7);
- функции (включая функции-члены и операторы) (раздел A.9, глава 8);
- перечисления (значения перечислений) (разделы A.11 и 9.5);
- макросы (разделы A.17.2 и 27.8).

Инициализатор может представлять собой список из нуля или большего количества элементов в фигурных скобках (разделы 3.9.2, 9.4.2, 18.2). Например:

```
vector<int> v {a,b,c,d};
int x {y*z};
```

Если тип объекта в определении указан как `auto`, объект должен быть инициализирован, а его тип совпадать с типом инициализатора (разделы 13.3, 21.2). Например:

```
auto x = 7;              // Тип x – int
const auto pi = 3.14;    // Тип pi – double
for (const auto& x : v) // x – ссылка на элемент v
```

А.7.1. Определения

Определение с инициализацией, резервирующее область памяти или каким-то иным путем сообщающее компилятору всю информацию, необходимую для использования имени в программе, называется *определением* (definition). Каждый тип, объект и функция в программе должны иметь только одно определение. Рассмотрим примеры.

```
double f();                 // Объявление
double f() { /* ... */ }; // Объявление и определение
extern const int x;         // Объявление
int y;                      // Объявление и определение
int z = 10;                 // Определение с явной инициализацией
```

Константы должны быть инициализированы. Для этого используется инициализатор, если только константа не объявлена с помощью ключевого слова `extern` (в таком случае инициализатор вместе с определением должны быть расположены в другом месте) или если константа не имеет тип, имеющий конструктор по умолчанию (раздел А.12.3). Константные члены класса должны инициализироваться в каждом конструкторе с помощью инициализаторов членов (раздел А.12.3).

А.8. Встроенные типы

Язык C++ имеет много фундаментальных типов и типов, созданных из фундаментальных типов с помощью модификаторов.

Встроенные типы	
`bool x`	`x` — булева переменная (`true` или `false`)
`char x`	`x` — символ (обычно 8 битов)
`short x`	`x` — короткий тип `int` (обычно 16 битов)
`int x`	`x` — обычный целый тип по умолчанию
`float x`	`x` — число с плавающей точкой ("короткий `double`")
`double x`	`x` — число с плавающей точкой двойной точности
`void* p`	`p` — указатель на ячейку памяти (неизвестного типа)
`T* p`	`p` — указатель на объект типа `T`

Встроенные типы	
`T *const p`	`p` — константный (неизменяемый) указатель на объект типа `T`
`T a[n]`	`a` — массив, состоящий из `n` элементов типа `T`
`T& r`	`r` — ссылка на объект типа `T`
`T f(arguments)`	`f` — функция, получающая список аргументов `arguments` и возвращающая объект типа `T`
`const T x`	`x` — объект константной (неизменяемой) версии типа `T`
`long T x`	`x` — объект длинного типа `T`
`unsigned T x`	`x` — объект типа `T` без знака
`signed T x`	`x` — объект типа `T` со знаком

Здесь `T` означает "некий тип", поэтому могут существовать варианты типов `long unsigned int`, `long double`, `unsigned char` и `const char*` (указатель на константный символ `char`). Однако эта система не совсем полная; например, в ней нет типа `short double` (его роль играет тип `float`); типа `signed bool` (совершенно бессмысленного); типа `short long int` (это было бы лишним) и типа `long long long long int`. Что касается типа `long long`, то гарантируется, что он содержит не менее 64 бит.

Типы с плавающей точкой (floating-point types) — это типы `float`, `double` и `long double`. Они являются приближением действительных чисел в языке C++.

Целочисленные типы (integer types) — это типы `bool`, `char`, `short`, `int`, `long` и `long long`, а также их варианты без знака. Обратите внимание на то, что тип или значения перечислений часто можно использовать вместо целочисленного типа или значения.

Размеры встроенных типов обсуждались в разделах 3.8, 17.3.1 и 25.5.1; указатели и массивы — в главах 17 и 18; ссылки — в разделах 8.5.4–8.5.6.

A.8.1. Указатели

Указатель (pointer) — это адрес объекта или функции. Указатели хранятся в переменных типов указателей. Корректный указатель на объект содержит адрес этого объекта:

```
int x  = 7;
int* pi = &x;     // Указатель pi указывает на объект x
int xx  = *pi;    // *pi - значение объекта, на который указывает pi, т.е. 7
```

Некорректный указатель — это указатель, не содержащий указателя ни на один объект.

```
int* pi2;         // Неинициализированный
*pi2 = 7;         // Неопределенное поведение
pi2 = nullptr;    // Нулевой указатель (указатель pi2
                  // остается некорректным)
```

```
*pi2 = 7;          // Неопределенное поведение
pi2 = new int(7); // Теперь pi2 становится корректным
int xxx = *pi2;    // OK: переменная xxx становится равной 7
```

Мы стараемся, чтобы все некорректные указатели были нулевыми (`nullptr`), поэтому можем проверять их соответствующим образом:

```
if (p2 == 0) {     // "Если указатель некорректный"
                   // Не используйте значение *p2
}
```

Или еще проще:

```
if (p2) {          // "Если указатель корректный"
                   // Используйте значение *p2
}
```

(См. разделы 17.4 и 18.6.4.)

Перечислим операции над указателями на объекты (не-`void`).

Операции над указателями	
`*p`	Разыменование/косвенное обращение к памяти
`p[i]`	Разыменование/индексация
`p=q`	Присваивание и инициализация
`p==q`	Равенство
`p!=q`	Неравенство
`p+i`	Добавление целого числа
`p-i`	Вычитание целого числа
`p-q`	Расстояние: вычисление указателей
`++p`	Префиксный инкремент (перемещение вперед)
`p++`	Постфиксный инкремент (перемещение вперед)
`--p`	Префиксный декремента (перемещение назад)
`p--`	Постфиксный декремент (перемещение назад)
`p+=i`	Перемещение вперед на `i` элементов
`p-=i`	Перемещение назад на `i` элементов

Заметим, что все виды операций арифметики указателей (например, `++p` и `p+=7`) могут применяться только к указателям, указывающим в массив, а результат разыменования указателя, указывающего на область памяти за пределами массива, не определен (и, скорее всего, не сможет быть проверен компилятором или системой времени выполнения языка). Сравнения `<`, `<=`, `>` и `>=` также могут применяться для указателей одного и того же типа, указывающих в один и тот же объект или массив.

Единственными операциями над указателем типа `void*` являются копирование (присваивание или инициализация), приведение (преобразование типа) и сравнение (`==`, `!=`, `<`, `<=`, `>` и `>=`).

Указатель на функцию (см. раздел 27.2.5) можно только копировать и вызывать. Рассмотрим пример.

```
using Handle_type = void(*)(int);
void my_handler(int);
Handle_type handle = my_handler;
handle(10);      // Эквивалентно my_handler(10)
```

A.8.2. Массивы

Массив (array) — это неразрывная последовательность объектов (элементов) одинакового типа, имеющая фиксированную длину.

```
int a[10]; // 10 целых чисел
```

Если массив является глобальным, то его элементы инициализируются соответствующим значением по умолчанию, принятым для данного типа. Например, значение `a[7]` будет равно 0. Если массив является локальным (переменная объявлена в функции) или создан с помощью оператора **new**, то элементы встроенных типов останутся неинициализированными, а элементы, имеющие пользовательский тип, будут инициализированы с помощью конструкторов.

Имя массива неявно преобразуется в указатель на его первый элемент; например:

```
int* p = a; // Указатель p указывает на элемент a[0]
```

Массив или указатель на элемент массива может индексироваться с помощью оператора `[]`:

```
a[7]   = 9;
int xx = p[6];
```

Элементы массива нумеруются начиная с нуля (раздел 18.6).

Диапазон индексов массива не проверяется. Кроме того, поскольку массивы часто передаются с помощью указателей, нет надежного способа обеспечить доступность для пользователей информации, необходимой для проверки диапазона. Мы рекомендуем вместо массивов использовать класс **vector**.

Размер массива — это сумма размеров его элементов, например:

```
int a[max]; // sizeof(a) == sizeof(int)*max
```

Можно определить и использовать массив массивов (двумерный массив), массив массивов массивов и т.д. (многомерные массивы). Например:

```
double da[100][200][300]; // 300 элементов типа, состоящего из
                          // 200 элементов типа, состоящего из
                          // 100 элементов типа double
da[7][9][11] = 0;
```

Нетривиальное использование многомерных массивов — тонкое и уязвимое для ошибок дело (см. раздел 24.4). Если у вас есть выбор, следует предпочесть класс **Matrix** (такой, как описан в главе 24).

А.8.3. Ссылки

Ссылка (reference) — это *псевдоним* (alias), т.е. альтернативное имя объекта.

```
int  a = 7;
int& r = a;
r = 8;        // Переменная a становится равной 8
```

Ссылки часто используются в качестве параметров функций, чтобы избежать копирования.

```
void f(const string& s);
// ...
f("Чтобы не копировать строку, используем ссылку");
```

(См. разделы 8.5.4–8.5.6.)

А.9. Функции

Функция (function) — это именованный фрагмент кода, получающий (возможно, пустой) набор аргументов и (необязательно) возвращающий значение. Функция объявляется с помощью указания типа возвращаемого значения, за которым следуют ее имя и список параметров.

```
char f(string, int);
```

Здесь **f** — это функция, принимающая аргументы типа **string** и **int** и возвращающая значение типа **char**. Если функция должна быть просто объявлена, но не определена, то ее объявление завершается точкой с запятой. Если функция должна быть определена, то за объявлением аргументов следует тело функции.

```
char f(string s, int i) { return s[i]; }
```

Телом функции должен быть блок (см. раздел 8.2) или блок **try** (см. раздел 5.6.3).

Функция, в объявлении которой указано, что она возвращает какое-то значение, должна его возвращать (используя ключевое слово **return**).

```
char f(string s, int i) {
    char c = s[i];      // Ошибка: ничего не возвращается
}
```

Функция **main()** представляет собой странное исключение из этого правила (см. раздел А.1.2). За исключением функции **main()**, если вы не

хотите возвращать значение, то должны поставить перед именем функции ключевое слово **void**. Другими словами, используйте слово **void** в качестве типа возвращаемого значения.

```
void increment(int& x) { ++x; } // OK: возвращать значение не требуется
```

Функция вызывается с помощью оператора вызова **()** с соответствующим списком аргументов.

```
char x1 = f(1,2); // Ошибка: первый аргумент функции f() должен быть строкой
string s = "Battle of Hastings";
char x2 = f(s);   // Ошибка: функция f() требует два аргумента
char x3 = f(s,2); // OK
```

Более подробную информацию о функциях см. в главе 8.

Определению функции может предшествовать ключевое слово **constexpr**. В этом случае функция должна быть достаточно проста для того, чтобы компилятор мог вычислить ее для аргументов, представляющих собой константные выражения. Функция, объявленная как **constexpr**, может использоваться в константных выражениях (раздел 8.5.9).

A.9.1. Разрешение перегрузки

Разрешение перегрузки (overload resolution) — это процесс выбора вызываемой функции на основе передаваемого множества аргументов. Рассмотрим пример.

```
void print(int);
void print(double);
void print(const std::string&);
print(123);   // Вызывается print(int)
print(1.23);  // Вызывается print(double)
print("123"); // Вызывается print(const string&)
```

Компилятор, руководствуясь правилами языка, может самостоятельно выбрать правильную функцию. К сожалению, эти правила довольно сложные, поскольку они пытаются учесть максимально сложные примеры. Здесь мы приведем лишь их упрощенный вариант.

Выбор правильного варианта перегруженной функции осуществляется на основе поиска наилучшего соответствия между типами аргументов функции и типами ее параметров (формальных аргументов). Для конкретизации нашего представления о выборе наилучшего соответствия сформулируем несколько критериев.

1. Точное совпадение, т.е. совпадение при полном отсутствии преобразований типов или при наличии только тривиальных преобразований (например, преобразование имени массива в указатель, имени функции в указатель на функцию и типа **T** в тип **const T**).

2. Совпадение после повышения, т.е. целочисленные повышения (`bool` в `int`, `char` в `int`, `short` в `int` и их беззнаковые аналоги; см. раздел А.8), а также преобразование типа `float` в `double`.
3. Совпадение после стандартных преобразований, например `int` в `double`, `double` в `int`, `double` в `long double`, `Derived*` в `Base*` (см. раздел 14.3), `T*` в `void*` (см. раздел 17.8), `int` в `unsigned int` (см. раздел 25.5.3).
4. Совпадение после применения пользовательских преобразований (см. раздел А.5.2.3).
5. Совпадение с использованием троеточия (...) в объявлении функции (раздел А.9.3).

Если на наивысшем из уровней, на которых обнаружены совпадения, найдено более одного совпадения, то вызов отклоняется как неоднозначный. Правила разрешения перегрузки ориентированы, в первую очередь, на встроенные числовые типы (см. раздел А.5.3).

Для разрешения перегрузки на основе нескольких аргументов мы сначала должны найти наилучшее совпадение для каждого аргумента. Выбирается та из функций, которая по каждому аргументу подходит так же хорошо, как и остальные функции, но лучше всех остальных соответствует вызову по одному из аргументов; в противном случае вызов считается неоднозначным. Рассмотрим пример.

```
void f(int, const string&, double);
void f(int, const char*, int);

f(1,"hello",1);            // OK: f(int, const char*, int)
f(1,string("hello"),1.0); // OK: f(int, const string&, double)
f(1,"hello",1.0);          // Ошибка: неоднозначность
```

В последнем вызове строка `"hello"` соответствует типу `const char*` без преобразования, а типу `const string&` — только после преобразования. С другой стороны, число `1.0` соответствует типу `double` без преобразования, а число типа `int` — только после преобразования, поэтому ни один из вариантов функции `f()` не соответствует вызову лучше других.

Если эти упрощенные правила не согласуются с тем, что сообщает компилятор, и тем, что кажется вам правильным, в первую очередь, следует предположить, что ваша программа сложнее, чем требуется. Если это так — постарайтесь упростить код; в противном случае проконсультируйтесь с экспертами.

А.9.2. Аргументы по умолчанию

Иногда функции имеют больше аргументов, чем это требуется в наиболее часто встречающихся ситуациях. Для того чтобы учесть это

обстоятельство, программист может предусмотреть аргументы по умолчанию, которые будут использоваться, если при вызове соответствующие аргументы не будут указаны. Рассмотрим пример.

```
void f(int, int=0, int=0);

f(1,2,3);
f(1,2); // Вызов f(1,2,0)
f(1);   // Вызов f(1,0,0)
```

Задавать по умолчанию можно только замыкающие аргументы из списка аргументов функции. Рассмотрим примеры.

```
void g(int, int =7, int); // Ошибка: по умолчанию задан
                          // не замыкающий аргумент
f(1,,1);                  // Ошибка: пропущен второй аргумент
```

Альтернативой аргументам, заданным по умолчанию, может быть перегрузка (и наоборот).

A.9.3. Неопределенные аргументы

Можно задать функцию, не указав ни количество аргументов, ни их типы. Для этого используется троеточие (...), означающее "и, возможно, другие аргументы". Например, вот как выглядят объявление и некоторые вызовы, вероятно, самой известной функции в языке C: **printf()** (см. разделы 27.6.1 и Б.11.2):

```
void printf(const char* format ...); // Получает строку формата
                                     // и, возможно, что-то еще
int x = 'x';
printf("hello, world!");
printf("print a char '%c'\n",x);     // Выводит целое число x
                                     // как символ
printf("print a string \"%s\"",x);   // "Выстрел себе в ногу"
```

Спецификаторы формата в форматной строке, такие как **%c** и **%s**, определяют способ использования аргументов. Как показано выше, это может привести к ужасным последствиям. В языке C++ неопределенных аргументов лучше избегать.

A.9.4. Спецификации связей

Код на языке C++ часто используется вместе с кодом на языке C в одной и той же программе; иначе говоря, одни части программы могут быть написаны на языке C++ (и скомпилированы с помощью компилятора языка C++), а другие — на языке C (и скомпилированы с помощью компилятора языка C). Чтобы упростить использование этой возможности, язык C++ предлагает программистам *спецификации связей* (*linkage specifications*),

указывающие, что та или иная функция подчиняется соглашениям о связях языка C. Соответствующая спецификация помещается перед объявлением функции.

```
extern "C" void callable_from_C(int);
```

В качестве альтернативы ее можно применить ко всем объявлениям в блоке.

```
extern "C" {
    void callable_from_C(int);
    int and_this_one_also(double, int*);
    /* ... */
}
```

Детальную информацию можно найти в разделе 27.2.3.

В языке C нет возможности перегрузки функций, поэтому указывать спецификацию связи с языком C можно не более чем для одной версии перегруженной функции.

A.10. Типы, определенные пользователем

Есть два способа определить новый (пользовательский) тип: в виде класса (**class**, **struct** или **union**; см. раздел A.12) и в виде перечисления (**enum**; см. раздел A.11).

A.10.1. Перегрузка операций

Программист может определить значение большинства операторов, принимающих операнды одного или нескольких пользовательских типов. Изменить стандартный смысл операторов для встроенных типов или ввести новый оператор невозможно. Имя оператора, определенного пользователем (перегруженного оператора), состоит из символа оператора, которому предшествует ключевое слово **operator**; например, имя функции, определяющей оператор +, выглядит как **operator+**:

```
Matrix operator+(const Matrix&, const Matrix&);
```

Примеры можно найти в определениях классов **std::ostream** (главы 10 и 11), **std::vector** (главы 17–19 и раздел Б.4), **std::complex** (раздел Б.9.3) и **Matrix** (глава 24).

Перегрузить можно все операторы за исключением следующих:

```
?:      .*      sizeof   typeid   alignas   noexcept
```

Функции, определяющие следующие операторы, должны быть членами класса:

```
=   []   ()   ->
```

Все остальные операторы можно определить и как члены-функции, и как самостоятельные функции.

Обратите внимание на то, что каждый пользовательский тип имеет операторы = (присваивание и инициализация), & (взятие адреса) и , (запятая), определенные по умолчанию.

При перегрузке операторов следует проявлять умеренность и придерживаться общепринятых соглашений.

A.11. Перечисления

Перечисление (enumeration) определяет тип, содержащий набор именованных значений (*перечислителей*).

```
// "Простое" перечисление:
enum Color { green, yellow, red };
// Перечисление с областью видимости
enum class Traffic_light { yellow, red, green };
```

Перечислители перечисления **enum class** находятся в области видимости перечисления, в то время как перечислители "простого" перечисления **enum** экспортируются в область видимости перечисления. Например:

```
Color col = red;          // OK
Traffic_light tl = red; // Ошибка: нельзя преобразовать
// целочисленное значение (например, Color::red) в Traffic_light
```

По умолчанию первый перечислитель равен нулю, так что **Color::green==0**, а остальные значения увеличиваются на единицу, так что **Color::yellow==1** и **Color::red==2**. Кроме того, можно явно определить значение перечислителя:

```
enum Day { Monday=1, Tuesday,Wednesday };
```

Здесь мы получим **Monday==1**, **Tuesday==2** и **Wednesday==3**.

Перечислители и значения перечислений "простого" **enum** неявно преобразовываются в целые числа, но целые числа не преобразовываются в типы перечислений неявно.

```
int x = green;    // OK: неявное преобразование Color в int
Color c = green; // OK
c = 2;            // Ошибка: нет неявного преобразования
                  // int в Color
c = Color(2);     // OK: (непроверяемое) явное преобразование
int y = c;        // OK: неявное преобразование Color в int
```

Перечислители и значения перечисления **enum class** не конвертируются в целочисленные значения, а целочисленные значения не конвертируются неявно в типы перечислений:

```
int x = Traffic_light::green; // Ошибка: нет неявного преобра-
                              // зования Traffic_light в int
Traffic_light c = green;      // Ошибка: нет неявного преобра-
                              // зования int в Traffic_light
```

Использование перечислений обсуждается в разделе 9.5.

A.12. Классы

Класс (class) — это тип, для которого пользователь определил представление его объектов и операции, допустимые для этих объектов.

```
class X {
public:
    // Пользовательский интерфейс
private:
    // Реализация
};
```

Переменные, функции и типы, определенные в объявлении класса, называются *членами* этого класса. Технические детали изложены в главе 9.

A.12.1. Доступ к членам класса

Открытый (**public**) член класса доступен для пользователей; закрытый (**private**) член класса доступен только членам класса.

```
class Date {
public:
    // ...
    int next_day();
private:
    int y, m, d;
};

void Date::next_day() { return d+1; } // OK

void f(Date d)
{
    int nd = d.d+1; // ошибка: Date::d – закрытый член класса
    // ...
}
```

Структура — это класс, члены которого по умолчанию являются открытыми.

```
struct S {
    // Члены (открытые, если явно не объявлены закрытыми)
};
```

Более подробная информация о доступе к членам класса, включая обсуждение защищенных (**protected**) членов, приведена в разделе 14.3.4.

К членам объекта можно обращаться с помощью оператора . (точка), примененного к его имени, или оператора -> (стрелка), примененного к указателю на него.

```
struct Date {
    int d, m, y;
    int day() const { return d; } // Определена в классе
    int month() const;            // Объявлена; определена
                                  // в другом месте
    int year() const;             // Объявлена; определена
                                  // в другом месте
};

Date x;
x.d = 15;           // Доступ через переменную
int y = x.day();    // Вызов через переменную
Date* p = &x;
p->m = 7;           // Доступ через указатель
int z = p->month(); // Вызов через указатель
```

На члены класса можно ссылаться с помощью оператора :: (разрешение области видимости):

```
int Date::year() const {  // Определение за пределами класса
    return y;
}
```

В функциях-членах класса на другие члены того же класса можно ссылаться, не указывая имя класса.

```
struct Date {
    int d, m, y;
    int day() const { return d; }
    // ...
};
```

Такие имена относятся к объекту, из которого вызвана функция:

```
void f(Date d1, Date d2)
{
    d1.day(); // Обращается к члену d1.d
    d2.day(); // Обращается к члену d2.d
    // ...
}
```

A.12.1.1. Указатель this

Если вы хотите явно указать на объект, из которого вызвана функция-член, то можете использовать предопределенный указатель **this**.

```
struct Date {
    int d, m, y;
    int month() const { return this->m; }
    // ...
};
```

Функция-член, объявленная с помощью модификатора **const** (константная функция-член), не может изменять значение члена объекта, из которого она вызвана.

```
struct Date {
    int d, m, y;
    int month() const { ++m; } // Ошибка: month() —
                               // константная функция
    // ...
};
```

Более подробная информация о константных функциях-членах изложена в разделе 9.7.4.

A.12.1.2. Друзья

Функция, не являющаяся членом класса, может получить доступ ко всем членам-класса, если ее объявить с помощью ключевого слова **friend**. Рассмотрим пример.

```
// Требует доступа к членам классов Matrix и Vector:
Vector operator*(const Matrix&, const Vector&);

class Vector {
    friend Vector operator*(const Matrix&,
                            const Vector&); // Есть доступ
    // ...
};

class Matrix {
    friend Vector operator*(const Matrix&,
                            const Vector&); // Есть доступ
    // ...
};
```

Как показано выше, обычно это относится к функциям, которым нужен доступ к двум классам. Другое предназначение ключевого слова **friend** — обеспечивать доступ для функции, которую нельзя вызывать как функцию-член.

```
class Iter {
public:
    int distance_to(const iter& a) const;
    friend int difference(const Iter& a, const Iter& b);
    // ...
};
```

```
void f(Iter& p, Iter& q)
{
    int x = p.distance_to(q); // Вызов функции-члена
    int y = difference(p,q);  // Вызов функции, не
                              // являющейся членом
    // ...
}
```

Отметим, что функцию, объявленную с помощью ключевого слова **friend**, нельзя объявлять виртуальной.

A.12.2. Определения членов класса

Члены класса, являющиеся целочисленными константами, функциями или типами, могут быть определены/инициализированы как *в* классе (раздел 9.7.3), так и *вне* его (раздел 9.4.4).

```
struct S {
    int c = 1;
    int c2;

    void f() { }
    void f2();

    struct SS { int a; };
    struct SS2;
};
```

Члены, которые не были определены в классе, должны быть определены "где-то в другом месте".

```
int S::c2 = 7;

void S::f2() { }

struct S::SS2 { int m; };
```

Если вам необходимо инициализировать члены-данные значениями, указываемыми при создании объекта, делайте это в конструкторе.

Функции-члены не занимают память, выделенную для объекта.

```
struct S {
    int m;
    void f();
};
```

Здесь **sizeof(S)==sizeof(int)**. На самом деле стандартом это условие не гарантируется, но во всех известных реализациях языка оно выполняется. Следует подчеркнуть, что класс с виртуальной функцией имеет один "скрытый" член, обеспечивающий виртуальные вызовы (см. раздел 14.3.1).

A.12.3. Создание, уничтожение и копирование

Определить смысл инициализации объекта класса можно, определив один или несколько *конструкторов* (constructors). Конструктор — это функция-член, не имеющая возвращаемого типа, имя которой совпадает с именем класса:

```
class Date {
public:
    Date(int yy, int mm, int dd) :y{yy}, m{mm}, d{dd} { }
    // ...
private:
      int y,m,d;
};

Date d1{2006,11,15}; // OK: инициализация с помощью конструктора
Date d2;             // Ошибка: нет инициализации
Date d3{11,15};      // Ошибка: неправильная инициализация
                     // (требуются три инициализатора)
```

Обратите внимание, что данные-члены могут быть инициализированы в конструкторе с помощью списка инициализации (инициализация базового класса и членов). Члены класса инициализируются в порядке их определения в классе.

Конструкторы обычно используются для установления инвариантов класса и захвата ресурсов (см. разделы 9.4.2 и 9.4.3).

Объекты класса создаются "снизу вверх", начиная с объектов базового класса (раздел 14.3.1) в порядке их объявления. Затем в порядке объявления создаются члены класса, после чего следует код самого конструктора. Если программист не сделает чего-нибудь очень странного, это гарантирует, что каждый объект класса будет создан до его использования.

Если конструктор с одним аргументом не объявлен с помощью ключевого слова **explicit**, то он определяет неявное преобразование типа своего аргумента в свой класс:

```
class Date {
public:
    Date(const char*);
    explicit Date(long);  // Целочисленное представление даты
    // ...
};

void f(Date);

Date d1 = "June 5, 1848"; // OK
f("June 5, 1848");        // OK

Date d2 = 2007*12*31+6*31+5; // Ошибка: конструктор Date(long)
                             // объявлен как explicit
```

```
f(2007*12*31+6*31+5);          // Ошибка: конструктор Date(long)
                               // объявлен как explicit

Date d3{2007*12*31+6*31+5};          // OK
Date d4 = Date{2007*12*31+6*31+5}; // OK
f(Date{2007*12*31+6*31+5});          // OK
```

Если базовые классы или члены производного класса не требуют явных аргументов и в классе нет других конструкторов, то автоматически генерируется *конструктор по умолчанию* (default constructor). Этот конструктор инициализирует каждый объект базового класса и каждый член, имеющий конструктор по умолчанию (оставляя члены, не имеющие конструкторов по умолчанию, неинициализированными). Рассмотрим пример.

```
struct S {
    string name, address;
    int x;
};
```

Этот класс `S` имеет неявный конструктор `S()`, инициализирующий члены `name` и `address`, но не `x`. Кроме того, класс без конструктора может быть инициализирован с помощью списка инициализаторов:

```
S s1 {"Hello!"};         // s1 становится { "Hello! ",0 }
S s2 {"Howdy!", 3};
S* p = new S{"G'day!"}; // *p становится { "G'day", 0 };
```

Как видно из приведенного кода, неуказанные замыкающие значения становятся значениями по умолчанию (в данном случае — значение `0` для типа `int`).

A.12.3.1. Деструкторы

Смысл операции удаления объекта (т.е. что происходит, когда объект выходит за пределы области видимости) можно определить с помощью *деструктора* (destructor). Имя деструктора состоит из символа ~ (оператор дополнения), за которым следует имя класса.

```
class Vector {  // Вектор чисел типа double
public:
    // Конструктор:
    explicit Vector(int s) : sz{s}, p{new double[s]} {}
    // Деструктор:
    ~Vector() { delete[] p; }
    // ...
private:
    int sz;
    double* p;
};
```

```
void f(int ss)
{
    Vector v(s);
    // ...
}   // При выходе из функции f() объект v будет уничтожен;
    // для этого будет вызван деструктор класса Vector
```

Деструкторы, вызывающие деструкторы членов класса, могут генерироваться компилятором. Если класс используется как базовый, он обычно должен иметь виртуальный деструктор (см. раздел 17.5.2).

Деструкторы, как правило, используются для "очистки" и освобождения ресурсов.

Объекты класса уничтожаются "сверху вниз", начиная с кода самого деструктора, за которым следуют члены в порядке, обратном порядку их объявления, а затем — объекты базового класса в порядке, обратном порядку их объявления, т.е. строго в порядке, обратном их созданию.

A.12.3.2. Копирование

Вы можете определить смысл операции *копирования* объекта класса:

```
class Vector {      // Вектор чисел типа double
public:
    // Конструктор:
    explicit Vector(int s) : sz{s}, p{new double[s]} { }
    ~Vector() { delete[] p; }          // Деструктор
    Vector(const Vector&);             // Копирующий конструктор
    Vector& operator=(const Vector&); // Копирующее присваивание
    // ...
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(ss);
    Vector v2 = v; // Используем копирующий конструктор
    // ...
    v = v2;        // Используем копирующее присваивание
    // ...
}
```

По умолчанию (т.е. если вы не определили копирующий конструктор и копирующее присваивание) компилятор сам генерирует копирующие операции. Копирование по умолчанию выполняется почленно (см. также разделы 14.2.4 и 18.3).

А.12.3.3. Перемещение

Вы можете также определить смысл операции *перемещения* объекта класса:

```
class Vector {      // Вектор чисел типа double
public:
    // Конструктор:
    explicit Vector(int s) : sz{s}, p{new double[s]} { }
    ~Vector() { delete[] p; }     // Деструктор
    Vector(Vector&&);             // Перемещающий конструктор
    Vector& operator=(Vector&&); // Перемещающее присваивание
    // ...
private:
    int sz;
    double* p;
};

Vector f(int ss)
{
    Vector v(ss);
    // ...
    return v;  // Используется перемещающий конструктор
}
```

По умолчанию (т.е. если вы не определили ни копирующий конструктор, ни копирующее присваивание) компилятор сгенерирует перемещающие операции вместо вас. Перемещение по умолчанию представляет собой почленное перемещение (см. также раздел 18.3.4).

А.12.4. Производные классы

Класс может быть определен как производный от других классов. В этом случае он наследует члены классов, от которых он порожден (своих базовых классов).

```
struct B {
    int mb;
    void fb() { };
};

class D : B {
    int md;
    void fd();
};
```

В данном случае класс `B` имеет два члена: `mb` и `fb()`, в то время как класс `D` — четыре: `mb`, `fb()`, `md` и `fd()`.

Как и члены класса, базовые классы могут быть открытыми и закрытыми (`public` или `private`).

```
Class DD : public B1, private B2 {
    // ...
};
```

В таком случае открытые члены класса B1 становятся открытыми членами класса DD, а открытые члены класса B2 — закрытыми членами класса DD. Производный класс не имеет особых привилегий доступа к членам базового класса, поэтому члены класса DD не имеют доступа к закрытым членам классов B1 и B2.

Если класс имеет несколько непосредственных базовых классов (как, например, класс DD), то говорят, что он использует *множественное наследование* (multiple inheritance).

Указатель на производный класс D может быть неявно преобразован в указатель на его базовый класс B при условии, что класс B является доступным и однозначным по отношению к классу D. Рассмотрим пример.

```
struct B { };
struct B1: B { }; // B  — открытый базовый класс класса B1
struct B2: B { }; // B  — открытый базовый класс класса B2
struct C { };
struct DD : B1, B2, private C {};

DD* p = new DD;
B1* pb1 = p; // OK
B* pb = p;   // Ошибка: неоднозначность: B1::B или B2::B?
C* pc = p;   // Ошибка: DD::C — закрытый класс
```

Аналогично ссылку на производный класс можно неявно преобразовать в ссылку на однозначный и доступный базовый класс.

Более подробную информацию о производных классах можно найти в разделе 14.3. Описание защищенного наследования (**protected**) изложено во многих учебниках повышенной сложности и в справочниках.

A.12.4.1. Виртуальные функции

Виртуальная функция (virtual function) — это функция-член, определяющая интерфейс вызова функций, имеющих одинаковые имена и одинаковые типы аргументов в производных классах. При обращении к виртуальной функции в зависимости от объекта, через который осуществляется обращение, может вызываться одна из этих функций, определенных в производных классах. Говорят, что производный класс *перекрывает* (override) виртуальную функцию-член базового класса.

```
class Shape {
public:
    virtual void draw(); // "virtual" означает
                         // "может быть перекрыта"
    virtual ~Shape() { } // Виртуальный деструктор
    // ...
```

```
};

class Circle : public Shape {
    public:
    void draw();          // Перекрывает функцию Shape::draw()
    ~Circle();            // Перекрывает Shape::~Shape()
    // ...
};
```

По сути, виртуальные функции базового класса (в данном случае класса **Shape**) определяют интерфейс вызова функций производного класса (в данном случае класса **Circle**).

```
void f(Shape& s)
{
    // ...
    s.draw();
}

void g()
{
    Circle c{Point{0,0}, 4};
    f(c); // Вызов функции draw() из класса Circle
}
```

Обратите внимание, что функция **f()** ничего не знает о классе **Circle**: ей известен только класс **Shape**. Объект класса, содержащего виртуальную функцию, содержит один дополнительный указатель, позволяющий найти набор перекрывающих функций (раздел 14.3).

Заметим, что класс, содержащий виртуальные функции, как правило, должен содержать виртуальный деструктор (как, например, класс **Shape**) (см. раздел 17.5.2).

Если вы хотите перекрыть виртуальную функцию базового класса, об этом можно заявить явно, используя суффикс **override**:

```
class Square : public Shape {
public:
    void draw() override;  // Перекрывает Shape::draw()
    ~Circle() override;    // Перекрывает Shape::~Shape()
    void silly() override; // Ошибка: в Shape нет виртуальной
                           // функции Shape::silly()
    // ...
};
```

A.12.4.2. Абстрактные классы

Абстрактный класс (abstract class) — это класс, который можно использовать только в качестве базового класса. Невозможно создать объект абстрактного класса.

```
Shape s;          // Ошибка: класс Shape является абстрактным

class Circle : public Shape {
public:
    void draw(); // Перекрывает Shape::draw()
    // ...
};

Circle c{p,20};  // OK: класс Circle не является абстрактным
```

Наиболее распространенным способом создания абстрактного класса является определение в нем как минимум одной *чисто виртуальной функции* (pure virtual function), т.е. функции, требующей перекрытия.

```
class Shape {
public:
    virtual void draw() = 0; // =0 означает "чисто виртуальная"
    // ...
};
```

(См. раздел 14.3.5.)

Реже, но не менее эффективно абстрактные классы создаются путем объявления всех их конструкторов защищенными (**protected**); см. раздел. 14.2.1.

A.12.4.3. Сгенерированные операции

При определении классов некоторые операции над их объектами будут определены по умолчанию.

- Конструктор по умолчанию.
- Копирующие операции (копирующее присваивание и копирующая инициализация).
- Перемещающие операции (перемещающее присваивание и перемещающая инициализация).
- Деструктор.

Каждый из них (также по умолчанию) рекурсивно применяет те же операции к каждому из базовых классов и членов. Создание производится "снизу вверх", т.е. объект базового класса создается до создания членов производного класса. Члены производного класса и объекты базовых классов создаются в порядке их объявления и уничтожаются в обратном порядке. Таким образом, конструктор и деструктор всегда работают с точно определенными объектами базовых классов и членов производного класса. Рассмотрим пример.

```
struct D : B1, B2 {
    M1 m1;
```

```
    M2 m2;
};
```

Предполагая, что классы **B1**, **B2**, **M1** и **M2** определены, мы можем написать следующий код:

```
D f()
{
    D d;       // Инициализация по умолчанию
    D d2 = d;  // Копирующая инициализация
    d = D{};   // Инициализация по умолчанию, за которой
               // следует копирующее присваивание
    return d; // d перемещается за пределы f()
}   // Объекты d и d2 уничтожаются здесь
```

Например, инициализация объекта **d** по умолчанию вызывает четыре конструктора по умолчанию (в указанном порядке): **B1::B1()**, **B2::B2()**, **M1::M1()** и **M2::M2()**. Если один из этих конструкторов не определен или не может быть вызван, создание объекта **d** невозможно. В инструкции **return** вызываются четыре перемещающих конструктора (в указанном порядке): **B1::B1()**, **B2::B2()**, **M1::M1()** и **M2::M2()**. Если один из этих конструкторов не определен или не может быть вызван, создание объекта **d** невозможно. Уничтожение объекта **d** выполняется путем вызова четырех деструкторов (в указанном порядке): **M2::~M2()**, **M1::~M1()**, **B2::~B2()** и **B1::~B1()**. Если один из этих деструкторов не определен или не может быть вызван, то уничтожение объекта **d** невозможно. Каждый из этих конструкторов и деструкторов может быть либо определен пользователем, либо сгенерирован автоматически.

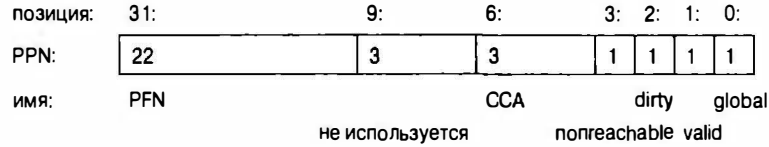
Если класс имеет конструктор, определенный пользователем, то неявный конструктор по умолчанию компилятором не генерируется.

A.12.5. Битовые поля

Битовое поле (bitfield) — это механизм упаковки многих маленьких значений в одно слово или в соответствии с определенным вне программы битовым форматом (например, форматом регистра некоторого устройства). Рассмотрим пример.

```
struct PPN {  // Номер физической страницы R6000
     unsigned int PFN : 22; // Номер кадра страницы
     int : 3;               // Не используется
     unsigned int CCA : 3;  // Алгоритм синхронизации кеша
     bool nonreacheable : 1;
     bool dirty : 1;
     bool valid : 1;
     bool global : 1;
};
```

Упаковка битовых полей в виде слова слева направо приводит к следующему формату (см. раздел 25.5.5).



Битовое поле не обязано иметь имя, но если его нет, то к нему невозможно обратиться.

Как это ни удивительно, но упаковка многих небольших значений в отдельное слово не всегда экономит память. На деле использование таких значений зачастую приводит к излишнему расходу памяти по сравнению с использованием типа `char` или даже `int` для представления одного бита. Причина заключается в том, что для извлечения бита из слова и для записи бита в слово без изменения других битов необходимо выполнить несколько инструкций (которые также хранятся где-то в памяти). Не пытайтесь использовать битовые поля для экономии памяти, если у вас нет очень большого количества объектов с очень маленькими полями данных.

A.12.6. Объединения

Объединение (union) — это класс, в котором все члены расположены в одной и той же области памяти. В каждый момент времени объединение может содержать только один элемент, причем считываться должен только тот элемент объединения, который был записан последним. Рассмотрим пример.

```
union U {
    int x;
    double d;
}

U a;
a.x = 7;
int x1 = a.x; // OK
a.d = 7.7;
int x2 = a.x; // Ой!
```

Правила согласованности чтения и записи членов объединения компилятором не проверяются. Мы вас предупредили.

A.13. Шаблоны

Шаблон (template) — это класс (или функция), параметризованный набором типов и/или целыми числами.

```
template<typename T>
class vector {
public:
    // ...
    int size() const;
private:
    int sz;
    T* p;
};

template<typename T>
int vector<T>::size() const
{
    return sz;
}
```

В списке аргументов шаблона ключевое слово **class** означает тип; его эквивалентной альтернативой является ключевое слово **typename**. Функция-член шаблонного класса по умолчанию является шаблонной функцией с тем же списком аргументов шаблона, что и у класса.

Целочисленные аргументы шаблона должны быть константными выражениями.

```
template<typename T, int sz>
class Fixed_array {
public:
    T a[sz];
    // ...
    int size() const { return sz; };
};

Fixed_array<char,256> x1; // ОК
int var = 226;
Fixed_array<char,var> x2; // Ошибка: неконстантный аргумент
```

A.13.1. Аргументы шаблонов

Аргументы шаблонного класса указываются каждый раз, когда используется его имя.

```
vector<int> v1;    // ОК
vector v2;         // Ошибка: пропущен шаблонный аргумент
vector<int,2> v3; // Ошибка: слишком много аргументов
vector<2> v4;      // Ошибка: ожидается аргумент типа
```

Аргументы шаблона функции обычно выводятся из аргументов функции.

```
template<typename T>
T find(vector<T>& v, int i)
{
    return v[i];
}
```

```
vector<int> v1;
vector<double> v2;
// ...
int x1 = find(v1,2);  // Здесь тип T  - это int
int x2 = find(v2,2);  // Здесь тип T  - это double
```

Можно объявить шаблон функции, для которого невозможно вывести шаблонные аргументы из аргументов ее функции. В этом случае мы должны указать аргументы шаблона явно (точно так же, как для шаблонных классов). Рассмотрим пример.

```
template<typename T, typename U>
T* make(const U& u) { return new T(u); }

int*  pi = make<int>(2);
Node* pn = make<Node>(make_pair("hello",17));
```

Этот код работает, только если объект класса **Node** можно инициализировать объектом класса **pair<const char*,int>** (раздел Б.6.3). Из механизма явной специализации шаблонной функции можно исключать только замыкающие шаблонные аргументы (которые будут выведены на основе аргументов функции).

A.13.2. Инстанцирование шаблонов

Вариант шаблона для конкретного набора шаблонных аргументов называется *специализацией* (specialization). Процесс генерации специализаций на основе шаблона и набора аргументов называется *инстанцированием шаблона* (template instantiation). Как правило, эту задачу решает компилятор, но программист также может самостоятельно определить отдельную специализацию. Обычно это делается, когда общий шаблон для конкретного набора аргументов неприемлем. Рассмотрим пример.

```
template<typename T>
struct Compare {                    // Обобщенное сравнение
    bool operator()(const T& a, const T& b) const
    {
        return a<b;
    }
};

template<>
struct Compare<const char*> {    // Сравнение С-строк
    bool operator()(const char* a, const char* b) const
    {
        return strcmp(a,b) < 0;
    }
};
```

```
Compare<int> c2;          // Обобщенное сравнение
Compare<const char*> c;   // Сравнение С-строк

bool b1 = c2(1,2);        // Обобщенное сравнение
bool b2 = c("asd","dfg"); // Сравнение С-строк
```

Грубым эквивалентом для функций является перегрузка.

```
template<typename T> bool compare(const T& a, const T& b)
{
    return a<b;
}

bool compare (const char* a, const char* b) // Сравнение С-строк
{
    return strcmp(a,b)==0;
}

bool b3 = compare(2,3);             // Обобщенное сравнение
bool b4 = compare("asd","dfg");     // Сравнение С-строк
```

Раздельная компиляция шаблонов (когда в заголовочных файлах содержатся только объявления, а в исходных файлах — единственные определения) не гарантирует переносимость программы, поэтому, если шаблон необходимо использовать в нескольких исходных файлах, в заголовочном файле следует дать его полное определение.

A.13.3. Шаблонные типы членов-классов

Шаблон может иметь как члены, являющиеся типами, так и члены, типами не являющиеся (такие, как данные-члены и функции-члены). Это значит, что в принципе трудно сказать, относится ли имя члена к типу. По техническим причинам, связанным с особенностями языка программирования, компилятор должен знать это, поэтому мы ему должны каким-то образом передать эту информацию. Для этого используется ключевое слово **typename**. Рассмотрим пример.

```
template<typename T> struct Vec {
    typedef T value_type; // Член-тип
    static int count;     // Член-данные
    // ...
};

template<typename T> void my_fct(Vec<T>& v)
{
    int x = Vec<T>::count; // По умолчанию имена членов
                           // считаются относящимися не к типу
    v.count = 7;           // Более простой способ сослаться
                           // на член, не являющийся типом
```

```
    typename                          // Здесь ключевое слово
        Vec<T>::value_type xx = x; // typename необходимо
    // ...
}
```

Более подробная информация о шаблонах приведена в главе 19.

А.14. Исключения

Исключения используются (с помощью инструкции **throw**) для того, чтобы сообщить вызывающей функции об ошибке, которую невозможно обработать локально. Например, сгенерируем исключение **Bad_size** в классе **Vector**.

```
struct Bad_size {
    int sz;
    Bad_size(int s) : sz{s} { }
};

class Vector {
    Vector(int s) { if (s<0 || maxsize<s) throw Bad_size{s}; }
    // ...
};
```

Как правило, мы генерируем тип исключения, определенный специально для представления данной конкретной ошибки. Вызывающая функция может перехватить исключение.

```
void f(int x)
{
    try {
        Vector v(x); // Может генерировать исключения
        // ...
    }
    catch (Bad_size bs) {
        cerr << "Вектор неверного размера (" << bs.sz << ")\n";
        // ...
    }
}
```

Для перехвата всех исключений можно использовать инструкцию **catch(...)**:

```
try {
    // ...
} catch (...) { // Перехват всех исключений
    // ...
}
```

Как правило, лучше (проще, легче, надежнее) применять идиому RAII ("Resource Acquisition Is Initialization" — "захват ресурсов есть инициализация"), чем использовать множество явных инструкций **try** и **catch** (см. раздел 19.5).

Инструкция **throw** без аргументов (т.е. просто **throw;**) повторно генерирует текущее исключение. Рассмотрим пример.

```
try {
    // ...
} catch (Some_exception& e) {
    // локальная очистка
    throw; // Остальное сделает вызывающая функция
}
```

В качестве исключений можно использовать типы, определенные пользователем. В стандартной библиотеке определено несколько типов исключений, которые также можно использовать (раздел Б.2.1). Никогда не используйте в качестве исключений встроенные типы (это может сделать кто-то еще, и ваши исключения могут внести путаницу).

Когда генерируется исключение, система поддержки времени выполнения языка C++ ищет вверх по стеку вызовов раздел **catch**, тип которого соответствует типу генерируемого объекта. Другими словами, она просматривает инструкции **try** в функции, генерирующей исключение, затем в функции, вызвавшей функцию, сгенерировавшую исключение, затем в функции, вызвавшей функцию, вызвавшую функцию..., и так далее, пока не найдет соответствие. Если соответствие не найдено, программа прекращает работу. В каждой функции, обнаруженной на этом пути, и в каждой области видимости, в которой проходит поиск, для выполнения очистки вызываются все деструкторы. Этот процесс называется *раскруткой стека* (stack unwinding).

Объект считается созданным в тот момент, когда заканчивает работу его конструктор. Он уничтожается либо в процессе раскрутки стека, либо при каком-либо ином выходе из области видимости. Отсюда следует, что частично созданные объекты (у которых некоторые члены или базовые объекты созданы, а некоторые — нет), массивы и переменные, находящиеся в области видимости, обрабатываются корректно. Подобъекты уничтожаются тогда и только тогда, когда они уже были созданы.

Не генерируйте исключение, которое может покинуть деструктор. Иначе говоря, деструктор не должен давать сбой. Рассмотрим пример.

```
X::~X() {
    if (in_a_real_mess()) throw Mess(); // Никогда так
}                                       // не делайте!
```

Основная причина этого драконовского правила заключается в том, что если деструктор сгенерирует исключение (и не перехватит его сам) в процессе раскрутки стека, то мы не сможем узнать, какое из исключений следует обрабатывать. Необходимо всеми силами избегать ситуаций, в которых выход из деструктора происходит из-за генерации исключения, поскольку мы не знаем систематического способа создания правильного

кода, в котором это допустимо. В частности, если это произойдет, не гарантируется правильная работа ни одной функции или класса стандартной библиотеки.

A.15. Пространства имен

Пространство имен (namespace) объединяет связанные друг с другом объявления и предотвращает коллизию имен.

```
int a;

namespace Foo {
    int a;
    void f(int i)
    {
        a+= i; // Это переменная a из пространства
               // имен Foo (Foo::a)
    }
}

void f(int);

int main()
{
    a = 7;      // Это глобальная переменная a (::a)
    f(2);       // Это глобальная функция f (::f)
    Foo::f(3); // Это функция f из пространства имен Foo
    ::f(4);     // Это глобальная функция f (::f)
}
```

Имена можно явно квалифицировать именами соответствующих пространств имен (например, `Foo::f(3)`) или оператором разрешения области видимости `::` (например, `::f(2)`), который относится к глобальному пространству имен.

Все имена в пространстве имен (например, в приведенном ниже коде — имена стандартного пространства имен `std`) можно сделать доступными с помощью одной директивы

```
using namespace std;
```

Будьте осторожны с директивой `using`. Удобство, которое она предоставляет, достигается ценой потенциальной коллизии имен. В частности, старайтесь избегать директив `using` в заголовочных файлах. Можно сделать доступным и отдельное имя из пространства имен:

```
using Foo::g;
g(2);       // Это функция g из пространства имен Foo (Foo::g)
```

Более подробная информация о пространствах имен содержится в разделе 8.7.

А.16. Псевдонимы

Для имени можно определить *альтернативное имя*, или *псевдоним* (alias); иначе говоря, можно определить символическое имя, которое будет означать то же самое, что и имя, с которым оно связано (для большинства применений этого имени).

```
using Pint = int*;              // Pint означает указатель на int

namespace
    Long_library_name { /* ... */ }
namespace
    Lib = Long_library_name; // Lib означает Long_library_name

int x = 7;
int& r = x;                     // r означает x
```

Ссылки (см. разделы 8.5.5 и А.8.3) — это механизм времени выполнения для указания на объекты. Псевдонимы **using** (см. раздел 20.5) и **namespace** представляют собой механизмы времени компиляции для указания на имена. В частности, инструкция **using** не вводит новый тип, а просто задает новое имя существующего типа. Рассмотрим пример.

```
using Pchar = char*; // Pchar – это имя типа char*
Pchar p = "Idefix";  // OK: p – это указатель типа char*
char* q = p;         // OK: p и q – указатели типа char*
int x = strlen(p);   // OK: p – указатель типа char*
```

Более старый код на языке программирования C++ вместо **using** для определения псевдонима типа использует ключевое слово **typedef** (раздел 27.3.1):

```
typedef char* Pchar; // Pchar является псевдонимом для char*
```

А.17. Директивы препроцессора

Каждая реализация языка C++ содержит *препроцессор* (preprocessor). В принципе, препроцессор запускается до компилятора и преобразовывает исходный код, написанный нами, в код, который видит компилятор. В действительности это действие встроено в компилятор и не представляет интереса, за исключением того, что оно может вызывать проблемы. Каждая строка, начинающаяся символом #, представляет собой директиву препроцессора.

A.17.1. Директива #include

Мы широко использовали препроцессор для включения заголовочных файлов:

```
#include "file.h"
```

Эта директива указывает препроцессору, что он должен включить содержимое файла `file.h` в исходный текст вместо самой директивы. Для стандартных заголовочных файлов используются не кавычки (`"..."`), а угловые скобки (`<...>`), например:

```
#include<vector>
```

Это рекомендованная запись для включения стандартных заголовочных файлов.

A.17.2. Директива #define

Препроцессор выполняет также определенные манипуляции символами, которые называются *макроподстановками* (macro substitution). Например, можно определить имя символьной строки.

```
#define FOO bar
```

Теперь везде, где препроцессор увидит символы `FOO`, они будут заменены символами `bar`.

```
int FOO = 7;
int FOOL = 9;
```

В таком случае компилятор увидит следующий текст:

```
int bar = 7;
int FOOL = 9;
```

Обратите внимание, что препроцессор знает об именах языка C++ достаточно много, чтобы не заменить символы `FOO`, являющиеся частью слова `FOOL`.

С помощью директивы `define` можно также определить макросы, принимающие параметры.

```
#define MAX(x,y) (((x)>(y))?(x) : (y))
```

Их можно использовать следующим образом:

```
int xx = MAX(FOO+1,7);
int yy = MAX(++xx,9);
```

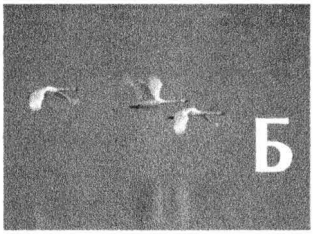
Эти выражения будут развернуты так:

```
int xx = (((bar+1)>( 7))?(bar+1) : (7));
int yy = (((++xx)>( 9))?(++xx) : (9));
```

Заметим, что скобки необходимы для того, чтобы получить правильный результат при вычислении выражения `FOO+1`. Кроме того, был дважды выполнен инкремент переменной **xx**, причем совершенно неочевидным образом. Макросы чрезвычайно популярны в основном потому, что у программистов на языке C было мало альтернатив. Обычные заголовочные файлы содержат определения тысяч макросов. Будьте осторожны!

Если уж вам приходится использовать макросы, то называйте их, используя только прописные буквы, например **ALL_CAPITAL_LETTERS**; при этом обычные имена в программе не должны состоять только из прописных букв. Прислушайтесь к хорошему совету и не слушайте тех, кто поступает иначе (например, в одном из вполне авторитетных и добропорядочных заголовочных файлов мы нашли макрос **max**).

(См. также раздел 27.8.)



Обзор стандартной библиотеки

По возможности вся сложность должна быть скрыта от посторонних взглядов.

— Дэвид Дж. Уилер (David J. Wheeler)

Это приложение содержит краткий обзор основных возможностей стандартной библиотеки языка C++. Изложенная в нем информация носит выборочный характер и предназначена для новичков, желающих получить общее представление о возможностях стандартной библиотеки и узнать немного больше, чем написано в основном тексте книги.

Б.1. Обзор

Это приложение является справочником и не предназначено для последовательного чтения от начала до конца, как обычная глава. В нем более или менее систематично описываются основные элементы стандартной

библиотеки языка C++. Впрочем, этот справочник не полон; он представляет собой лишь краткий обзор с немногочисленными примерами, иллюстрирующими ключевые возможности. За более подробным объяснением читателям часто придется обращаться к соответствующим главам данной книги. Кроме того, следует подчеркнуть, что мы не стремились обеспечить точность и терминологию, эквивалентные точности и терминологии стандарта. Более подробную информацию читатели найдут в книге Stroustrup, *The C++ Programming Language*. Определение языка C++ изложено в стандарте ISO C++, но этот документ не подходит для новичков. Впрочем, он для них и не был предназначен. Не забудьте о возможности использовать документацию, имеющуюся в сети.

Какая польза от такого выборочного (а значит, неполного) обзора? Вы можете быстро найти известную вам операцию или бегло просмотреть раздел в поисках доступных операций. Вы можете найти очень подробную информацию в других источниках, но что конкретно искать, вам подскажет именно этот краткий обзор. В этом приложении содержатся перекрестные ссылки на учебный материал из других глав, а также кратко изложены возможности стандартной библиотеки. Пожалуйста, не старайтесь запомнить изложенные в нем сведения; они предназначены не для этого. Наоборот, это приложение позволит вам избавиться от необходимости запоминать лишнее.

Здесь вы можете найти готовые средства, вместо того, чтобы изобретать их самостоятельно. Все, что есть в стандартной библиотеке (и особенно все, что перечислено в приложении), оказалось весьма полезным для многих людей. Стандартные возможности библиотеки практически всегда разработаны, реализованы и документированы намного лучше, чем это можете сделать вы, находясь в цейтноте. Кроме того, их переносимость из одной системы в другую обеспечена намного лучше. Таким образом, по возможности всегда следует отдавать предпочтение стандартным библиотечным (а не "самопальным") средствам. В таком случае ваш код будет намного понятнее.

Если вы чувствительная натура, то огромное количество возможностей может вас напугать. Не бойтесь, просто игнорируйте то, что вам не нужно. Если же вы дотошный человек, то обнаружите, что о многом мы не сказали. Полнота нужна лишь для справочников, предназначенных для экспертов, и для онлайн-документации. В любом случае многое покажется вам загадочным и, возможно, интересным. Постигайте эти тайны!

Б.1.1. Заголовочные файлы

Интерфейсы средств стандартной библиотеки определены в заголовочных файлах. Используйте данный раздел, чтобы узнать, какие средства вам доступны, а также разобраться, где они определены и описаны.

STL (контейнеры, итераторы и алгоритмы)	
`<algorithm>`	Алгоритмы; `sort()`, `find()` и т.д. (разделы 21.1 и Б.5)
`<array>`	Массив фиксированного размера (раздел 20.9)
`<bitset>`	Массив `bool` (раздел 25.5.2)
`<deque>`	Двусторонняя очередь
`<functional>`	Функциональные объекты (раздел Б.6.2)
`<iterator>`	Итераторы (раздел Б.4.4)
`<list>`	Двусвязный список (разделы 20.4, Б.4)
`<forward_list>`	Односвязный список
`<map>`	Классы `map` и `multimap` (разделы 21.6.1–21.6.3, Б.4)
`<memory>`	Распределители памяти для контейнеров
`<queue>`	Классы `queue` и `priority_queue`
`<set>`	Классы `set` и `multiset` (разделы 21.6.5 и Б.4)
`<stack>`	Класс `stack`
`<unordered_map>`	Хешированные отображения (раздел 21.6.4)
`<unordered_set>`	Хешированные множества
`<utility>`	Операторы и класс `pair` (раздел Б.6.3)
`<vector>`	Класс `vector` (динамически расширяемый) (разделы 20.8 и Б.4)

Потоки ввода-вывода	
`<iostream>`	Объекты потоков ввода-вывода (раздел Б.7)
`<fstream>`	Файловые потоки (раздел Б.7.1)
`<sstream>`	Строковые потоки (раздел Б.7.1)
`<iosfwd>`	Объявление (но не определение) средств потоков ввода-вывода
`<ios>`	Базовые классы потоков ввода-вывода
`<streambuf>`	Потоковые буфера
`<istream>`	Потоки ввода (раздел Б.7)
`<ostream>`	Потоки вывода (раздел Б.7)
`<iomanip>`	Форматирование и манипуляторы (раздел Б.7.6)

Работа со строками	
`<string>`	Класс `string` (раздел Б.8.2)
`<regex>`	Регулярные выражения (глава 23)

Работа с числами	
`<complex>`	Арифметика комплексных чисел (раздел Б.9.3)
`<random>`	Генерация случайных чисел (раздел Б.9.6)
`<valarray>`	Числовые массивы
`<numeric>`	Обобщенные численные алгоритмы, например `accumulate()` (раздел Б.9.5)
`<limits>`	Предельные значения (раздел Б.9.1)

Вспомогательные средства и языковая поддержка	
`<exception>`	Типы исключений (раздел Б.2.1)
`<stdexcept>`	Иерархия исключений (раздел Б.2.1)
`<locale>`	Форматирование с учетом местных особенностей
`<typeinfo>`	Стандартная информация о типах (оператор `typeid`)
`<new>`	Функции для распределения памяти и ее освобождения
`<memory>`	Указатели для управления ресурсами, например `unique_ptr` (раздел Б.6.5)

Поддержка параллельных вычислений	
`<thread>`	Потоки (выходят за рамки данной книги)
`<future>`	Межпоточные коммуникации (выходят за рамки данной книги)
`<mutex>`	Взаимоисключения (выходят за рамки данной книги)

Стандартная библиотека языка C	
`<cstring>`	Работа с C-строками (раздел Б.11.3)
`<cstdio>`	Ввод-вывод в стиле C (раздел Б.11.2)
`<ctime>`	Функции `clock()`, `time()` и т.д. (раздел Б.11.5)
`<cmath>`	Стандартные математические функции для работы с числами с плавающей точкой (раздел Б.9.2)
`<cstdlib>`	Функции `abort()`, `abs()`, `malloc()`, `qsort()` и т.д. (глава 27)
`<cerrno>`	Обработка ошибок в стиле C (раздел 24.8)
`<cassert>`	Макрос `assert` (раздел 27.9)
`<clocale>`	Форматирование с учетом местных особенностей
`<climits>`	Предельные значения в стиле C (раздел Б.9.1)
`<cfloat>`	Предельные значения чисел с плавающей точкой (раздел Б.9.1)
`<cstddef>`	Поддержка языка C; `size_t` и пр.
`<cstdarg>`	Макрос для обработки переменного количества аргументов
`<csetjmp>`	Функции `setjmp()` и `longjmp()` (никогда их не используйте)
`<csignal>`	Обработка сигналов
`<cwchar>`	Широкие символы
`<cctype>`	Классификация символов (раздел Б.8.1)
`<cwctype>`	Классификация широких символов

Для каждого заголовка стандартной библиотеки языка C существует аналогичный заголовочный файл без первой буквы `c` в имени и с расширением `.h`, например заголовочный файл `<time.h>` для заголовочного файла `<ctime>`. В отличие от файлов без расширения, версии заголовочных файлов с расширением `.h` определяют глобальные имена, а не имена в пространстве имен `std`.

Некоторые — но не все — средства, определенные в этих заголовочных файлах, описаны в следующих разделах и главах основного текста книги.

Если вам необходима более полная информация, обратитесь к онлайн-документации или к книге по языку C++ уровня экспертов.

Б.1.2. Пространство имен std

Средства стандартной библиотеки определены в пространстве имен **std**, поэтому, чтобы использовать их, необходимо указать их явную квалификацию, выполнить объявление **using** или директиву **using**.

```
std::string s;          // Явная квалификация

using std::vector;      // Объявление using
vector<int>v(7);

using namespace std;    // Директива using
map<string,double> m;
```

В этой книге для доступа к пространству имен **std** мы использовали директиву **using**. Будьте очень осторожны при использовании директив **using** (см. раздел А.15).

Б.1.3. Стиль описания

Полное описание даже простой операции из стандартной библиотеки, например конструктора или алгоритма, может занять несколько страниц. По этой причине мы используем чрезвычайно лаконичный стиль представления. Рассмотрим пример.

Пример обозначений	
p=op(b,e,x)	Функция **op** выполняет некоторые действия над диапазоном [**b**,**e**) и переменной **x**, возвращая переменную **p**
foo(x)	**foo()** делает что-то с переменной **x**, но не возвращает никакого результата
bar(b,e,x)	Должен ли объект **x** делать что-то с диапазоном [**b**,**e**)?

Мы старались выбирать мнемонические идентификаторы, поэтому символы **b** и **e** будут обозначать итераторы, задающие начало и конец диапазона; **p** — указатель или итератор; **x** — некое значение, полностью зависящее от контекста. В этой системе обозначений отличить функцию, не возвращающую никакого результата, от функции, возвращающей переменную булева типа, без дополнительных комментариев невозможно, поэтому, если не приложить дополнительных усилий, их можно перепутать. Для операций, возвращающих переменную типа **bool**, в объяснении обычно стоит вопросительный знак.

Если алгоритмы следуют общепринятым соглашениям, возвращая конец входной последовательности для обозначения событий "отказ", "не найден" и т.п. (раздел Б.3.1), то мы это явно не указываем.

Б.2. Обработка ошибок

Стандартная библиотека состоит из компонентов, которые разрабатывались более сорока лет. По этой причине ее стиль и принципы обработки ошибок являются несогласованными.

- Библиотека в стиле языка C состоит из функций, многие из которых для индикации ошибок устанавливают значение переменной **errno** (см. раздел 24.8).
- Многие алгоритмы, работающие с последовательностями элементов, возвращают итератор, установленный на элемент, следующий за последним, отмечая тем самым, что произошла ошибка или искомый элемент не найден.
- Библиотека потоков ввода-вывода для сообщений об ошибках использует состояние каждого потока и может (если пользователь этого потребует) генерировать исключения (см. разделы 10.6 и Б.7.2).
- Некоторые компоненты стандартной библиотеки, такие как **vector**, **string** и **bitset**, при обнаружении ошибок генерируют исключения.

Стандартная библиотека разработана так, чтобы все ее средства удовлетворяли "базовой гарантии" (см. раздел 19.5.3). Иначе говоря, даже если исключение сгенерировано, ни один ресурс (например, память) не будет потерян и ни один инвариант класса из стандартной библиотеки не будет нарушен.

Б.2.1. Исключения

Некоторые средства стандартной библиотеки сообщают об ошибках, генерируя исключения.

Стандартные исключения	
bitset	Генерирует исключения **invalid_argument**, **out_of_range** и **overflow_error**
dynamic_cast	Генерирует исключение **bad_cast**, если не может выполнить преобразование
iostream	Генерирует исключение **ios_base::failure**, если исключения включены
new	Генерирует исключение **bad_alloc**, если не может выделить память
regex	Генерирует исключение **regex_error**
string	Генерирует исключения **length_error** и **out_of_range**
typeid	Генерирует исключение **bad_typeid**, если не может вернуть объект **type_info**
vector	Генерирует исключение **out_of_range**

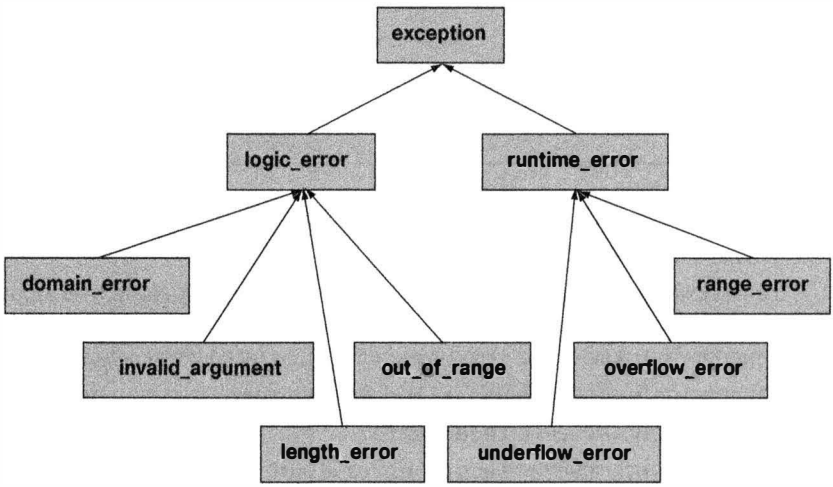
Эти исключения могут возникнуть в любом коде, явно или неявно использующем указанные средства библиотеки. Целесообразно всегда перехватывать один из корневых классов иерархии исключений стандартной библиотеки (такой, как **exception**) в каком-то месте программы (например, в функции **main()**). Это можно не делать, только если вы точно *знаете*, что все использованные средства были применены так, что никакие исключения просто не могут быть сгенерированы.

Мы настоятельно рекомендуем не генерировать исключения встроенных типов, например числа типа **int** или С-строки. Вместо этого следует генерировать объекты типов, специально разработанных для использования в качестве исключений. Для этого можно использовать класс, производный от стандартного библиотечного класса **exception**.

```
class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
```

Функцию **what()** можно использовать для получения строки, предназначенной для указания информации об ошибке, вызвавшей исключение.

Приведенная ниже иерархия стандартных исключений может оказаться полезной для лучшего понимания исключений стандартной библиотеки.



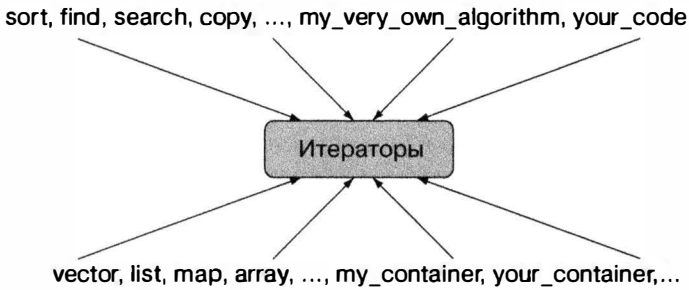
Можете определить собственное исключение, породив его из стандартного библиотечного исключения, например, следующим образом:

```
struct My_error : runtime_error {
    My_error(int x) : interesting_value{x} { }
```

```
    int interesting_value;
    const char* what() const override { return "My_error"; }
};
```

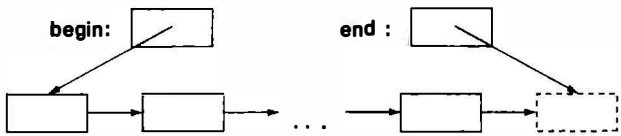
Б.3. Итераторы

Итераторы — это клей, скрепляющий алгоритмы стандартной библиотеки с их данными. Итераторы можно также назвать механизмом, минимизирующим зависимость алгоритмов от структур данных, с которыми они работают (см. раздел 20.3).



Б.3.1. Модель итераторов

Итератор — это аналог указателя, в котором реализованы операции косвенного доступа (например, оператор * для разыменования) и перехода к новому элементу (например, оператор ++ для перехода к следующему элементу). Последовательность элементов определяется парой итераторов, задающих полуоткрытый диапазон [**begin**,**end**).



Таким образом, итератор **begin** указывает на первый элемент последовательности, а итератор **end** — на элемент, следующий за последним элементом последовательности. Никогда не считывайте и не записывайте значение ***end**. Для пустой последовательности всегда выполняется условие **begin==end**. Другими словами, для любого итератора **p** последовательность [**p**,**p**) является пустой.

Для того чтобы читать последовательность, алгоритм обычно получает пару итераторов (**b**,**e**) и перемещается по элементам с помощью оператора **++**, пока не достигнет конца последовательности.

```
while (b!=e) {     // Используйте !=, а не <
    // Какие-то операции
```

```
    ++b;          // Переход к следующему элементу
}
```

Алгоритмы, выполняющие поиск элемента в последовательности, в случае неудачи обычно возвращают итератор, установленный на конец последовательности. Рассмотрим пример.

```
p = find(v.begin(),v.end(),x); // Ищем x в последовательности v
if (p != v.end()) {
    // x найден в элементе p
}
else {
    // x в диапазоне [v.begin(),v.end()) не найден
}
```

(См. раздел 20.3.)

Алгоритмы, записывающие элементы последовательности, часто получают только один итератор, указывающий на ее первый элемент. В таком случае программист сам несет ответственность за предотвращение выхода за пределы этой последовательности. Рассмотрим пример.

```
template<typename Iter> void f(Iter p, int n)
{
    while (n>0) *p++ = --n;
}

vector<int> v(10);
f(v.begin(),v.size()); // OK
f(v.begin(),1000);     // Большая проблема
```

Некоторые реализации стандартной библиотеки проверяют выход за пределы допустимого диапазона, т.е. генерируют исключение при последнем вызове функции `f()`, но этот код нельзя считать переносимым; многие реализации такую проверку не выполняют.

Перечислим операции над итераторами.

Операции над итераторами	
`++p`	Префиксный инкремент: устанавливает итератор `p` на следующий элемент последовательности или на элемент, следующий за последним ("на один элемент вперед"); результатом является значение `p+1`
`p++`	Постфиксный инкремент: устанавливает итератор `p` на следующий элемент последовательности или на элемент, следующий за последним ("на один элемент вперед"); результатом является значение `p` (до инкремента)
`--p`	Префиксный декремент: устанавливает итератор `p` на предыдущий элемент ("на один элемент назад"); результатом является значение `p-1`
`p--`	Постфиксный декремент: устанавливает итератор `p` на предыдущий элемент ("на один элемент назад"); результатом является значение `p` (до декремента)
`*p`	Доступ (разыменование): `*p` ссылается на элемент, на который указывает итератор `p`

Операции над итераторами	
`p[n]`	Доступ (индексирование): `p[n]` ссылается на элемент, на который указывает итератор `p+n`; эквивалент выражения `*(p+n)`
`p->m`	Доступ (доступ к члену); эквивалент выражения `(*p).m`
`p==q`	Равенство: истина, если итераторы `p` и `q` указывают на один и тот же элемент или если оба указывают на элемент, следующий за последним
`p!=q`	Неравенство: `!(p==q)`
`p<q`	Указывает ли итератор `p` на элемент, расположенный до элемента, на который указывает итератор `q`?
`p<=q`	`p<q || p==q`
`p>q`	Указывает ли итератор `p` на элемент, расположенный после элемента, на который указывает итератор `q`?
`p>=q`	`p>q || p==q`
`p+=n`	Перемещение на `n` элементов: устанавливает итератор `p` на `n`-й элемент, считая вперед от элемента, на который он ссылается в данный момент
`p-=n`	Перемещение на `-n` элементов: устанавливает итератор `p` на `n`-й элемент, считая назад от элемента, на который он ссылается в данный момент
`q=p+n`	Итератор `q` ссылается на `n`-й элемент, считая вперед от элемента, на который ссылается итератор `p`
`q=p-n`	Итератор `q` ссылается на `n`-й элемент, считая назад от элемента, на который ссылается итератор `p`; после его выполнения `q+n==p`
`advance(p,n)`	Перемещение вперед: аналог выражения `p+=n`; функцию `advance()` можно использовать, даже если итератор `p` не является итератором произвольного доступа; эта операция может потребовать выполнения `n` шагов (по списку)
`x=distance(p,q)`	Подобно выражению `q-p` функцию `distance()` можно использовать, даже если итератор `p` не является итератором произвольного доступа; эта операция может потребовать выполнения `n` шагов (по списку)

Обратите внимание на то, что не каждый вид итераторов (раздел Б.3.2) поддерживает все операции над итераторами.

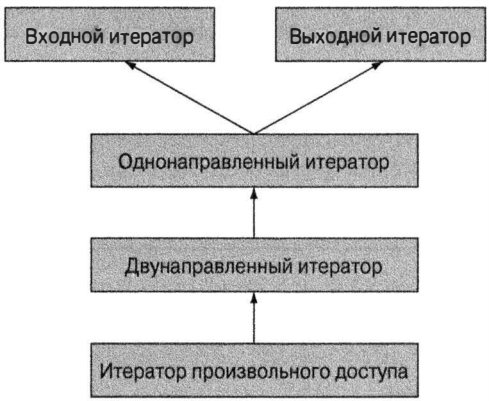
Б.3.2. Категории итераторов

В стандартной библиотеке предусмотрены пять разновидностей итераторов (категорий итераторов).

Категории итераторов	
Входной итератор (итератор для чтения) (input iterator)	Можем перемещаться вперед с помощью оператора `++` и однократно считывать каждый элемент с помощью оператора `*`. Итераторы можно сравнивать с помощью операторов `==` и `!=`. Этот вид итераторов предоставляется классом `istream` (см. раздел 21.7.2)
Выходной итератор (итератор для записи) (output iterator)	Можем перемещаться вперед с помощью оператора `++` и однократно записывать каждый элемент с помощью оператора `*`. Этот вид итераторов предоставляется классом `ostream` (см. раздел 21.7.2)

Категории итераторов	
Однонаправленный итератор (forward iterator)	Можем перемещаться вперед, многократно применяя оператор ++, а также считывать и записывать (неконстантные) элементы с помощью оператора *. Если итератор указывает на объект класса, то для доступа к его членам можно использовать оператор ->
Двунаправленный итератор (bidirectional iterator)	Можем перемещаться вперед (используя оператор ++) и назад (используя оператор --), а также считывать и записывать (неконстантные) элементы с помощью оператора *. Этот вид итераторов реализован в классах **list**, **map** и **set**
Итератор произвольного доступа (random-access iterator)	Можем перемещаться вперед (с помощью операторов ++ и +=) и назад (с помощью операторов -- и -=), а также считывать и записывать (не константные) элементы с помощью операторов * и []. Можно применять индексацию, добавлять к итератору произвольного доступа целое число с помощью оператора +, а также вычитать из него целое число с помощью оператора -. Можно вычислить расстояние между двумя такими итераторами, указывающими на одну и ту же последовательность, вычитая один из другого. Итераторы произвольного доступа можно сравнивать с помощью операторов <, <=, > и >=. Этот вид итераторов реализован в классе **vector**

Логически итераторы образуют следующую иерархию (см. раздел 20.8).



Поскольку категории итераторов не являются классами, эту иерархию нельзя считать иерархией классов, реализованной с помощью наследования. Если вам требуется выполнить над итераторами нетривиальные действия, поищите класс **iterator_traits** в профессиональном справочнике.

Каждый контейнер имеет собственные итераторы конкретной категории:

- **vector** — итераторы произвольного доступа;
- **list** — двунаправленные итераторы;
- **forward_list** — однонаправленные итераторы;
- **deque** — итераторы произвольного доступа;
- **bitset** — итераторов нет;

- `set` — двунаправленные итераторы;
- `multiset` — двунаправленные итераторы;
- `map` — двунаправленные итераторы;
- `multimap` — двунаправленные итераторы;
- `unordered_set` — однонаправленные итераторы;
- `unordered_multiset` — однонаправленные итераторы;
- `unordered_map` — однонаправленные итераторы;
- `unordered_multimap` — однонаправленные итераторы.

Б.4. Контейнеры

Контейнер содержит последовательность элементов. Элементы этой последовательности имеют тип, который является типом-членом `value_type` контейнера. Наиболее полезными контейнерами являются следующие.

Последовательные контейнеры	
`array<T,N>`	Массив фиксированного размера, состоящий из `N` элементов типа `T`
`deque<T>`	Двусторонняя очередь
`list<T>`	Двусвязный список
`forward_list<T>`	Односвязный список
`vector<T>`	Динамический массив элементов типа `T`

Ассоциативные контейнеры	
`map<K,V>`	Отображение элементов типа `K` на элементы типа `V`; последовательность пар (`K`, `V`)
`multimap<K,V>`	Отображение `K` на `V`; допускаются дубликаты ключей
`set<K>`	Множество элементов типа `K`
`multiset<K>`	Множество элементов типа `K` (допускаются дубликаты)
`unordered_map<K,V>`	Отображение элементов типа `K` на элементы типа `V`, использующее хеш-функцию
`unordered_multimap<K,V>`	Отображение элементов типа `K` на элементы типа `V`, использующее хеш-функцию; допускаются дубликаты ключей
`unordered_set<K>`	Множество элементов типа `K`, использующее хеш-функцию
`unordered_multiset<K>`	Множество элементов типа `K`, использующее хеш-функцию; допускаются дубликаты ключей

Упорядоченные ассоциативные контейнеры (`map`, `set` и т.п.) имеют дополнительный шаблонный аргумент, указывающий тип, используемый для сравнения; например, `set<K,C>` использует `C` для сравнения значений `K`.

Эти контейнеры определены в заголовочных файлах `<vector>`, `<list>` и др. (см. раздел Б.1.1). Последовательные контейнеры занимают непрерыв-

Адаптеры контейнеров	
`priority_queue<T>`	Очередь с приоритетами
`queue<T>`	Очередь с функциями `push()` и `pop()`
`stack<T>`	Стек с функциями `push()` и `pop()`

ную область памяти или представляют собой связанные списки, содержащие элементы соответствующего типа `value_type` (выше мы обозначали его буквой `T`). Ассоциативные контейнеры представляют собой связанные структуры (деревья) с узлами соответствующего типа `value_type` (выше мы обозначали его как `pair(K,V)`). Последовательность элементов в контейнерах `set`, `map` или `multimap` упорядочена по значениям ключей (`K`). Последовательность в контейнерах, название которых начинается со слова `unordered`, не имеет гарантированного порядка. Контейнер `multimap` отличается от контейнера `map` тем, что в первом случае значение ключа может повторяться много раз. Адаптеры контейнеров — это контейнеры со специальными операциями, созданные из других контейнеров.

Если сомневаетесь, используйте класс `vector`. Если у вас нет весомой причины использовать другой контейнер, используйте класс `vector`.

Для выделения и освобождения памяти (см. раздел 19.3.7) контейнеры используют распределители памяти. Мы не описываем их здесь; при необходимости читатели найдут информацию о них в профессиональных справочниках. По умолчанию распределитель памяти использует операторы `new` и `delete`, для того чтобы занять или освободить память, необходимую для элементов контейнера.

Там, где это целесообразно, операция доступа реализована в двух вариантах: один — для константных объектов, другой — для неконстантных (см. раздел 18.5).

В этом разделе перечислены общие и "почти общие" члены стандартных контейнеров (более подробную информацию см. в главе 20). Члены, характерные для какого-то конкретного контейнера, такие как функция `splice()` из класса `list`, не указаны; их описание можно найти в профессиональных справочниках.

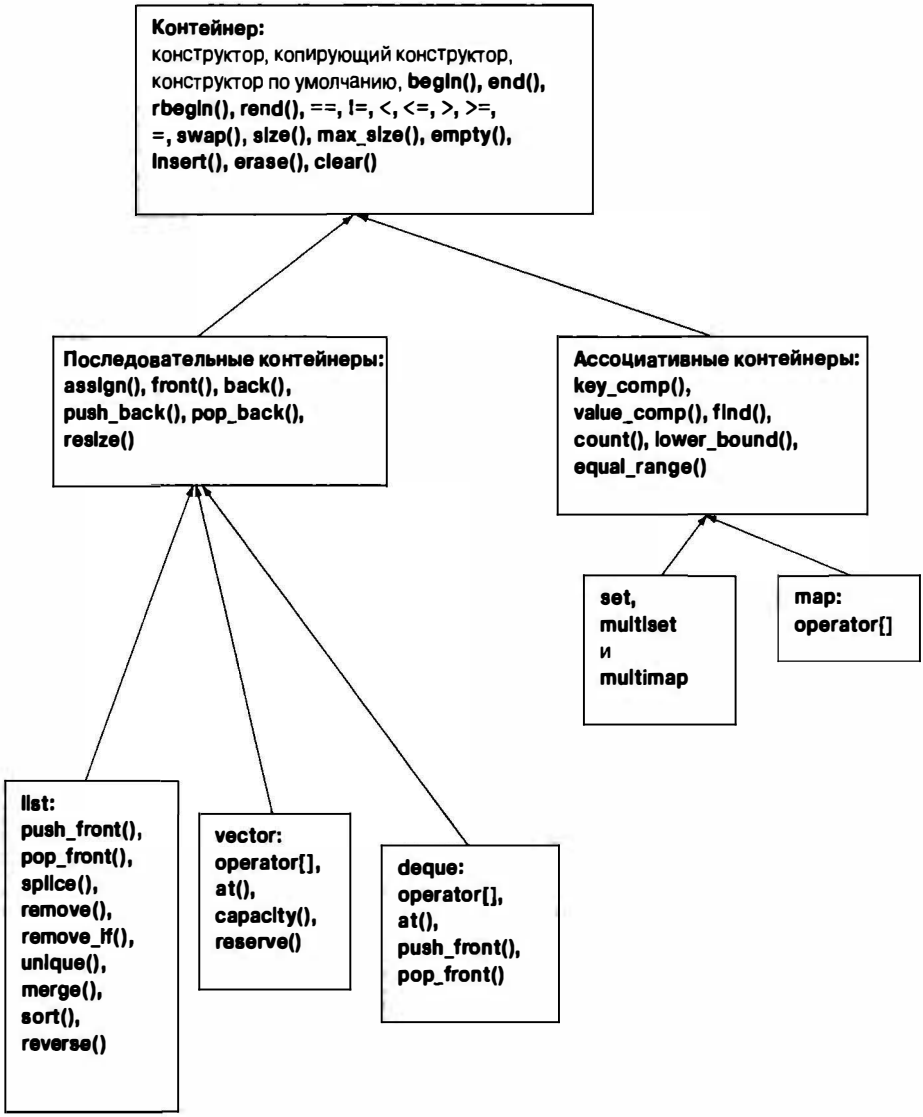
Некоторые типы данных обеспечивают большинство операций, требующихся от стандартного контейнера, но все-таки не все. Иногда такие типы называют "почти контейнерами". Перечислим наиболее интересные из них.

"Почти контейнеры"	
`T[n]` (встроенный массив)	Нет функции `size()` и других функций-членов. Если есть возможность выбора, в качестве альтернативы рекомендуем использовать контейнеры, такие как `vector`, `string` или `array`
`string`	Содержит только символы, но обеспечивает операции, полезные для работы с текстом, например конкатенацию (`+` и `+=`). Рекомендуем использовать стандартный класс `string` вместо обычных строк

"Почти контейнеры"	
`valarray`	Числовой вектор с векторными операциями, но со многими ограничениями, обусловленными стремлением к высокопроизводительной реализации. Рекомендуем использовать, только если требуется выполнять очень много векторной арифметики

Б.4.1. Обзор

Операции, предусмотренные в стандартных контейнерах, можно проиллюстрировать следующим образом.



Мы не приводим здесь `array` и `forward_list`, поскольку они не полностью соответствуют идеалам взаимозаменяемости стандартной библиотеки.

- `array` не является дескриптором, не может изменять количество своих элементов после инициализации и должен быть инициализирован с помощью списка инициализации, а не конструктора.
- `forward_list` не поддерживает ряд важных операций, в частности у него нет функции `size()`. Лучше всего рассматривать этот контейнер как оптимизированный для пустых и почти пустых последовательностей.

Б.4.2. Члены-типы

Контейнер определяет ряд типов, являющихся его членами.

Типы-члены	
`value_type`	Тип элементов
`size_type`	Тип индексов, счетчиков элементов и т.п.
`difference_type`	Тип разности между итераторами
`iterat`	Аналог указателя `value_type*`
`const_iterat`	Аналог указателя `const value_type*`
`reverse_iterat`	Аналог указателя `value_type*`
`const_reverse_iterator`	Аналог указателя `const value_type*`
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	Аналог указателя `value_type*`
`const_pointer`	Аналог указателя `const value_type*`
`key_type`	Тип ключа (только для ассоциативных контейнеров)
`mapped_type`	Тип отображаемого значения (только для ассоциативных контейнеров)
`key_compare`	Тип критерия для сравнения (только для ассоциативных контейнеров)
`allocator_type`	Тип диспетчера памяти

Б.4.3. Конструкторы, деструкторы и присваивания

Контейнеры имеют много разнообразных конструкторов и операций присваивания. Перечислим конструкторы, деструкторы и операции присваивания для контейнера `C` (например, типа `vector<double>` или `map<string,int>`).

Конструкторы, деструкторы и присваивания	
`C c;`	`c` — пустой контейнер
`C{}`	Создает пустой контейнер

Конструкторы, деструкторы и присваивания	
`C c(n);`	Контейнер `c` инициализируется `n` элементами, значения которых заданы по умолчанию (не для ассоциативных контейнеров)
`C c(n,x);`	Контейнер `c` инициализируется `n` копиями объекта `x` (не для ассоциативных контейнеров)
`C c{b,e};`	Контейнер `c` инициализируется элементами из диапазона [`b`,`e`)
`C c{elems};`	Контейнер `c` инициализируется элементами из списка инициализации `elems`.
`C c{c2};`	Контейнер `c` представляет собой копию контейнера `c2`
`~C()`	Уничтожение контейнера типа `C` и всех его элементов (обычно вызывается неявно)
`c1=c2`	Копирующее присваивание; копирует все элементы из контейнера `c2` в контейнер `c1`; после присваивания выполняется условие `c1==c2`
`c.assign(n,x)`	Присваивает `n` копий объекта `x` (не для ассоциативных контейнеров)
`c.assign(b,e)`	Присваивает объекты из диапазона [`b`,`e`)

Обратите внимание, что для некоторых контейнеров и типов элементов конструктор или копирование элементов может генерировать исключения.

Б.4.4. Итераторы

Контейнер можно интерпретировать как последовательность, порядок следования элементов в которой либо определен итератором контейнера, либо является обратным к нему. Для ассоциативного контейнера порядок определяется критерием сравнения (по умолчанию оператором <).

Итераторы	
`p=c.begin()`	`p` указывает на первый элемент контейнера `c`
`p=c.end()`	`p` указывает на элемент, следующий за последним элементом контейнера `c`
`p=c.rbegin()`	`p` указывает на первый элемент последовательности `c`, записанной в обратном порядке
`p=c.rend()`	`p` указывает на элемент, следующий за последним элементом последовательности `c`, записанной в обратном порядке

Б.4.5. Доступ к элементам

К некоторым элементам можно обратиться непосредственно.

Доступ к элементам	
`c.front()`	Ссылка на первый элемент контейнера `c`
`c.back()`	Ссылка на последний элемент контейнера `c`
`c[i]`	Ссылка на `i`-й элемент контейнера `c`; доступ без проверки (не для списка)

Доступ к элементам	
`c.at(i)`	Ссылка на `i`-й элемент контейнера `c`; доступ с проверкой (только для контейнеров `vector` и `deque`)

Некоторые реализации — особенно их тестовые версии — всегда выполняют проверку диапазонов, но рассчитывать на корректность или наличие такой проверки на разных компьютерах нельзя. Если этот вопрос важен, просмотрите документацию.

Б.4.6. Операции над стеком и очередью

Стандартные контейнеры `vector` и `deque` обеспечивают эффективные операции в конце последовательности элементов. Кроме того, контейнеры `list` и `deque` обеспечивают аналогичные операции над началом своих последовательностей.

Операции над стеком и очередью	
`c.push_back(x)`	Вставка объекта `x` в конец контейнера `c`
`c.pop_back()`	Удаление последнего элемента из контейнера `c`
`c.emplace_back(args)`	Вставка `T{args}` в конец контейнера `c`; `T` — тип значения элементов контейнера `c`
`c.push_front(x)`	Вставка объекта `x` в начало контейнера `c` (перед первым элементом) (только для контейнеров `list` и `deque`)
`c.pop_front()`	Удаление первого элемента из контейнера `c` (только для контейнеров `list` и `deque`)
`c.emplace_front(args)`	Вставка `T{args}` в начало контейнера `c` (перед первым элементом); `T` — тип значения элементов контейнера `c`

Обратите внимание на то, что функции `push_front()` и `push_back()` копируют элемент в контейнер. Это значит, что размер контейнера увеличивается (на единицу). Если копирующий конструктор элемента может генерировать исключения, то вставка может завершиться неудачно.

Операции `push_front()` и `push_back()` копируют объекты-аргументы в контейнер:

```
vector<pair<string,int>> v;
v.push_back(make_pair("Cambridge",1209));
```

Если первоначальное создание объекта с последующим копированием выглядит неудобным или потенциально неэффективным, можно создать объект непосредственно во вновь выделенном слоте последовательности:

```
v.emplace_back("Cambridge",1209);
```

"Emplace" означает "разместить на месте" или "поместить в позицию".

Заметим, что операции удаления элементов не возвращают значений. Если бы они это делали, то копирующие конструкторы, генерирующие исключения, могли бы серьезно усложнить реализацию. Для доступа к элементам стека и очереди рекомендуем использовать функции `front()` и `back()` (см. раздел Б.4.5). Мы не ставили себе задачу перечислить все множество требований; попробуйте догадаться об остальных (как правило, компиляторы сообщают пользователям об их неверных догадках) или обратитесь к более подробной документации.

Б.4.7. Операции со списком

Ниже перечислены операции со списками.

Операции со списками	
`q=c.insert(p,x)`	Вставка объекта `x` перед узлом `p`
`q=c.insert(p,n,x)`	Вставка `n` копий объекта `x` перед узлом `p`
`q=c.insert(p,first,last)`	Вставка элементов из диапазона [`first`,`last`) перед узлом `p`
`q=c.emplace(p,args)`	Вставка `T{args}` перед `p`; `T` — тип значения элементов контейнера `c`
`q=c.erase(p)`	Удаление из контейнера `c` элемента, находящегося в узле `p`
`q=c.erase(first,last)`	Удаление из контейнера `c` элементов, находящихся в диапазоне [`first`,`last`)
`c.clear()`	Удаление всех элементов из контейнера `c`

Результат `q` функции `insert()` указывает на последний вставленный элемент. Результат `q` функции `erase()` указывает на элемент, следующий за последним удаленным элементом.

Б.4.8. Размер и емкость

Размер — это количество элементов в контейнере; емкость — это количество элементов, которое контейнер может сохранить до того, как потребуется дополнительно увеличить память.

Размер и емкость	
`x=c.size()`	`x` — количество элементов в контейнере `c`
`c.empty()`	Пуст ли контейнер `c`?
`x=c.max_size()`	`x` — наибольшее возможное количество элементов в контейнере `c`
`x=c.capacity()`	`x` — память, выделенная для контейнера `c` (только для классов `vector` и `string`)
`c.reserve(n)`	Выделяет память для `n` элементов из контейнера `c` (только для классов `vector` и `string`)

Размер и емкость	
`c.resize(n)`	Изменяет размер контейнера `c`, делая его равным `n` (только для классов `vector`, `string`, `list` и `deque`)

При изменении размера или емкости элементы могут переместиться в новое место. Из этого следует, что итераторы (а также указатели и ссылки) на элементы при выполнении такой операции могут стать некорректными (т.е. относиться к старым адресам).

Б.4.9. Другие операции

Контейнеры можно копировать (см. раздел Б.4.3), сравнивать и обменивать их содержимое.

Сравнения и обмен	
`c1==c2`	Все ли соответствующие один другому элементы контейнеров `c1` и `c2` равны друг другу?
`c1!=c2`	Есть ли в контейнерах `c1` и `c2` соответствующие один другому элементы, которые не равны друг другу?
`c1<c2`	Контейнер `c1` предшествует контейнеру `c2` в лексикографическом порядке?
`c1<=c2`	Контейнер `c1` предшествует контейнеру `c2` в лексикографическом порядке или они равны?
`c1>c2`	Контейнер `c1` следует за контейнером `c2` в лексикографическом порядке?
`c1>=c2`	Контейнер `c1` следует за контейнером `c2` в лексикографическом порядке или они равны?
`swap(c1,c2)`	Контейнеры `c1` и `c2` обмениваются своими элементами
`c1.swap(c2)`	Контейнеры `c1` и `c2` обмениваются своими элементами

Если сравнение контейнеров производится с помощью некоторого оператора (например, `<`), то их элементы сравниваются с помощью эквивалентного оператора для сравнения элементов (например, `<`).

Б.4.10. Операции над ассоциативными контейнерами

Ассоциативные контейнеры обеспечивают поиск элементов на основе ключей.

Операции над ассоциативными контейнерами	
`c[k]`	Ссылка на элемент с ключом `k` (в контейнерах с уникальными ключами)
`p=c.find(k)`	Итератор `p` указывает на первый элемент с ключом `k`
`p=c.lower_bound(k)`	Итератор `p` указывает на первый элемент с ключом `k`

Операции над ассоциативными контейнерами	
`p=c.upper_bound(k)`	Итератор `p` указывает на первый элемент с ключом, большим ключа `k`
`pair(p1,p2)=c.equal_range(k)`	Диапазон [`p1`,`p2`) состоит из элементов с ключами `k`
`r=c.key_comp()`	Объект `r` — это копия объекта для сравнения ключей
`r=c.value_comp()`	Объект `r` — это копия объекта для сравнения отображенных значений

Если ключ не найден, возвращается итератор `c.end()`

Первый итератор пары, возвращенной функцией `equal_range`, равен `lower_bound`, а второй — `upper_bound`. Вы можете вывести на печать значения всех элементов, имеющих ключ `"Marian"` в контейнере `multimap<string, int>`, написав следующий код:

```
string k = "Marian";
auto pp = m.equal_range(k);
if (pp.first!=pp.second)
    cout << "Элементы со значением ' " << k << " ':\n";
else
    cout << "Элементов со значением ' " << k << " ' нет\n";
for (auto p = pp.first; p!=pp.second; ++p)
    cout << p->second << '\n';
```

В качестве альтернативы можно также воспользоваться инструкцией

```
auto pp = make_pair(m.lower_bound(k),m.upper_bound(k));
```

Однако такая инструкция выполняется вдвое дольше. Алгоритмы `equal_range`, `lower_bound` и `upper_bound` предоставляются также для упорядоченных последовательностей (раздел Б.5.4). Определение класса `pair` приведено в разделе Б.6.3.

Б.5. Алгоритмы

В заголовочном файле `<algorithm>` определено около 60 алгоритмов. Все они относятся к последовательностям, определенным парами итераторов (для ввода) или одним итератором (для вывода).

При копировании, сравнении и выполнении других операций над двумя последовательностями первая из них задается парой итераторов [`b`,`e`), а вторая — только одним итератором `b2`, который считается началом последовательности, содержащей элементы, количество которых достаточно для выполнения алгоритма; например, их может быть столько же, сколько элементов в первой последовательности: [`b2`,`b2+(e-b)`).

Некоторые алгоритмы, такие как `sort`, используют итераторы произвольного доступа, а многие другие, такие как `find`, поочередно считывают элементы с помощью однонаправленного итератора.

Многие алгоритмы придерживаются обычного соглашения и возвращают конец последовательности в качестве признака события "не найден". Мы больше не будем упоминать об этом каждый раз, описывая очередной алгоритм.

Б.5.1. Немодицифирующие алгоритмы для последовательностей

Немодифицирующий алгоритм просто считывает элементы последовательности; он не изменяет порядок следования элементов последовательности и не изменяет их значения.

Алгоритмы, не модифицирующие последовательности	
`f=for_each(b,e,f)`	Применяет функциональный объект `f` к каждому элементу из диапазона [`b`,`e`); возвращает `f`
`p=find(b,e,v)`	Итератор `p` указывает на первое вхождение элемента `v` в диапазон [`b`,`e`)
`p=find_if(b,e,f)`	Итератор `p` указывает на первый элемент из диапазона [`b`,`e`), удовлетворяющий условию `f(*p)`
`p=find_first_of(b,e,b2,e2)`	Итератор `p` указывает на первый элемент из диапазона [`b`,`e`), удовлетворяющий условию `*p==*q` для некоторого элемента `q` из диапазона [`b2`,`e2`)
`p=find_first_of(b,e,b2,e2,f)`	Итератор `p` указывает на первый элемент из диапазона [`b`,`e`), удовлетворяющий условию `f(*p,*q)` для некоторого элемента `q` из диапазона [`b2`,`e2`)
`p=adjacent_find(b,e)`	Итератор `p` указывает на первый элемент из диапазона [`b`,`e`), удовлетворяющий условию `*p==*(p+1)`
`p=adjacent_find(b,e,f)`	Итератор `p` указывает на первый элемент из диапазона [`b`,`e`), удовлетворяющий условию `f(*p,*(p+1))`
`equal(b,e,b2)`	Равны ли поэлементно диапазоны [`b`,`e`) и [`b2`,`b2+(e-b)`)?
`equal(b,e,b2,f)`	Равны ли поэлементно диапазоны [`b`,`e`) и [`b2`,`b2+(e-b)`) при условии, что для проверки используется `f(*p,*q)`?
`pair(p1,p2)=mismatch(b,e,b2)`	Пара итераторов `(p1,p2)` указывает первую пару элементов из диапазонов [`b`,`e`) и [`b2`,`b2+(e-b)`), для которых выполняется условие `!(*p1==*p2)`
`pair(p1,p2)=mismatch(b,e,b2,f)`	Пара итераторов `(p1,p2)` указывает первую пару элементов из диапазонов [`b`,`e`) и [`b2`,`b2+(e-b)`), для которых выполняется условие `!f(*p1,*p2)`
`p=search(b,e,b2,e2)`	Поиск "подстроки": итератор `p` указывает на первый элемент `*p` из диапазона [`b`,`e`), начиная с которого в первом диапазоне располагаются элементы диапазона [`b2`,`e2`)

Алгоритмы, не модифицирующие последовательности	
`p=search(b.e,b2,e2,f)`	Поиск "подстроки": итератор `p` указывает на первый элемент `*p` из диапазона [`b`,`e`), начиная с которого в первом диапазоне располагаются элементы, для которых выполняется `f(*p,*q)` для соответствующих элементов из обоих диапазонов
`p=find_end(b,e,b2,e2)`	Итератор `p` указывает на последний элемент `*p` в диапазоне [`b`,`e`), равный какому-то элементу из диапазона [`b2`,`e2`)
`p=find_end(b,e,b2,e2,f)`	Итератор `p` указывает на последний элемент `*p` в диапазоне [`b`,`e`), для которого выполняется условие `f(*p,*q)`, где `*q` — некоторый элемент из диапазона [`b2`,`e2`)
`p=search_n(b,e,n,v)`	Итератор `p` указывает на первый элемент `*p` в диапазоне [`b`,`e`), такой, что каждый элемент в диапазоне [`p`,`p+n`) имеет значение `v`
`p=search_n(b,e,n,v,f)`	Итератор `p` указывает на первый элемент `*p` в диапазоне [`b`,`e`), такой, что для каждого элемента `*q` из диапазона [`p`,`p+n`) выполняется условие `f(*q,v)`
`x=count(b,e,v)`	`x` — количество вхождений значения `v` в диапазон [`b`,`e`)
`x=count_if(b,e,v,f)`	`x` — количество элементов в диапазоне [`b`,`e`), удовлетворяющих условию `f(*p,v)`

Предотвратить модификацию элементов операцией, передаваемой алгоритму `for_each`, невозможно; такая модификация считается приемлемой. Передача операции, изменяющей проверяемые ею элементы, другим алгоритмам (например, `count` или `==`) недопустима.

Рассмотрим пример правильного использования алгоритма.

```
bool odd(int x) { return x&1; }
int n_even(const vector<int>& v) // Подсчитывает количество
                                 // четных значений в v
{
    return v.size() - count_if(v.begin(),v.end(),odd);
}
```

Б.5.2. Алгоритмы, модифицирующие последовательности

Модифицирующие алгоритмы могут изменять элементы последовательностей, являющихся их аргументами.

Алгоритмы, модифицирующие последовательности	
`p=transform(b,e,out,f)`	Применяет функциональный объект `*p2=f(*p1)` к каждому элементу `*p1` из диапазона [`b`,`e`), записывая соответствующее значение `*p2` в диапазон [`out`,`out+(e-b)`); `p=out+(e-b)`

Алгоритмы, модифицирующие последовательности	
`p=transform(b,e,b2,out,f)`	Применяет функцию `*p3=f(*p1,*p2)` к каждому элементу `*p1` из диапазона [`b`,`e`) и соответствующему элементу `*p2` из [`b2`,`b2+(e-b)`), записывая значение `*p3` в диапазон [`out`,`out+(e-b)`); `p=out+(e-b)`
`p=copy(b,e,out)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`)
`p=copy_backward(b,e,out)`	Копирует диапазон [`b`,`e`) в диапазон [`p`,`out`), начиная со своего последнего элемента
`p=unique(b,e)`	Перемещает элементы в диапазоне [`b`,`e`) так, чтобы в диапазоне [`b`,`p`) не было смежных дубликатов (дубликаты определяются с помощью оператора ==)
`p=unique(b,e,f)`	Перемещает элементы в диапазоне [`b`,`e`) так, чтобы в диапазоне [`b`,`p`) не было смежных дубликатов (дубликаты определяются функцией `f`)
`p=unique_copy(b,e,out)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`), удаляя смежные дубликаты
`p=unique_copy(b,e,out,f)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`), удаляя смежные дубликаты (дубликаты определяются функцией `f`)
`replace(b,e,v,v2)`	Заменяет элементы `*q` из диапазона [`b`,`e`), для которых выполняется равенство `*q==v`, значением `v2`
`replace(b,e,f,v2)`	Заменяет элементы `*q` из диапазона [`b`,`e`), для которых выполняется условие `f(*q)`, значением `v2`
`p=replace_copy(b,e,out,v,v2)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`), заменяя элементы `*q` из диапазона [`b`,`e`), для которых выполняется условие `*q==v`, значением `v2`
`p=replace_copy(b,e,out,f,v2)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`), заменяя элементы `*q` из диапазона [`b`,`e`), для которых выполняется условие `f(*q)`, значением `v2`
`p=remove(b,e,v)`	Перемещает элементы `*q` в диапазоне [`b`,`e`) так, чтобы диапазон [`b`,`p`) содержал элементы, для которых выполняется условие `!(*q==v)`
`p=remove(b,e,v,f)`	Перемещает элементы `*q` в диапазоне [`b`,`e`) так, чтобы диапазон [`b`,`p`) содержал элементы, для которых выполняется условие `!f(*q)`
`p=remove_copy(b,e,out,v)`	Копирует элементы из диапазона [`b`,`e`), для которых выполняется условие `!(*q==v)`, в диапазон [`out`,`p`)
`p=remove_copy_if(b,e,out,f)`	Копирует элементы из диапазона [`b`,`e`), для которых выполняется условие `!f(*q,v)`, в диапазон [`out`,`p`)
`reverse(b,e)`	Меняет порядок элементов в диапазоне [`b`,`e`) на обратный
`p=reverse_copy(b,e,out)`	Копирует диапазон [`b`,`e`) в диапазон [`out`,`p`) в обратном порядке

Алгоритмы, модифицирующие последовательности	
`rotate(b,m,e)`	Выполняет циклическую перестановку элементов: диапазон [`b`,`e`) интерпретируется как окружность, в которой первый элемент следует сразу за последним. Перемещает элемент `*b` на место элемента `*m` и вообще перемещает элемент `*(b+i)` на место элемента `*((b+(i+(e-m))%(e-b))`
`p=rotate_copy(b,m,e,out)`	Копирует диапазон [`b`,`e`) в последовательность [`out`,`p`) с применением циклической перестановки
`random_shuffle(b,e)`	Перетасовывает элементы диапазона [`b`,`e`) с помощью генератора равномерно распределенных случайных чисел
`random_shuffle(b,e,f)`	Перетасовывает элементы диапазона [`b`,`e`) с `f` в качестве генератора случайных чисел

Алгоритм перетасовки изменяет последовательность точно так же, как перетасовывается колода карт; иначе говоря, после перетасовки элементы следуют в случайном порядке, причем смысл слова "случайно" определяется распределением, порожденным датчиком случайных чисел.

Следует подчеркнуть, что эти алгоритмы не знают, являются ли их аргументы контейнерами, поэтому не могут добавлять или удалять элементы. Таким образом, такой алгоритм, как `remove`, не может уменьшить длину входной последовательности, удалив (стерев) ее элементы; вместо этого он передвигает эти элементы к началу последовательности.

```
template<typename Iter>
void print_digits(const string& s, Iter b, Iter e)
{
    cout << s;
    while (b!=e) { cout << *b; ++b; )
    cout << '\n';
}

void ff()
{
    vector<int> v {1,1,1, 2,2, 3, 4,4,4, 3,3,3, 5,5,5,5, 1,1,1};
    print_digits("all:  ",v.begin(), v.end());

    auto pp = unique(v.begin(),v.end());
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());

    pp=remove(v.begin(),pp,4);
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());
}
```

Вывод при этом имеет следующий вид.

```
all:  1112234443335555111
head: 1234351
tail: 443335555111
head: 123351
tail: 1443335555111
```

Б.5.3. Вспомогательные алгоритмы

Технически вспомогательные алгоритмы также могут модифицировать последовательности, но мы считаем, что лучше их перечислить отдельно, чтобы они не затерялись в длинном списке.

Вспомогательные алгоритмы	
`swap(x,y)`	Меняет местами `x` и `y`
`iter_swap(p,q)`	Меняет местами `*p` и `*q`
`swap_ranges(b,e,b2)`	Обменивает элементы диапазонов [`b`,`e`) и [`b2`,`b2+(e-b)`)
`fill(b,e,v)`	Присваивает значение `v` каждому элементу диапазона [`b`,`e`)
`fill_n(b,n,v)`	Присваивает значение `v` каждому элементу диапазона [`b`,`b+n`)
`generate(b,e,f)`	Присваивает значение `f()` каждому элементу диапазона [`b`,`e`)
`generate_n(b,n,f)`	Присваивает значение `f()` каждому элементу диапазона [`b`,`b+n`)
`uninitialized_fill(b,e,v)`	Инициализирует все элементы из диапазона [`b`,`e`) значением `v`
`uninitialized_copy(b,e,out)`	Инициализирует каждый элемент из диапазона [`out`,`out+(e-b)`) соответствующим элементом из диапазона [`b`,`e`)

Обратите внимание на то, что неинициализированные последовательности должны использоваться только на самых нижних уровнях программирования, как правило, в реализации контейнеров. Целевые элементы алгоритмов `uninitialized_fill` и `uninitialized_copy` должны иметь встроенный тип или быть неинициализированными.

Б.5.4. Сортировка и поиск

Сортировка и поиск относятся к категории фундаментальных алгоритмов. Потребности программистов при этом оказываются довольно разнообразными. Сравнение по умолчанию выполняется с помощью оператора `<`, а эквивалентность пар значений `a` и `b` определяется условием `!(a<b)&&!(b<a)`, а не оператором `==`.

Сортировка и поиск	
`sort(b,e)`	Упорядочивает диапазон [`b`,`e`)
`sort(b,e,f)`	Упорядочивает диапазон [`b`,`e`), используя в качестве критерия сортировки функциональный объект `f(*p,*q)`

Сортировка и поиск	
`stable_sort(b,e)`	Упорядочивает диапазон [`b`,`e`), сохраняя порядок эквивалентных элементов
`stable_sort(b,e,f)`	Упорядочивает диапазон [`b`,`e`), используя в качестве критерия сортировки функциональный объект `f(*p,*q)` и сохраняя порядок эквивалентных элементов
`partial_sort(b,m,e)`	Упорядочивает элементы диапазона [`b`,`e`) в поддиапазоне [`b`,`m`); поддиапазон [`m`,`e`) может оставаться неупорядоченным
`partial_sort(b,m,e,f)`	Упорядочивает элементы диапазона [`b`,`e`) в поддиапазоне [`b`,`m`), используя в качестве критерия сортировки функциональный объект `f(*p,*q)`; поддиапазон [`m`,`e`) может оставаться неупорядоченным
`partial_sort_copy(b,e,b2,e2)`	Упорядочивает достаточное количество элементов диапазона [`b`,`e`) для копирования `e2-b2` первых элементов в диапазон [`b2`,`e2`)
`partial_sort_copy(b,e,b2,e2,f)`	Упорядочивает достаточное количество элементов диапазона [`b`,`e`) для копирования `e2-b2` первых элементов в диапазон [`b2`,`e2`); в качестве критерия сортировки используется функциональный объект `f`
`nth_element(b,p,e)`	Помещает `n`-й (`n==p-b`) элемент диапазона [`b`,`e`) в правильное местоположение
`nth_element(b,p,e,f)`	Вставляет `n`-й (`n==p-b`) элемент диапазона [`b`,`e`) в соответствующее место, используя в качестве критерия сортировки функциональный объект `f`
`p=lower_bound(b,e,v)`	Итератор `p` указывает на первое вхождение значения `v` в диапазон [`b`,`e`)
`p=lower_bound(b,e,v,f)`	Итератор `p` указывает на первое вхождение значения `v` в диапазон [`b`,`e`); в качестве критерия сравнения используется функциональный объект `f`
`p=upper_bound(b,e,v)`	Итератор `p` указывает на первое вхождение значения, превышающего значение `v` в диапазон [`b`,`e`)
`p=upper_bound(b,e,v,f)`	Итератор `p` указывает на первое вхождение значения, превышающего значение `v` в диапазон [`b`,`e`); в качестве критерия сравнения используется функциональный объект `f`
`binary_search(b,e,v)`	Есть ли значение `v` в упорядоченной последовательности [`b`,`e`)?
`binary_search(b,e,v,f)`	Есть ли значение `v` в упорядоченной последовательности [`b`,`e`)? В качестве критерия сравнения используется функциональный объект `f`
`pair(p1,p2)=equal_range(b,e,v)`	[`p1`,`p2`) — это подпоследовательность диапазона [`b`,`e`), содержащая значение `v`; по сути, алгоритм выполняет бинарный поиск значения `v`

Сортировка и поиск	
`pair(p1,p2)=equal_range(b,e,v,f)`	[p1,p2) — это подпоследовательность диапазона [b,e), содержащая значение v; по сути, алгоритм выполняет бинарный поиск значения v, используя в качестве критерия сравнения функциональный объект f
`p=merge(b,e,b2,e2,out)`	Объединяет две упорядоченные последовательности [b2,e2) и [b,e) в последовательность [out,p)
`p=merge(b,e,b2,e2,out,f)`	Объединяет две упорядоченные последовательности [b2,e2) и [b,e) в последовательность [out,p), используя в качестве критерия сравнения функциональный объект f
`inplace_merge(b,m,e)`	Объединяет две упорядоченные подпоследовательности [b,m) и [m,e) в упорядоченную последовательность [b,e)
`inplace_merge(b,m,e,f)`	Объединяет две упорядоченные подпоследовательности [b,m) и [m,e) в упорядоченную последовательность [b,e), используя в качестве критерия сравнения функциональный объект f
`p=partition(b,e,f)`	Помещает элементы, для которых выполняется условие `f(*p1)`, в диапазон [b,p), а остальные — в диапазон [p,e)
`p=stable_partition(b,e,f)`	Помещает элементы, для которых выполняется условие `f(*p1)`, в диапазон [b,p), а остальные — в диапазон [p,e), сохраняя относительный порядок элементов

Рассмотрим следующий пример:

```
vector<int> v {3,1,4,2};
list<double> lst {0.5,1.5,3,2.5}; // lst упорядочен
sort(v.begin(),v.end());          // Упорядочение v
vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),
      lst.end(),back_inserter(v2));
for (auto x : v2) cout << x << ", ";
```

Итераторы вставки описаны в разделе Б.6.1. В итоге получается следующий результат:

```
0.5, 1, 1.5, 2, 2, 2.5, 3, 4,
```

Алгоритмы `equal_range`, `lower_bound` и `upper_bound` используются точно так же, как и их эквиваленты для ассоциативных контейнеров (раздел Б.4.10).

Б.5.5. Алгоритмы для множеств

Эти алгоритмы интерпретируют последовательность как множество элементов и выполняют основные операции над множествами. Входные и выходные последовательности предполагаются упорядоченными.

Алгоритмы для множеств	
`includes(b,e,b2,e2)`	Все ли элементы диапазона `[b2,e2)` имеются в диапазоне `[b:e)`?
`includes(b,e,b2,e2,f)`	Все ли элементы диапазона `[b2,e2)` одновременно принадлежат диапазону `[b,e)`, если в качестве критерия сравнения используется функциональный объект `f`?
`p=set_union (b,e,b2,e2,out)`	Создает упорядоченную последовательность `[out,p)`, состоящую из элементов, принадлежащих хотя бы одному из диапазонов `[b,e)` и `[b2,e2)`
`p=set_union (b,e,b2,e2,out,f)`	Создает упорядоченную последовательность `[out,p)`, состоящую из элементов, принадлежащих хотя бы одному из диапазонов `[b,e)` и `[b2,e2)`, причем в качестве критерия сравнения используется функциональный объект `f`
`p=set_intersection (b,e,b2,e2,out)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих как диапазону `[b,e)`, так и диапазону `[b2,e2)`
`p=set_intersection (b,e,b2,e2,out,f)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих как диапазону `[b,e)`, так и диапазону `[b2,e2)`, причем в качестве критерия сравнения используется функциональный объект `f`
`p=set_difference (b,e,b2,e2,out)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих диапазону `[b,e)`, но не диапазону `[b2,e2)`
`p=set_difference (b,e,b2,e2,out,f)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих диапазону `[b,e)`, но не диапазону `[b2,e2)`, причем в качестве критерия сравнения используется функциональный объект `f`
`p=set_symmetric_ difference (b,e,b2,e2,out)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих либо диапазону `[b,e)`, либо диапазону `[b2,e2)`, но не обоим одновременно
`p=set_symmetric_ difference(b,e,b2, e2,out,f)`	Создает упорядоченную последовательность `[out,p)` элементов, принадлежащих либо диапазону `[b,e)`, либо диапазону `[b2,e2)`, но не обоим одновременно, причем в качестве критерия сравнения используется функциональный объект `f`

Б.5.6. Пирамиды

Пирамида (heap) — это структура данных, первым элементом которой является элемент с наибольшим значением. Алгоритмы для работы с пирамидами позволяют программистам рассматривать последовательность с произвольным доступом как пирамиды.

Операции над пирамидами	
`make_heap(b,e)`	Создает последовательность, готовую к использованию в качестве пирамиды
`make_heap(b,e,f)`	Создает последовательность, готовую к использованию в качестве пирамиды, используя в качестве критерия сравнения функциональный объект `f`
`push_heap(b,e)`	Добавляет элемент (в соответствующее место) пирамиды
`push_heap(b,e,f)`	Добавляет элемент (в соответствующее место) пирамиды, используя в качестве критерия сравнения функциональный объект `f`
`pop_heap(b,e)`	Удаляет из пирамиды наибольший (первый) элемент
`pop_heap(b,e,f)`	Удаляет из пирамиды наибольший (первый) элемент, используя в качестве критерия сравнения функциональный объект `f`
`sort_heap(b,e)`	Упорядочивает пирамиду
`sort_heap(b,e,f)`	Упорядочивает пирамиду, используя в качестве критерия сравнения функциональный объект `f`

Пирамида позволяет быстро добавлять элементы и обеспечивает быстрый доступ к элементу с наибольшим значением. В основном кучи используются при реализации очередей с приоритетами.

Б.5.7. Перестановки

Перестановки используются для генерации комбинаций элементов последовательности. Например, перестановками последовательности `abc` являются последовательности `abc`, `acb`, `bac`, `bca`, `cab` и `cba`.

Перестановки	
`x=next_permutation(b,e)`	Создает следующую в лексикографическом порядке перестановку последовательности [`b`,`e`)
`x=next_permutation(b,e,f)`	Создает следующую в лексикографическом порядке перестановку последовательности [`b`,`e`), используя в качестве критерия сравнения функциональный объект `f`
`x=prev_permutation(b,e)`	Создает предыдущую в лексикографическом порядке перестановку последовательности [`b`,`e`)
`x=prev_permutation(b,e,f)`	Создает предыдущую в лексикографическом порядке перестановку последовательности [`b`,`e`), используя в качестве критерия сравнения функциональный объект `f`

Если последовательность [`b`,`e`) уже содержит последнюю перестановку (в приведенном выше примере это перестановка `cba`), то алгоритм `next_permutation` возвращает значение `false`; в таком случае алгоритм создает первую перестановку (в данном примере это перестановка `abc`). Если последовательность [`b`,`e`) уже содержит первую перестановку (в данном

примере это перестановка `abc`), то алгоритм `prev_permutation` возвращает значение `false`; в таком случае алгоритм создает последнюю перестановку (в данном примере это перестановка `cba`).

Б.5.8. Функции `min` и `max`

Сравнение значений полезно в различных контекстах.

`min` и `max`	
`x=max(a,b)`	`x` — большее из значений `a` и `b`
`x=max(a,b,f)`	`x` — большее из значений `a` и `b`; в качестве критерия сравнения используется функциональный объект `f`
`x=max({elems})`	`x` — наибольшее значение среди `{elems}`
`x=max({elems},f)`	`x` — наибольшее значение среди `{elems}`; в качестве критерия сравнения используется функциональный объект `f`
`x=min(a,b)`	`x` — меньшее из значений `a` и `b`
`x=min(a,b,f)`	`x` — меньшее из значений `a` и `b`; в качестве критерия сравнения используется функциональный объект `f`
`x=min({elems})`	`x` — наименьшее значение среди `{elems}`
`x=min({elems},f)`	`x` — наименьшее значение среди `{elems}`; в качестве критерия сравнения используется функциональный объект `f`
`pair(x,y)=minmax(a,b)`	`x` равно `min(a,b)` и `y` равно `max(a,b)`
`pair(x,y)=minmax(a,b,f)`	`x` равно `min(a,b,f)` и `y` равно `max(a,b,f)`
`pair(x,y)=minmax({elems})`	`x` равно `min({elems})` и `y` равно `max({elems})`
`pair(x,y)=minmax({elems},f)`	`x` равно `min({elems},f)` и `y` равно `max({elems},f)`
`p=max_element(b,e)`	Итератор `p` указывает на наибольший элемент из диапазона [`b`,`e`)
`p=max_element(b,e,f)`	Итератор `p` указывает на наибольший элемент из диапазона [`b`,`e`); в качестве критерия сравнения используется функциональный объект `f`
`p=min_element(b,e)`	Итератор `p` указывает на наименьший элемент из диапазона [`b`,`e`)
`p=minelement(b,e,f)`	Итератор `p` указывает на наименьший элемент из диапазона [`b`,`e`); в качестве критерия сравнения используется функциональный объект `f`
`lexicographical_compare(b,e,b2,e2)`	Выполнятся ли условие [`b`,`e`)<[`b2`,`e2`)?
`lexicographical_compare(b,e,b2,e2,f)`	Выполняется ли условие [`b`,`e`)<[`b2`,`e2`), если в качестве критерия сравнения используется функциональный объект `f`?

Б.6. Утилиты библиотеки STL

В стандартной библиотеке есть несколько инструментов для облегчения использования алгоритмов стандартной библиотеки.

Б.6.1. Итераторы вставки

Запись результатов в контейнер с помощью итератора подразумевает, что элементы, на которые указывает итератор, можно перезаписать. Это открывает возможность для переполнения, а значит, повреждения памяти. Рассмотрим следующий пример:

```
void f(vector<int>& vi)
{
   fill_n(vi.begin(),200,7); // Присваиваем значение 7 элементам vi[0]..[199]
}
```

Если вектор `vi` содержит меньше 200 элементов, у нас проблемы.

В заголовочном файле `<iterator>` стандартной библиотеки предусматриваются три итератора, позволяющих решить эту проблему с помощью добавления (вставки) элементов в контейнер, а не перезаписи его старых элементов. Для создания этих трех итераторов вставки используются три функции.

Алгоритмы вставки	
`r=back_inserter(c)`	`*r=x` выполняет вызов `c.push_back(x)`
`r=front_inserter(c)`	`*r=x` выполняет вызов `c.push_front(x)`
`r=inserter(c,p)`	`*r=x` выполняет вызов `c.insert(p,x)`

Для правильной работы `inserter(c,p)` необходимо, чтобы итератор `p` был корректным итератором контейнера `c`. Естественно, каждый раз при записи очередного элемента с помощью итератора вставки контейнер увеличивается на один элемент. При записи итератор вставки добавляет новый элемент в последовательность с помощью функций `push_back(x)`, `push_front()` или `insert()`, а не перезаписывает существующий элемент:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7); // Добавляем 200 элементов,
                                     // равных 7, в конец vi
}
```

Б.6.2. Функциональные объекты

Многие стандартные алгоритмы принимают в качестве аргументов функциональные объекты (или функции), чтобы уточнить способ решения задачи. Обычно эти функциональные объекты используются в качестве

критериев сравнения, предикатов (функций, возвращающих значения типа `bool`) и арифметических операций. Несколько самых распространенных функциональных объектов описано в заголовочном файле `<functional>` стандартной библиотеки.

Предикаты	
`p=equal_to<T>{}`	`p(x,y)` означает `x==y`, если `x` и `y` имеют тип `T`
`p=not_equal_to<T>{}`	`p(x,y)` означает `x!=y`, если `x` и `y` имеют тип `T`
`p=greater<T>{}`	`p(x,y)` означает `x>y`, если `x` и `y` имеют тип `T`
`p=less<T>{}`	`p(x,y)` означает `x<y`, если `x` и `y` имеют тип `T`
`p=greater_equal<T>{}`	`p(x,y)` означает `x>=y`, если `x` и `y` имеют тип `T`
`p=less_equal<T>{}`	`p(x,y)` означает `x<=y`, если `x` и `y` имеют тип `T`
`p=logical_and<T>{}`	`p(x,y)` означает `x&&y`, если `x` и `y` имеют тип `T`
`p=logical_or<T>{}`	`p(x,y)` означает `x||y`, если `x` и `y` имеют тип `T`
`p=logical_not<T>{}`	`p(x)` означает `!x`, если `x` имеет тип `T`

Рассмотрим следующий пример:

```
vector<int> v;
// ...
sort(v.begin(),v.end(),greater<int>{}); // Сортировка v в
                                        // убывающем порядке
```

Обратите внимание на то, что предикаты `logical_and` и `logical_or` всегда вычисляют оба свои аргумента (в то время как операторы `&&` и `||` — нет).

Арифметические операции	
`f=plus<T>{}`	`f(x,y)` означает `x+y`, где `x` и `y` имеют тип `T`
`f=minus<T>{}`	`f(x,y)` означает `x-y`, где `x` и `y` имеют тип `T`
`f=multiplies<T>{}`	`f(x,y)` означает `x*y`, где `x` и `y` имеют тип `T`
`f=divides<T>{}`	`f(x,y)` означает `x/y`, где `x` и `y` имеют тип `T`
`f=modulus<T>{}`	`f(x,y)` означает `x%y`, где `x` и `y` имеют тип `T`
`f=negate<T>{}`	`f(x)` означает `-x`, где `x` имеет тип `T`
Адаптеры	
`f=bind(g,args)`	`f(x)` означает `g(x,args)`, где `args` — один или несколько аргументов
`f=mem_fun(mf)`	`f(p,args)` означает `p->mf(args)`, где `args` — один или несколько аргументов
`function<F>f{g}`	`f(args)` означает `g(args)`, где `args` — один или несколько аргументов, а `F` представляет собой тип `g`
`f=not1(g)`	`f(x)` означает `!g(x)`
`f=not2(g)`	`f(x,y)` означает `!g(x,y)`

Обратите внимание, что **function** представляет собой шаблон, так что вы можете определить переменные типа **function<T>** и присваивать этим переменным вызывающие объекты, например:

```
int f1(double);
function<int(double)> fct {f1}; // Инициализация функцией f1
int x = fct(2.3);               // Вызов f1(2.3)
function<int(double)> fun;      // fun может хранить любую
fun = f1;                       // функцию int(double)
```

Б.6.3. Классы pair и tuple

В заголовочном файле **<utility>** стандартной библиотеки содержится несколько вспомогательных компонентов, включая класс **pair**.

```
template <typename T1, typename T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;

        // Операции копирования и перемещения
    };

template <typename T1, typename T2>
constexpr pair<T1,T2> make_pair(T1 x, T2 y) {
    return pair<T1,T2>{x,y};
}
```

Функция **make_pair()** упрощает использование пар. Например, рассмотрим схему функции, возвращающей значение и индикатор ошибки.

```
pair<double,error_indicator> my_fct(double d)
{
    errno = 0; // Сброс признака ошибок в стиле языка C
    // ... выполняем много вычислений, связанных
    // с переменной d, и вычисляем x ...
    error_indicator ee = errno;
    errno = 0; // Сброс признака ошибок в стиле языка C
    return make_pair(x,ee);
}
```

Этот пример является полезной идиомой, которую можно использовать следующим образом:

```
pair<int,error_indicator> res = my_fct(123.456);
if (res.second==0) {
    // ... используем res.first ...
}
else {
    // Ошибка
}
```

Мы используем **pair**, когда нам требуются ровно два элемента и не хочется определять отдельный тип. Если же нам нужны нуль или большее количество элементов, мы можем воспользоваться шаблоном **tuple** из заголовочного класса **<tuple>**:

```
template <typename... Types>
struct tuple {
    // Создание из N значений:
    explicit constexpr tuple(const Types&...);

    // Создание из N значений:
    template<typename... Atypes>
    explicit constexpr tuple(const Atypes&&...);

    // ... операции копирования и перемещения ...
};

// Создание кортежа из N элементов:
template <class... Types>
constexpr tuple<Types...> make_tuple(Types&&...);
```

Реализация **tuple** использует возможность языка, выходящую за рамки данной книги, — шаблоны с переменным числом аргументов (на что указывает троеточие ...). Однако использовать **tuple** можно практически так же, как и **pair**, например:

```
auto t0 = make_tuple();              // Элементов нет
auto t1 = make_tuple(123.456);       // Один элемент типа double
auto t2 = make_tuple(123.456, 'a'); // Два элемента типа double
                                     // и char
auto t3 = make_tuple(12,'a',         // Три элемента с типами
                  string{"How?"});   // int, char и string
```

Кортеж **tuple** может иметь много элементов, так что мы не можем просто использовать **first** и **second** для доступа к ним. Вместо этого используется функция **get**:

```
auto d = get<0>(t1); // double
auto n = get<0>(t3); // int
auto c = get<1>(t3); // char
auto s = get<2>(t3); // string
```

Индекс передается функции **get** как шаблонный аргумент. Как видно из приведенного примера, индексы **tuple** начинаются с нуля.

Кортежи в основном используются в обобщенном программировании.

Б.6.4. Список инициализации

В заголовочном файле **<initializer_list>** содержится определение **initializer_list**:

```
template<typename T>
class initializer_list {
public:
    initializer_list() noexcept;
    size_t size() const noexcept;     // Количество элементов
    const T* begin() const noexcept; // Первый элемент
    const T* end() const noexcept;   // Элемент после последнего
    // ...
};
```

Когда компилятор встречает список инициализации `{}` с элементами типа `X`, этот список используется для создания `initializer_list<X>` (разделы 14.2.1, 18.2). К сожалению, `initializer_list` не поддерживает оператор индексации (`[]`).

Б.6.5. Указатели управления ресурсами

Встроенные указатели не позволяют выяснить, представляют ли они владение объектом, на который указывают. Это может серьезно упростить программирование (§19.5). Для решения этой проблемы предназначены указатели управления ресурсами `unique_ptr` и `shared_ptr`, определенные в заголовочном файле `<memory>`:

- `unique_ptr` (раздел 19.5.4) представляет исключительное владение; может быть только один `unique_ptr`, указывающий на объект, и этот объект удаляется при уничтожении указателя `unique_ptr`.
- `shared_ptr` представляет совместное владение; может иметься много указателей `shared_ptr`, указывающих на объект, и объект удаляется только при уничтожении последнего указателя `shared_ptr`.

unique_ptr<p> (упрощенный)	
`unique_ptr up {};`	Конструктор по умолчанию: `up` содержит `nullptr`
`unique_ptr up {p};`	`up` содержит `p`
`unique_ptr up {up2};`	Перемещающий конструктор: `up` содержит `p` из `up2`; `up2` получает `nullptr`
`up.~unique_ptr()`	Удаление указателя `up`
`p=up.get()`	`p` — указатель, содержащийся в `up`
`p=up.release()`	`p` — указатель, содержавшийся в `up`; после вызова `up` содержит `nullptr`
`up.reset(p)`	Удаление указателя, содержавшегося в `up`; после вызова `up` содержит `p`
`up=make_unique<X>(args)`	`up` содержит `new<X>(args)` (C++14)

Над указателем `unique_ptr` могут выполняться обычные операции над указателями, такие как `*`, `->`, `==` и `<`. Кроме того, `unique_ptr` может быть

определен так, что будет использовать для удаления действие, отличное от вызова обычного оператора `delete`.

shared_ptr<p> (упрощенный)	
`shared_ptr sp {};`	Конструктор по умолчанию: `sp` содержит `nullptr`
`shared_ptr sp {p};`	`sp` содержит `p`
`shared_ptr sp {sp2};`	Копирующий конструктор: и `sp`, и `sp2` содержат `p` из `sp2`
`shared_ptr sp {move(sp2)};`	Перемещающий конструктор: `sp` содержит `p` из `sp2`; `sp2` содержит `nullptr`
`sp.~shared_ptr()`	Удаление указателя, содержащегося в `sp`, если `sp` является последним `shared_ptr` для этого указателя
`sp = sp2`	Копирующее присваивание: если `sp` является последним общим указателем, содержащим данный указатель `p`, этот указатель удаляется; после присваивания и `sp`, и `sp2` содержат указатель `p` из `sp2`
`sp = move(sp2)`	Перемещающее присваивание: если `sp` является последним общим указателем, содержащим данный указатель `p`, этот указатель удаляется; после присваивания `sp` содержит указатель `p` из `sp2`, а `sp2` содержит `nullptr`
`p=sp.get()`	`p` — указатель, содержащийся в `sp`
`n=sp.use_count()`	Сколько `shared_ptr` содержат указатель, содержащийся в `sp`?
`sp.reset(p)`	Если `sp` является последним общим указателем, содержащим некоторый указатель, этот указатель удаляется; `sp` содержит `p`
`sp=make_shared<X>(args)`	`sp` содержит `new<X>(args)`

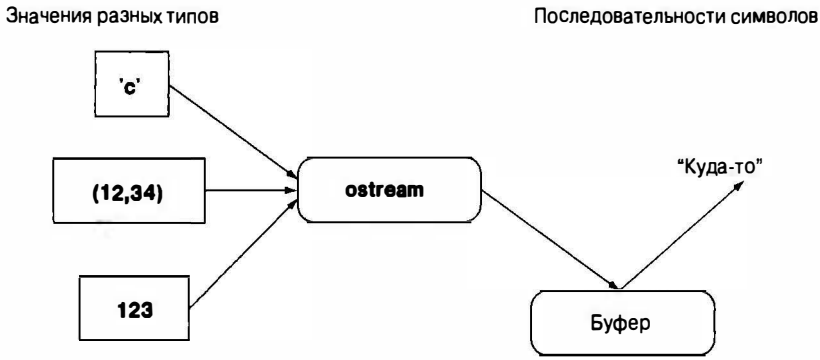
Над указателем `shared_ptr` могут выполняться обычные операции над указателями, такие как `*`, `->`, `==` и `<`. Кроме того, `shared_ptr` может быть определен так, что будет использовать для удаления действие, отличное от вызова обычного оператора `delete`.

Имеется также указатель `weak_ptr`, позволяющий разрушать циклы из указателей `shared_ptr`.

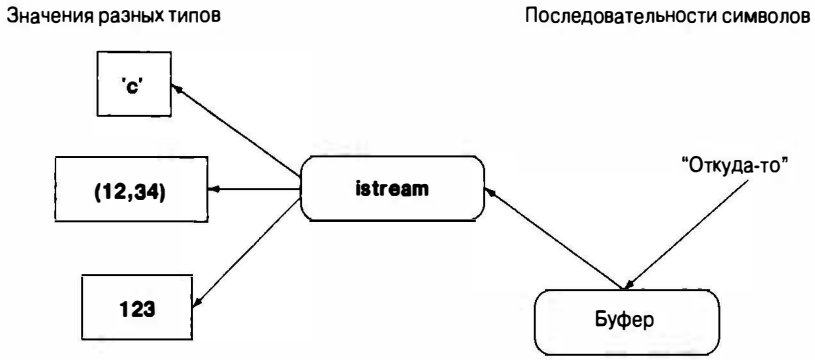
Б.7. Потоки ввода-вывода

Библиотека потоков ввода-вывода содержит средства форматированного и неформатированного буферизованного ввода-вывода текста и числовых значений. Определения потоков ввода-вывода находятся в заголовочных файлах `<istream>`, `<ostream>` и т.п. (см. раздел Б.1.1).

Объект класса `ostream` преобразовывает объекты, имеющие тип, в поток символов (байтов).



Объект класса **istream** преобразовывает поток символов (байтов) в объекты, имеющие тип.



Объект класса **iostream** — это поток, который может действовать и как объект класса **istream**, и как объект класса **ostream**. Буфера, изображенные на диаграмме, являются потоковыми (**streambuf**). Если читателям потребуется перейти от потоков класса **iostream** к новым видам устройств, файлов или памяти, они смогут найти их описание в профессиональных учебниках.

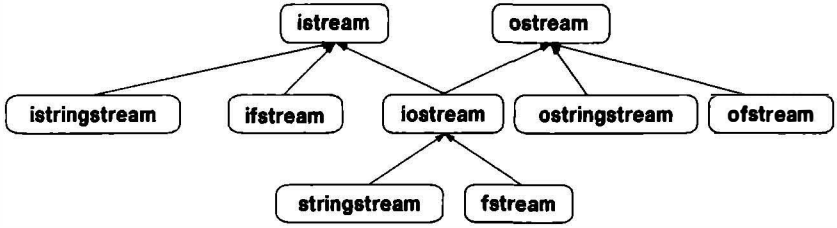
Существуют три стандартных потока.

Стандартные потоки ввода-вывода	
cout	Стандартный поток вывода (по умолчанию, как правило, экран)
cin	Стандартный поток ввода (по умолчанию, как правило, клавиатура)
cerr	Стандартный поток сообщений об ошибках (небуферизованный)

Б.7.1. Иерархия потоков ввода-вывода

Поток **istream** можно связать с устройством ввода (например, клавиатурой), файлом или объектом класса **string**. Аналогично поток **ostream** можно связать с устройством вывода (например, текстовым окном), файлом

или объектом класса `string`. Потоки ввода-вывода образуют иерархию классов.



Поток можно открыть с помощью либо конструктора, либо вызова функции `open()`.

Потоки ввода	
`stringstream(m)`	Создает пустой строковый поток с режимом `m`
`stringstream(s,m)`	Создает строковый поток, содержащий строку `string s`, с режимом `m`
`fstream()`	Создает файл, который будет открыт позднее
`fstream(s,m)`	Открывает файл `s` с режимом `m` и создает файловый поток, ссылающийся на него
`fs.open(s,m)`	Открывает файл `s` с режимом `m` и устанавливает связь между потоком `fs` и этим файлом
`fs.is_open()`	Открыт ли поток `fs`?

Для файловых потоков имя файлов представляет собой С-строку.

Открыть файл можно в одном из режимов, приведенных ниже.

Режимы потоков	
`ios_base::app`	Добавление (т.е. запись в конец файла)
`ios_base::ate`	"ate" означает "at end" (т.е. открытие и установка текущей позиции в конец файла)
`ios_base::binary`	Бинарный режим. Остерегайтесь этого режима, поскольку он зависит от специфики конкретной системы
`ios_base::in`	Для чтения
`ios_base::out`	Для записи
`ios_base::trunc`	Урезать файл до нулевой длины

В каждом из этих режимов точный результат открытия файла может зависеть от операционной системы и ее возможностей учесть требование программиста открыть файл именно так, а не иначе. В результате поток может оказаться не в состоянии `good()`. Рассмотрим пример.

```
void my_code(ostream& os); // Функция my_code может использовать
                           // любой поток вывода
ostringstream os;          // Буква "o" означает "для вывода"
```

```
ofstream of("my_file");
if (!of) error("невозможно открыть 'my_file' для записи");
my_code(os);                    // Используется объект класса string
my_code(of);                    // Используется файл
```

(См. раздел 11.3.)

Б.7.2. Обработка ошибок

Поток **iostream** может находиться в одном из четырех состояний.

Состояния потока	
good()	Операции выполнены успешно
eof()	Достигнут конец файла ("end of file")
fail()	Произошло что-то непредвиденное (например, искали цифру, а вместо нее обнаружили символ **'x'**)
bad()	Произошло что-то непредвиденное и серьезное (например, ошибка при чтении данных с диска)

Используя функцию **s.exceptions()**, программист может потребовать, чтобы поток **iostream** сгенерировал исключение, если из состояния **good()** он перешел в другое состояние (см. раздел 10.6).

Любая операция, в результате которой поток не находится в состоянии **good()**, не имеет никакого эффекта — такую операцию можно рассматривать как отсутствие операции.

Объект класса **iostream** можно использовать как условие. В данном случае условие является истинным (успешным), если поток **iostream** находится в состоянии **good()**. Это обстоятельство стало основой для распространенной идиомы, предназначенной для считывания потока значений.

```
for (X buf; cin>>buf; ) { // "Буфер ввода" для хранения
                          // одного значения типа X
    // Какие-то действия с объектом x
}
// Мы окажемся здесь, когда оператор >> не сможет прочитать
// очередной объект класса X из потока cin
```

Б.7.3. Операции ввода

Почти все операции ввода описаны в заголовочном файле **<istream>**, за исключением операций ввода в строку **string**; эти операции описаны в заголовочном файле **<string>**.

Форматированный ввод	
in >> x	Ввод из потока **in** в объект **x** в соответствии с типом **x**
getline(in,s)	Ввод строки из потока **in** в строку **s**

Если не указано иное, операция ввода возвращает ссылку на объект класса `istream`, поэтому можно создавать цепочки таких операций, например `cin>>x>>y;`.

Неформатированный ввод	
`x=in.get()`	Вводит один символ из потока `in` и возвращает его целочисленное значение
`in.get(c)`	Вводит символ из потока `in` в переменную `c`
`in.get(p,n)`	Вводит не более `n` символов из потока `in` в массив, начинающийся с адреса `p`
`in.get(p,n,t)`	Вводит не более `n` символов из потока `in` в массив, начинающийся с адреса `p`; символ `t` считается признаком конца ввода
`in.getline(p,n)`	Вводит не более `n` символов из потока `in` в массив, начинающийся с адреса `p`; удаляет признак конца ввода из потока `in`
`in.getline(p,n,t)`	Вводит не более `n` символов из потока `in` в массив, начинающийся с позиции `p`; символ `t` считается признаком конца ввода; удаляет признак конца ввода из потока `in`
`in.read(p,n)`	Вводит не более `n` символов из потока `in` в массив, начинающийся с адреса `p`
`x=in.gcount()`	`x` — это количество символов, введенных во время выполнения последней операции неформатированного ввода из потока `in`
`in.unget()`	Восстановление состояния потока, так что очередной считываемый символ — тот же, что и только что прочитанный
`in.putback(x)`	Помещает символ `x` "назад" в поток, так что очередным считываемым символом будет символ `x`

Функции `get()` и `getline()` помещают после символов (если таковые были считаны), записанных в ячейки массива `p`, число 0; функция `getline()` удаляет признак конца ввода (`t`) из потока ввода, если он обнаружен, а функция `get()` этого не делает. Функция `read(p,n)` не записывает число 0 в массив после считанных символов. Очевидно, что операторы форматированного ввода проще в использовании и менее уязвимы для ошибок, чем операции неформатированного ввода.

Б.7.4. Операции вывода

Почти все операции вывода описаны в заголовочном файле `<ostream>`, за исключением операции записи в строки `string`; такие операции описаны в заголовочном файле `<string>`.

Операции вывода	
`out << x`	Записывает объект `x` в поток `out` в соответствии с типом `x`
`out.put(c)`	Записывает символ `c` в поток `out`
`out.write(p,n)`	Записывает символы `p[0]..p[n-1]` в поток `out`

Если не указано иное, операции вставки в поток `ostream` возвращают ссылку на объект потока, поэтому можно создавать цепочки операций вывода, например `cout << x << y;`.

Б.7.5. Форматирование

Формат потока ввода-вывода управляется комбинацией типа объекта, состояния потока, информацией о локализации (см. заголовочный файл `<locale>`) и явными операциями. Большая часть информации об этом изложена в главах 10 и 11. Здесь мы просто перечислим стандартные манипуляторы (операции, модифицирующие состояние потока), поскольку они обеспечивают наиболее простой способ изменения формата.

Вопросы локализации выходят за рамки рассмотрения настоящей книги.

Б.7.6. Стандартные манипуляторы

В стандартной библиотеке предусмотрены манипуляторы, соответствующие различным состояниям формата и их изменениям. Стандартные манипуляторы определены в заголовочных файлах `<ios>`, `<istream>`, `<ostream>`, `<iostream>` и `<iomanip>` (для манипуляторов, получающих аргументы).

Манипуляторы ввода-вывода	
`s << boolalpha`	Использовать символьное представление значений `true` и `false` (ввод и вывод)
`s << noboolalpha`	`s.unsetf(ios_base::boolalpha)`
`s << showbase`	Показывать префиксы вывода `oct` в виде `0` и `hex` в виде `0x`
`s << noshowbase`	`s.unsetf(ios_base::showbase)`
`s << showpoint`	Всегда показывать десятичную точку
`s << noshowpoint`	`s.unsetf(ios_base::showpoint)`
`s << showpos`	Показывать символ + перед положительными числами
`s << noshowpos`	`s.unsetf(ios_base::showpos)`
`s >> skipws`	Пропускать пробелы
`s >> noskipws`	`s.unsetf(ios_base::skipws)`
`s << uppercase`	Использовать верхний регистр при выводе чисел, например `1.2E10` и `0X1A2`, а не `1.2e10` и `0x1a2`
`s << nouppercase`	Выводить `x` и `e`, а не `X` и `E`
`s << internal`	Вставлять пробелы в местах, указанных в шаблоне формата
`s << left`	Вставлять пробелы после значения
`s << right`	Вставлять пробелы перед значением
`s << dec`	Основание счисления равно 10
`s << hex`	Основание счисления равно 16

Манипуляторы ввода-вывода	
`s << oct`	Основание счисления равно 8
`s << fixed`	Формат чисел с плавающей точкой `dddd.dd`
`s << scientific`	Научный формат `d.ddddEdd`
`s << defaultfloat`	Формат, дающий наиболее точный вывод значений с плавающей точкой
`s << endl`	Вставить `'\n'` и сбросить буфер
`s << ends`	Вставить `'\0'`
`s << flush`	Сброс потока
`s >> ws`	Удалить разделители
`s << resetiosflags(f)`	Сбросить флаги `f`
`s << setiosflags(f)`	Установить флаги `f`
`s << setbase(b)`	Выводить целые числа по основанию `b`
`s << setfill(c)`	Сделать символ `c` символом заполнения
`s << setprecision(n)`	Точность равна `n` цифр
`s << setw(n)`	Ширина следующего поля равна `n` символам

Каждая из этих операций возвращает ссылку на свой первый операнд потока `s`. Рассмотрим пример.

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

Этот код выводит на экран следующую строку:

```
1234,4d2,2322
```

В свою очередь, код

```
cout << '(' << setw(4) << setfill('#') << 12
     << ") (" << 12 << ")\n";
```

выводит на экран строку

```
(##12) (12)
```

Для того чтобы явно установить общий формат вывода чисел с плавающей точкой, используйте следующую инструкцию:

```
b.setf(ios_base::fmtflags(0), ios_base::floatfield)
```

(См. главу 11.)

Б.8. Работа со строками

В стандартной библиотеке предусмотрены операции классификации символов (заголовочный файл `<cctype>`), строки и соответствующие операции (заголовочный файл `<string>`), регулярные выражения (заголовочный файл `<regex>`) и поддержка C-строк (заголовочный файл `<cstring>`).

Б.8.1. Классификация символов

Символы из основного набора могут быть классифицированы так, как показано ниже.

Классификация символов	
`isspace(c)`	Является ли символ `c` пробельным (`' '`, `'\t'`, `'\n'` и т.д.)?
`isalpha(c)`	Является ли символ `c` буквой (`'a'..'z'`, `'A'..'Z'`)? (Примечание: но не `'_'`.)
`isdigit(c)`	Является ли символ `c` десятичной цифрой (`'0'..'9'`)?
`isxdigit(c)`	Является ли символ `c` шестнадцатеричной цифрой (т.е. десятичной цифрой или символом `'a'..'f'` или `'A'..'F'`)?
`isupper(c)`	Является ли символ `c` буквой в верхнем регистре?
`islower(c)`	Является ли символ `c` буквой в нижнем регистре?
`isalnum(c)`	Является ли символ `c` буквой или десятичной цифрой?
`iscntrl(c)`	Является ли символ `c` управляющим символом (ASCII 0..31 и 127)?
`ispunct(c)`	Является ли символ `c` не буквой, не цифрой, не пробельным и не невидимым управляющим символом?
`isprint(c)`	Можно ли напечатать символ `c` (т.е. является ли он элементом набора ASCII от `' '` до `'~'`)?
`isgraph(c)`	Является ли символ `c` буквой, цифрой или знаком пунктуации (`isalpha()` или `isdigit()` или `ispunct()`)? (Примечание: но не пробелом.)

Кроме того, в стандартной библиотеке описаны две полезные функции для изменения регистра символа.

Верхний и нижний регистры	
`toupper(c)`	Символ `c` или его эквивалент в верхнем регистре
`tolower(c)`	Символ `c` или его эквивалент в нижнем регистре

Расширенные наборы символов, такие как Unicode, также поддерживаются стандартной библиотекой, но эта тема выходит за рамки рассмотрения настоящей книги.

Б.8.2. Строки

Класс `string` из стандартной библиотеки представляет собой специализацию общего шаблонного класса `basic_string` для символьного типа `char`; иначе говоря, объект `string` — это последовательность символов `char`.

Операции над строками	
`s=s2`	Присваивает строку `s2` строке `s`; операнд `s2` может быть объектом класса `string` или С-строкой

Операции над строками	
`s+=x`	Добавляет `x` в конец строки `s`; операнд `x` может быть символом, объектом класса `string` или С-строкой
`s[i]`	Индексирование
`s+s2`	Конкатенация; результатом является новая строка, содержащая символы строки `s`, за которыми следуют символы строки `s2`
`s==s2`	Сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s!=s2`	Сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s<s2`	Лексикографическое сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s<=s2`	Лексикографическое сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s>s2`	Лексикографическое сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s>=s2`	Лексикографическое сравнение строк; один из операндов `s` и `s2` может быть С-строкой, но не оба одновременно
`s.size()`	Количество символов в строке `s`
`s.length()`	Количество символов в строке `s`
`s.c_str()`	Представление объекта `s` в виде С-строки (завершающейся нулевым символом)
`s.begin()`	Итератор, указывающий на первый символ
`s.end()`	Итератор, указывающий на символ, следующий за последним символом строки `s`
`s.insert(pos,x)`	Вставляет `x` перед символом `s[pos]`; `x` может быть строкой `string` или С-строкой
`s.append(x)`	Вставляет `x` после последнего символа `s`; `x` может быть строкой `string` или С-строкой
`s.erase(pos)`	Удаляет завершающие символы `s`, начиная с элемента `s[pos]`. Размер строки `s` становится равным `pos`.
`s.erase(pos,n)`	Удаляет `n` символов из строки `s`, начиная с элемента `s[pos]`. Размер строки `s` становится равным `max(pos,size-n)`
`s.push_back(c)`	Добавляет символ `c` в конец строки
`pos=s.find(x)`	Находит `x` в строке `s`; `x` может быть символом, строкой `string` или С-строкой; операнд `pos` — это индекс первого найденного символа или значение `string::npos` (позиция, следующая за концом строки `s`)
`in>>s`	Вводит слово в строку `s` из потока `in`

Б.8.3. Регулярные выражения

Возможности регулярных выражений описываются в заголовочном файле **<regex>**. Основными ее функциями являются следующие.

- *Поиск* (searching) строки, соответствующей регулярному выражению, в (произвольно длинном) потоке данных; обеспечивается функцией **regex_search()**.
- *Сопоставление* (matching) регулярного выражения со строкой (известного размера); обеспечивается функцией **regex_match()**.
- *Замена соответствий* (replacement of matches); обеспечивается функцией **regex_replace()**; в данной книге не описывается, обратитесь к профессиональным учебникам или справочникам.

Результатом работы функций **regex_search()** и **regex_match()** является набор соответствий, как правило, представленный в виде объекта **smatch**.

```
regex row("^[\\w ]+( \\d+)(   \\d+)( \\d+)$"); // Строка данных

while (getline(in,line)) {                     // Проверка строки
       smatch matches;
       if (!regex_match(line, matches, row))
              error("некорректная строка", lineno);

       // Проверка строки:
       int field1 = from_string<int>(matches[1]);
       int field2 = from_string<int>(matches[2]);
       int field3 = from_string<int>(matches[3]);
       // ...
}
```

Синтаксис регулярных выражений основан на символах, имеющих особое значение (см. главу 23).

Специальные символы в регулярных выражениях	
.	Любой отдельный символ
[	Класс символов
{	Количество
(	Начало группы
)	Конец группы
\	Следующий символ имеет особое значение
*	Нуль или больше
+	Один или больше
?	Необязательный (нуль или один)
|	Альтернатива (или)
^	Начало строки; отрицание
$	Конец строки

Повторения	
`{n}`	Ровно `n` раз
`{n,}`	`n` или больше раз
`{n,m}`	Не менее `n` и не более `m` раз
`*`	Нуль или больше, т.е. `{0,}`
`+`	Один или больше, т.е. `{1,}`
`?`	Необязательный (нуль или один), т.е. `{0,1}`

Классы символов	
`alnum`	Любой алфавитно-цифровой символ или символ подчеркивания
`alpha`	Любой алфавитный символ
`blank`	Любой разделитель, не являющийся разделителем строки
`cntrl`	Любой управляющий символ
`d`	Любая десятичная цифра
`digit`	Любая десятичная цифра
`graph`	Любой графический символ
`lower`	Любой символ в нижнем регистре
`print`	Любой печатный символ
`punct`	Любой знак пунктуации
`s`	Любой пробельный символ
`space`	Любой пробельный символ
`upper`	Любой символ в верхнем регистре
`w`	Любой символ слова (алфавитно-цифровой символ)
`xdigit`	Любая шестнадцатеричная цифра

Некоторые классы символов поддерживаются аббревиатурами.

Аббревиатуры классов символов		
`\d`	Десятичная цифра	`[[:digit:]]`
`\l`	Символ в нижнем регистре	`[[:lower:]]`
`\s`	Пробельный символ (пробел, табуляция и т.п.)	`[[:space:]]`
`\u`	Символ в верхнем регистре	`[[:upper:]]`
`\w`	Буква, десятичная цифра или символ подчеркивания (_)	`[[:alnum:]]`
`\D`	Не `\d`	`[^[:digit:]]`
`\L`	Не `\l`	`[^[:lower:]]`
`\S`	Не `\s`	`[^[:space:]]`
`\U`	Не `\u`	`[^[:upper:]]`
`\W`	Не `\w`	`[^[:alnum:]]`

Б.9. Работа с числами

В стандартной библиотеке языка C++ содержатся основные строительные конструкции для математических (научных, инженерных и т.д.) вычислений.

Б.9.1. Предельные значения

Каждая реализация языка C++ определяет свойства встроенных типов, чтобы программисты могли использовать эти средства для проверки предельных значений, установки ограничителей и т.д.

В заголовочном файле **<limits>** определен шаблонный класс **numeric_limits<T>** для каждого встроенного или библиотечного типа **T**. Кроме того, программист может определить класс **numeric_limits<X>** для пользовательского числового типа **X**. Рассмотрим пример.

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;
    static constexpr int radix = 2;     // Основание системы
                // счисления (в данном случае двоичная)
    static constexpr int digits = 24;  // Количество цифр в
                // мантиссе в текущей системе счисления
    static constexpr int digits10 = 6; // Количество
                // десятичных цифр в мантиссе
    static constexpr bool is_signed = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;

    // Далее все значения представляют собой примеры:
    static constexpr float min() { return 1.17549435E-38F; }
    static constexpr float max() { return 3.40282347E+38F; }
    static constexpr float lowest() { return -3.40282347E+38F; }

    static constexpr float epsilon() { return 1.19209290E-07F; }
    static constexpr float round_error() { return 0.5F; }

    static constexpr float infinity()
        { return /* Некоторое значение */; }
    static constexpr float quiet_NaN()
        { return /* Некоторое значение */; }
    static constexpr float signaling_NaN()
        { return /* Некоторое значение */; }
    static constexpr float denorm_min() { return min(); }

    static constexpr int min_exponent = -125;
    static constexpr int min_exponent10 = -37;
    static constexpr int max_exponent = +128;
    static constexpr int max_exponent10 = +38;
```

```
    static constexpr bool has_infinity = true;
    static constexpr bool has_quiet_NaN = true;
    static constexpr bool has_signaling_NaN = true;
    static constexpr float_denorm_style has_denorm
                = denorm_absent;
    static constexpr bool has_denorm_loss = false;

    // Соответствие IEC-559:
    static constexpr bool is_iec559 = true;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo = false;
    static constexpr bool traps = true;
    static constexpr bool tinyness_before = true;

    static constexpr float_round_style round_style
                = round_to_nearest;
};
```

В заголовочных файлах **<limits.h>** и **<float.h>** определены макросы, определяющие основные свойства целых чисел и чисел с плавающей точкой, включая приведенные в следующей таблице (но не ограничиваясь ими).

Макросы предельных значений	
CHAR_BIT	Количество битов в типе **char** (обычно 8)
CHAR_MIN	Минимальное значение типа **char**
CHAR_MAX	Максимальное значение типа **char** (обычно 127, если тип **char** знаковый, и 255, если тип **char** беззнаковый)
INT_MIN	Наименьшее значение типа **int**
INT_MAX	Наибольшее значение типа **int**

Б.9.2. Стандартные математические функции

В стандартной библиотеке определены наиболее употребительные математические функции (в заголовочных файлах **<cmath>** и **<complex>**).

Стандартные математические функции	
abs(x)	Абсолютная величина
ceil(x)	Наименьшее целое число **>= x**
floor(x)	Наибольшее целое число **<= x**
round(x)	Округление к ближайшему целому (0.5 округляется в сторону увеличения)
sqrt(x)	Корень квадратный; аргумент **x** должен быть неотрицательным
cos(x)	Косинус
sin(x)	Синус
tan(x)	Тангенс
acos(x)	Арккосинус; результат неотрицательный

Стандартные математические функции	
`asin(x)`	Арксинус; возвращается результат, ближайший к нулю
`atan(x)`	Арктангенс
`sinh(x)`	Гиперболический синус
`cosh(x)`	Гиперболический косинус
`tanh(x)`	Гиперболический тангенс
`exp(x)`	Экспонента; основание равно *e* (2.718281828...)
`log(x)`	Натуральный логарифм (по основанию e); `x` должен быть положительным
`log10(x)`	Десятичный логарифм

Существуют версии этих функций, принимающие аргументы типа `float`, `double`, `long double` и `complex`. У каждой из этих функций тип возвращаемого значения совпадает с типом аргумента.

Если стандартная математическая функция не может выдать математически корректный результат, она устанавливает соответствующее значение переменной `errno`.

Б.9.3. Комплексные числа

В стандартной библиотеке определены типы для комплексных чисел `complex<float>`, `complex<double>` и `complex<long double>`. Класс `complex<Scalar>`, где `Scalar` — некий другой тип, поддерживающий обычные арифметические операции, как правило, работоспособен, но не гарантирует переносимости программ.

```
template<typename Scalar> class complex {
    // Комплексное число – это пара скалярных значений,
    // по сути – пара координат.
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i)
        :re{r}, im{i} { }
    constexpr complex(const Scalar&r) :re{r},im{Scalar{}} {}
    constexpr complex() :re{Scalar{}}, im{Scalar{}} {}
    Scalar real() { return re; } // Действительная часть
    Scalar imag() { return im; } // Мнимая часть
    // operators: = += -= *= /=
};
```

Кроме этих членов, в классе `<complex>` предусмотрено много полезных операций.

Операции с комплексными числами	
`z1+z2`	Сложение
`z1-z2`	Вычитание

Операции с комплексными числами	
`z1*z2`	Умножение
`z1/z2`	Деление
`z1==z2`	Равенство
`z1!=z2`	Неравенство
`norm(z)`	Квадрат величины `abs(z)`
`conj(z)`	Сопряженное число: если `z` — это пара `{re,im}`, то `conj(z)` — это пара `{re,-im}`
`polar(x,y)`	Возвращает комплексное число для заданных полярных координат (`rho`, `theta`)
`real(z)`	Действительная часть
`imag(z)`	Мнимая часть
`abs(z)`	Синоним `rho`
`arg(z)`	Синоним `theta`
`out << z`	Вывод комплексного числа
`in >> z`	Ввод комплексного числа

Кроме того, к комплексным числам можно применять стандартные математические функции (см. раздел Б.9.2). Примечание: в классе `complex` нет операций `<` или `%` (см. также раздел 24.9).

Б.9.4. Класс `valarray`

Объект стандартного класса `valarray` — это одномерный массив чисел; он предоставляет арифметические операции для массивов (наподобие `Matrix` из главы 24) и поддерживает срезы (slices) и шаги по индексу (strides).

Б.9.5. Обобщенные численные алгоритмы

Эти алгоритмы из заголовочного файла `<numeric>` предоставляют обобщенные версии распространенных операций над последовательностями числовых значений.

Числовые алгоритмы	
`x=accumulate(b,e,i)`	`x` — это сумма `i` и элементов последовательности [`b`,`e`)
`x=accumulate(b,e,i,f)`	Накапливание, где вместо оператора `+` выполняется функция `f`
`x=inner_product(b,e,b2,i)`	`x` — скалярное произведение последовательностей [`b`,`e`) и [`b2`,`b2+(e-b)`), т.е. сумма чисел `i` и `(*p1)*(*p2)` для всех элементов `p1` в последовательности [`b`,`e`) и всех соответствующих элементов `p2` в последовательности [`b2`,`b2+(e-b)`)
`x=inner_product(b,e,b2,i,f,f2)`	`inner_product`, но вместо операторов `+` и `*` выполняются функции `f` и `f2` соответственно

Числовые алгоритмы	
`p=partial_sum(b,e,out)`	Элемент `i` последовательности [`out`,`p`) является суммой элементов `0..i` из последовательности [`b`,`e`)
`p=partial_sum(b,e,out,f)`	`partial_sum`, где вместо оператора + выполняется функция `f`
`p=adjacent_difference(b,e,out)`	Элемент `i` последовательности [`out`,`p`) равен `*(b+i) - *(b+i-1)` для `i>0`; если `e-b>0`, то значение `*out` равно `*b`
`p=adjacent_difference(b,e,out,f)`	`adjacent_difference`, где вместо оператора - выполняется функция `f`
`itoa(b,e,v)`	Каждому элементу последовательности [`b`,`e`) присваивается значение `++v`

Например:

```
vector<int> v(100);
iota(v.begin(),v.end(),0); // v=={1,2,3,4,5 ... 100}
```

Б.9.6. Случайные числа

В заголовочном файле `<random>` стандартная библиотека предоставляет механизмы и распределения случайных чисел (§24.7). По умолчанию используется механизм `default_random_engine`, выбранный в связи с широкой применимостью и низкой стоимостью.

Предоставляемые распределения включают следующие.

Распределения	
`uniform_int_distribution<int>{low, high}`	Значение из [`low`,`high`]
`uniform_real_distribution<int>{low,high}`	Значение из [`low`,`high`)
`exponential_distribution<double>{lambda}`	Значение из [0,∞)
`bernoulli_distribution{p}`	Значение из [`true`,`false`]
`normal_distribution<double>{median,spread}`	Значение из (-∞,∞)

Распределение вызывается с механизмом в качестве аргумента, например:

```
uniform_real_distribution<> dist;
default_random_engine engn;
for (int i = 0; i<10; ++i)
    cout << dist(engn) << ' ';
```

Б.10. Работа со временем

В заголовочном файле `<chrono>` стандартная библиотека предоставляет средства для измерения времени. Часы измеряют время в количестве тактов (clock ticks) и возвращают текущее время с помощью функции `now()`. Определены трое часов:

- **system_clock**: системные часы по умолчанию
- **steady_clock**: часы **c**, для которых при последовательных вызовах **now()** выполняется условие **c.now()<=c.now()**, а время между тактами часов является константой
- **high_resolution_clock**: часы с наивысшим разрешением, доступные в системе

Количество тактов для данных часов преобразуется в обычные единицы времени, такие как **seconds**, **milliseconds** и **nanoseconds**, с помощью функции **duration_cast<>()**:

```
auto t = steady_clock::now();
// ... какие-то действия ...
auto d = steady_clock::now()-t; // Действия заняли
                                // d единиц времени
cout << "Действия заняли "
     << duration_cast<milliseconds>(d).count() << "ms";
```

В результате выполнения кода будет выведено время выполнения действий, измеренное в миллисекундах. (См. также раздел 26.6.1.)

Б.11. Функции стандартной библиотеки языка C

Стандартная библиотека языка C включена в стандартную библиотеку языка C++ с минимальными изменениями. В ней предусмотрено относительно небольшое количество функций, полезность которых подтверждена многолетним опытом использования в разнообразных предметных областях, особенно в низкоуровневом программировании. Библиотека языка C разделена на несколько категорий.

- Ввод-вывод в стиле языка C.
- Строки в стиле языка C.
- Управление памятью.
- Дата и время.
- Прочее.

Библиотека языка C содержит намного больше функций, чем описано в этой книге; рекомендуем читателям обратиться к хорошим учебникам по языку C, например к книге Kernighan and Ritchie, *The C Programming Language* (K&R).

Б.11.1. Файлы

Система ввода-вывода, описанная в заголовочном файле **<cstdio>**, основана на файлах. Указатель на файл (**FILE***) может относиться как к файлу, так и к стандартным потокам ввода и вывода, **stdin**, **stdout** и **stderr**.

Стандартные потоки доступны по умолчанию; остальные файлы должны быть открыты явным образом.

Открытие и закрытие файла	
`f=fopen(s,m)`	Открывает файловый поток для файла с именем `s` с режимом `m`
`x=fclose(f)`	Закрывает файловый поток `f`; в случае успеха возвращает `0`

"Режим" представляет собой строку, содержащую одну или несколько директив, определяющих, как именно должен быть открыт файл.

Режимы открытия файлов	
`"r"`	Для чтения
`"w"`	Для записи (предыдущее содержимое уничтожается)
`"a"`	Для добавления (данные дописываются в конец)
`"r+"`	Для чтения и записи
`"w+"`	Для чтения и записи (предыдущее содержимое уничтожается)
`"b"`	Бинарный; используется в сочетании с одним или несколькими иными режимами

В конкретной операционной системе может быть (и, как правило, так и есть) больше возможностей. Некоторые режимы могут комбинироваться, например инструкция `fopen("foo","rb")` пытается открыть файл `foo` для чтения в бинарном режиме. Режимы ввода-вывода для потоков из библиотек `stdio` и `iostream` должны быть одними и теми же (см. раздел Б.7.1).

Б.11.2. Семейство функций `printf()`

Наиболее популярными функциями в стандартной библиотеке языка C являются функции ввода-вывода. Тем не менее мы рекомендуем использовать библиотеку `iostream`, потому что она безопасна с точки зрения типов и допускает расширение. Функция форматированного вывода `printf()` используется очень широко (в том числе и в программах на языке C++) и часто имитируется в других языках программирования.

Функция `printf`	
`n=printf(fmt,args)`	Выводит форматную строку `fmt` в поток `stdout`, вставляя в соответствующие места аргументы `args`
`n=fprintf(f,fmt,args)`	Выводит форматную строку `fmt` в файл `f`, вставляя в соответствующие места аргументы `args`
`n=sprintf(s,fmt,args)`	Выводит форматную строку `fmt` в C-строку `s`, вставляя в соответствующие места аргументы `args`

В каждой версии число `n` — это количество записанных символов, а в случае неудачи — отрицательное число. На самом деле значение, возвращаемое функцией `printf()`, практически всегда игнорируется.

Объявление функции `printf()` имеет следующий вид:

```
int printf(const char* format ...);
```

Иначе говоря, эта функция получает C-строку (как правило, строковый литерал), за которой следует список, состоящий из произвольного количества аргументов произвольного типа. Смысл этих дополнительных аргументов задается спецификаторами преобразования в форматной строке, например `%c` (вывести символ) и `%d` (вывести целое число). Рассмотрим пример.

```
int x = 5;
const char* p = "asdf";
printf("значение x равно '%d', а значение p равно '%s'\n",x,p);
```

Символ, следующий за знаком `%`, управляет обработкой аргументов. Первый знак `%` применяется к первому дополнительному аргументу (в данном примере спецификатор `%d` применяется к переменной `x`), второй знак `%` относится ко второму дополнительному аргументу (в данном примере спецификатор `%s` применяется к переменной `p`) и т.д. В частности, рассмотренный выше вызов функции `printf()` приводит к следующему результату:

```
Значение x равно '5', а значение p равно 'asdf'
```

Затем происходит переход на новую строку.

В общем случае соответствие между директивой преобразования `%` и типом, к которому она применяется, проверить невозможно. Рассмотрим пример.

```
printf("x равно '%s', а p равно '%d'\n",x,p); // Ой!
```

Набор спецификаторов преобразования довольно велик и обеспечивает большую гибкость (а также много возможностей сделать ошибку). За символом `%` могут следовать спецификаторы, описанные ниже.

`-`	Необязательный знак, означающий выравнивание преобразованного значения по левому краю поля
`+`	Необязательный знак, означающий, что перед значением, имеющим тип со знаком, всегда будет стоять знак + или –
`0`	Необязательный знак, указывающий, что для выравнивания числового значения используются ведущие нули. Если в спецификации формата указан знак – или точность, то знак 0 игнорируется
`#`	Необязательный знак, указывающий, что значения с плавающей точкой будут выводиться с десятичной точкой, даже если дробная часть содержит одни нули; что будут выводиться замыкающие нули; что восьмеричные числа будут выводиться с префиксом 0, а шестнадцатеричные числа — с префиксом `0x` или `0X`
`d`	Необязательная строка цифр, задающая ширину поля. Если преобразованное значение содержит меньше символов, чем ширина поля, она будет дополнена пробелами слева (или справа, если указан индикатор выравнивания по левому

краю), чтобы заполнить всю ширину поля. Если ширина поля начинается с нуля, то вместо пробелов для дополнения значений будет использоваться нуль

. Необязательная точка, служащая разделителем между шириной поля и следующей строкой цифр

dd Необязательная строка цифр, задающая точность, т.е. количество цифр после десятичной точки для преобразований **e** и **f**, или максимальное количество символов, которое можно вывести в строке

***** Ширина поля или точность может задаваться не строкой цифр, а символом *****. В таком случае ширина поля или точность задается целочисленным аргументом

h Необязательный символ **h**, указывающий, что следующий за ним спецификатор **d**, **o**, **x** или **u** соответствует аргументу типа **short int**

l Необязательный символ (буква **l**), указывающий, что следующий за ним спецификатор **d**, **o**, **x** или **u** соответствует аргументу типа **long int**

L Необязательный символ **L**, указывающий, что следующий за ним спецификатор **e**, **E**, **g**, **G** или **f** соответствует аргументу типа **long double**

% Означает, что на печать будет выведен символ **%**; аргументы не используются

c Символ, задающий тип применяемого преобразования. Далее перечислены символы преобразования и их значение

- **d** Целочисленный аргумент в десятичной записи
- **i** Целочисленный аргумент в десятичной записи
- **o** Целочисленный аргумент в восьмеричной записи
- **x** Целочисленный аргумент в шестнадцатеричной записи
- **X** Целочисленный аргумент в шестнадцатеричной записи
- **f** Аргумент типа **float** или **double**, преобразованный в десятичный вид **[-]***ddd*.*ddd*. Число после десятичной точки задает точность аргумента. При необходимости число округляется. Если точность не указана, на печать выводятся шесть цифр; если точность явно задана как **0**, а символ **#** не указан, то ни цифры, ни десятичная точка не выводятся
- **e** Аргумент типа **float** или **double**, преобразованный в десятичный вид в научном формате **[-]***d*.*ddde+dd* или **[-]***d*.*ddde-dd*, где перед десятичной точкой стоит одна цифра, а количество цифр после десятичной точки равно точности аргумента. При необходимости число округляется. Если точность не указана, на печать выводятся шесть цифр; если точность явно задана с помощью символа **0**, а символ **#** не указан, то ни цифры, ни десятичная точка не выводятся
- **E** Действует так же, как и спецификатор **e**, но для вывода показателя степени используется буква **E** в верхнем регистре
- **g** Аргумент типа **float** или **double** выводится в стиле **d**, **f** или **e** в зависимости от того, какой из этих форматов дает максимальную точность с минимальным количеством знаков

- **G** Действует так же, как и спецификатор **g**, но для вывода показателя степени используется буква **E** в верхнем регистре
- **c** На печать выводится символьный аргумент. Нулевые символы игнорируются
- **s** Аргумент является строкой (указателем на символ), символы из строки выводятся, пока не встретится нулевой символ или не будет выведено количество символов, равное точности. Однако если точность равна 0 или не указана, будут выведены все символы, пока не будет обнаружен нулевой символ
- **p** Аргументом является указатель. Его вывод на печать зависит от особенностей реализации языка
- **u** Беззнаковый целочисленный аргумент преобразовывается в десятичный вид
- **n** Количество символов, выведенных к текущему моменту с помощью функций **printf()**, **fprintf()** и **sprintf()**; записывается в переменную типа **int**, на которую указывает соответствующий аргумент-указатель

Нулевая или слишком маленькая ширина поля никогда не приводит к усечению вывода; дополнение вывода нулями или пробелами производится только тогда, когда заданная ширина поля превышает реальную.

Поскольку в языке С нет пользовательских типов в смысле языка С++, в нем нет возможностей для определения форматов вывода для таких классов, как **complex**, **vector** или **string**.

Стандартный поток вывода **stdout** в языке С соответствует потоку **cout**. Стандартный поток ввода **stdin** в языке С соответствует потоку **cin**. Стандартный поток сообщений об ошибках **stderr** в языке С соответствует потоку **cerr**. Эти соответствия между стандартными потоками ввода-вывода в языке С и С++ настолько близки, что потоки ввода-вывода как в стиле языка С, так и стиле языка С++ могут использовать один и тот ж буфер. Например, для создания одного и того же потока вывода можно использовать комбинацию операций над объектами **cout** и **stdout** (такая ситуация часто встречается в смешанном коде, написанном на языках С и С++). Эта гибкость имеет свою стоимость. Для того чтобы получить более высокую производительность, не смешивайте операции с потоками из библиотек **stdio** и **iostream** при работе с одним и тем же потоком и вызовите **ios_base::sync_with_stdio(false)** перед выполнением первой операции ввода-вывода.

В библиотеке **cstdio** определена функция **scanf()**, т.е. операция ввода, похожая на функцию **printf()**. Рассмотрим пример.

```
int x;
char s[buf_size];
int i = scanf("x равно '%d', а s равно '%s'\n",&x,s);
```

Здесь функция **scanf()** пытается считать целое число в переменную **x**, а последовательность символов, не являющихся пробельными, в массив **s**.

Неформатные символы указывают, что они должны содержаться в строке ввода. Рассмотрим пример.

```
"x равно '123', a s равно 'string '\n"
```

Программа введет число 123 в переменную x и строку "string", за которой следует 0, в массив s. Если вызов функции scanf() завершится успешно, результирующее значение (i в предыдущем вызове) будет равно количеству присвоенных аргументов-указателей (в данном примере это число равно 2); в противном случае оно равно EOF. Этот способ индикации ввода уязвим для ошибок (например, что произойдет, если вы забудете вставить пробел после строки "string" в строке ввода?). Все аргументы функции scanf() должны быть указателями. Мы настоятельно рекомендуем не использовать эту функцию.

Как же ввести данные, если мы вынуждены использовать библиотеку stdio? Один из распространенных ответов гласит: "Используйте стандартную библиотечную функцию gets()".

```
// Очень опасный код:
char s[buf_size];
char* p = gets(s); // Считывает строку в массив s
```

Вызов p=gets(s) будет вводить символы в массив s, пока не обнаружится символ перехода на новую строку или не будет достигнут конец файла, и в конец строки s после последнего символа будет вставлен 0. Если обнаружен конец файла или возникла ошибка, то указатель p устанавливается равным NULL (т.е. 0); в противном случае он устанавливается равным s. Никогда не используйте функцию gets(s) или ее грубый эквивалент scanf("%s",s)! За прошедшие годы создатели вирусов облюбовали их слабые места: генерируя входную строку, переполняющую буфер ввода (в данном примере строку s), они научились взламывать программы и атаковать компьютеры. От таких же проблем, связанных с переполнением буфера, страдает и функция sprintf().

Библиотека stdio содержит также простые и полезные функции чтения и записи символов.

Функции ввода символов из библиотеки stdio	
x=getc(st)	Вводит символ из потока ввода st; возвращает целочисленное значение символа; если обнаружен конец файла или возникла ошибка, то x==EOF
x=putc(c,st)	Записывает символ c в поток вывода st; возвращает целочисленное значение записанного символа; если произошла ошибка, то x==EOF
x=getchar()	Считывает символ из потока stdin; возвращает целочисленное значение символа; если обнаружен конец файла или возникла ошибка, то x==EOF

Функции ввода символов из библиотеки stdio	
`x=putchar(c)`	Записывает символ `c` в поток `stdout`; возвращает целочисленное значение символа; если возникла ошибка, то `x==EOF`
`x=ungetc(c,st)`	Возвращает символ `c` обратно в поток ввода `st`; возвращает целочисленное значение символа; если возникла ошибка, то `x==EOF`

Обратите внимание на то, что результатом этих функций является число типа `int` (а не значение типа `char`, так как иначе не могло бы быть возвращено значение `EOF`). Рассмотрим типичный цикл ввода в программе на языке C.

```
int ch; /* Но не char ch; */
while ((ch=getchar())!=EOF) { /* Какие-то действия */ }
```

Не применяйте к потоку два последовательных вызова `ungetc()`. Результат такого действия непредсказуем, а значит, программа не будет переносимой.

Мы описали не все функции из библиотеки `stdio`, более полную информацию можно найти в хороших учебниках по языку C, например в книге *K&R*.

Б.11.3. C-строки

C-строки представляют собой массивы элементов типа `char`, завершающиеся нулем. Эти строки обрабатываются функциями, описанными в заголовочных файлах `<cstring>` (или `<string.h>`; заметьте: *не* `<string>`!) и `<cstdlib>`. Эти функции оперируют C-строками с помощью указателей `char*` (указатели `const char*` ссылаются на ячейки памяти, предназначенные только для чтения).

Операции над C-строками	
`x=strlen(s)`	Подсчитывает символы (исключая завершающий нуль)
`p=strcpy(s,s2)`	Копирует строку `s2` в строку `s`; диапазоны [`s`,`s+n`) и [`s2`,`s2+n`) не должны перекрываться; `p=s`; завершающий нуль копируется
`p=strcat(s,s2)`	Копирует строку `s2` в конец строки `s`; `p=s`; завершающий нуль копируется
`x=strcmp(s,s2)`	Сравнение в лексикографическом порядке: если `s<s2`, то `x` — отрицательное число; если `s==s2`, то `x==0`; если `s>s2`, то `x` — положительное число
`p=strncpy(s,s2,n)`	`strcpy`; не более `n` символов; может не копировать завершающий нуль; `p=s`
`p=strncat(s,s2,n)`	`strcat`; не более `n` символов; может не копировать завершающий нуль; `p=s`
`x=strncmp(s,s2,n)`	`strcmp`; сравнение не более `n` символов
`p=strchr(s,c)`	Устанавливает указатель `p` на первый встреченный символ `c` в строке `s`

Операции над С-строками	
`p=strrchr(s,c)`	Устанавливает указатель `p` на последний встреченный символ `c` в строке `s`
`p=strstr(s,s2)`	Устанавливает указатель `p` на первый символ строки `s`, с которого начинается подстрока, равная `s2`
`p=strpbrk(s,s2)`	Устанавливает указатель `p` на первый символ строки `s`, содержащийся в строке `s2`
`x=atof(s)`	Извлекает число типа `double` из строки `s`
`x=atoi(s)`	Извлекает число типа `int` из строки `s`
`x=atol(s)`	Извлекает число типа `long int` из строки `s`
`x=strtod(s,p)`	Извлекает число типа `double` из строки `s`; устанавливает указатель `p` на первый символ, следующий за числом типа `double`
`x=strtol(s,p)`	Извлекает число типа `long int` из строки `s`; устанавливает указатель `p` на первый символ, следующий за числом
`x=strtoul(s,p)`	Извлекает число типа `unsigned long int` из строки `s`; устанавливает указатель `p` на первый символ, следующий за числом

Обратите внимание на то, что в языке C++ функции `strchr()` и `strstr()` дублируются, чтобы обеспечить безопасность типов (они не могут преобразовать тип `const char*` в тип `char*`, как их аналоги в языке C) (см. также раздел 27.5).

Функции извлечения символов просматривают строку в стиле языка C в поисках соответственно форматированного представления числа, например `"124"` и `" 1.4"`. Если такое представление не найдено, функция извлечения возвращает `0`:

```
int x = atoi("fortytwo"); /* x равно 0 */
```

Б.11.4. Память

Функции управления памятью работают с "неформатной памятью" (без известного типа) с помощью указателей типа `void*` (указатели `const void*` указывают на ячейки памяти, предназначенные только для чтения).

Функции управления памятью в стиле языка C	
`q=memcpy(p,p2,n)`	Копирует `n` байтов из области памяти, адресованной указателем `p2`, в область памяти, адресованную указателем `p` (как функция `strcpy`); диапазоны `[p,p+n)` и `[p2,p2+n)` не должны перекрываться; `q=p`
`q=memmove(p,p2,n)`	Копирует `n` байтов из области памяти, адресованной указателем `p2`, в область памяти, адресованную указателем `p`; `q=p`; диапазоны могут перекрываться

Функции управления памятью в стиле языка C	
`x=memcmp(p,p2,n)`	Сравнивает `n` байтов из областей памяти, адресованной указателем `p2`, с соответствующими `n` байтами из области памяти, адресованной указателем `p` (как функция `strcmp`)
`q=memchr(p,c,n)`	Находит символ `c` (преобразованный в тип `unsigned char`) в диапазоне `p[0]..p[n-1]` и устанавливает указатель `q` на этот элемент; если символ `c` не найден, `q=0`
`q=memset(p,c,n)`	Копирует символ `c` (преобразованный в тип `unsigned char`) в каждую ячейку диапазона `p[0]..p[n-1]`; `q=p`
`p=calloc(n,s)`	Выделяет в динамической памяти `n*s` байтов, инициализированных нулем; если `n*s` байтов выделить невозможно, то `p=0`
`p=malloc(s)`	Выделяет `s` неинициализированных байтов в динамической памяти; если `s` байтов выделить невозможно, то `p=0`
`q=realloc(p,s)`	Выделяет `s` байтов в динамической памяти; указатель `p` должен быть результатом функции `malloc()` или `calloc()`; если возможно, повторно использует область памяти, на которую ссылается указатель `p`; если невозможно, копирует все байты области, адресованной указателем `p`, в новую область памяти; если `s` байтов выделить невозможно, то `q=0`
`free(p)`	Освобождает память, адресованную указателем `p`, который должен быть результатом функции `malloc()`, `calloc()` или `realloc()`

Функции `malloc()` и ей подобные не вызывают конструкторы, а функция `free()` не вызывает деструкторы. Не применяйте эти функции к типам, имеющим конструкторы или деструкторы. Кроме того, функция `memset()` также никогда не должна применяться к типам, имеющим конструктор.

Функции, начинающиеся с приставки `mem`, описаны в заголовочном файле `<cstring>`, а функции выделения памяти — в заголовочном файле `<cstdlib>`.

(См. также раздел 27.5.2.)

Б.11.5. Дата и время

В заголовочном файле `<ctime>` можно найти несколько типов и функций, связанных с датами и временем.

Типы, связанные с датой и временем	
`clock_t`	Арифметический тип для хранения коротких интервалов времени (до нескольких минут)
`time_t`	Арифметический тип для хранения долгих интервалов времени (до сотен лет)
`tm`	Структура для хранения даты и времени (с 1900-го года).

Структура `tm` определяется примерно так:

```
struct tm {
    int tm_sec;   // Секунда минуты [0:61]; 60 и 61
                  // представляют "високосные" секунды
    int tm_min;   // Минута часа [0,59]
    int tm_hour;  // Час дня [0,23]
    int tm_mday;  // День месяца [1,31]
    int tm_mon;   // Месяц года [0,11]; 0  — январь
                  // (заметьте: не [1,12]!)
    int tm_year;  // Год с 1900-го года; 0  — 1900-й год,
                  // 102  — 2002-й год
    int tm_wday;  // День недели, начиная с воскресенья [0,6];
                  // 0  — воскресенье
    int tm_yday;  // Дней после 1 января [0,365]; 0  — 1 января
    int tm_isdst; // Часы летнего времени
};
```

Вот функции для работы с датами и временем.

```
clock_t clock();                    // Количество тактов таймера
                                    // после старта программы

time_t time(time_t* pt);            // Текущее календарное время
double difftime(time_t t2,
                time_t t1);         // t2-t1 в секундах

tm* localtime(const time_t* pt); // локальное время для *pt
tm* gmtime(const time_t* pt);    // Гринвичское время (GMT) tm
                                 // для *pt; или 0

time_t mktime(tm* ptm);          // time_t для *ptm или
                                 // time_t(-1)

char* asctime(const tm* ptm);    // Представление *ptm в виде
                                 // С-строки
char* ctime(const time_t* t) { return asctime(localtime(t)); }
```

Пример результата вызова функции **asctime()**:

```
"Sun Sep 16 01:03:52 1973\n".
```

Неимоверное разнообразие вариантов форматирования для **tm** предоставляет функция под названием **strftime()**. Если вам нужен вывод времени в виде строки, обратитесь к документации по этой функции.

Б.11.6. Другие функции

В заголовочном файле **<cstdlib>** можно найти следующие функции.

Другие функции библиотеки stdlib	
abort()	Аварийное прекращение работы программы
exit(n)	Прекращает работу программы с кодом выхода **n**; **n==0** означает успешное завершение

Другие функции библиотеки stdlib	
`system(s)`	Выполняет системную команду, представленную в виде С-строки (зависит от системы)
`qsort(b,n,s,cmp)`	Упорядочивает массив, начинающийся с указателя `b` и состоящего из `n` элементов размера `s`, используя для сравнения функцию `cmp`
`bsearch(k,b,n,s,cmp)`	Ищет аргумент `k` в упорядоченном массиве, начинающемся с указателя `b`, состоящего из `n` элементов размера `s`, используя для сравнения функцию `cmp`

Функция для сравнения (`cmp`), используемая функциями `qsort()` и `bsearch()`, должна иметь тип

```
int (*cmp)(const void* p, const void* q);
```

Иначе говоря, функции сортировки не известен тип упорядочиваемых элементов: она просто интерпретирует массив как последовательность байтов и возвращает целое число, удовлетворяющее следующим условиям:

- оно отрицательно, если `*p` меньше, чем `*q`;
- оно равно нулю, если `*p` равно `*q`;
- оно положительно, если `*p` больше, чем `*q`.

Подчеркнем, что функции `exit()` и `abort()` не вызывают деструкторы. Если хотите вызывать деструкторы для статических объектов и объектов, созданных автоматически (см. раздел А.4.2), генерируйте исключение.

Более полную информацию о функциях из стандартной библиотеки можно найти в книге *K&R* или другом авторитетном справочнике по языку C++.

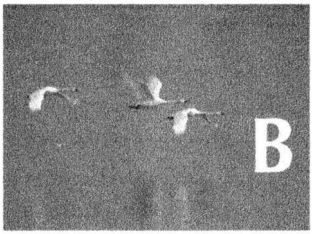
Б.12. Другие библиотеки

Исследуя возможности стандартной библиотеки, вы, конечно, найдете не так уж много полезного. По сравнению с задачами, стоящими перед программистами, и огромным количеством доступных библиотек, стандартная библиотека языка C++ является довольно скромной. Существует множество библиотек, предназначенных для решения разнообразных задач.

- Графические пользовательские интерфейсы.
- Сложные математические вычисления.
- Доступ к базам данных.
- Работа в сети.
- XML.
- Дата и время.
- Работа с файловой системой.

- Трехмерная графика.
- Анимация.
- И многое другое...

Тем не менее эти библиотеки не являются частью стандарта. Вы можете найти их в вебе или у своих друзей и коллег. Не следует думать, что полезными являются только библиотеки, представляющие собой часть стандарта.



Начала работы с Visual Studio

Вселенная не только необычнее, чем мы себе представляем, она необычнее, чем мы можем себе представить.

— Дж. Б.С. Холдейн (J.B.S. Haldane)

В этом приложении описаны шаги, которые необходимо сделать для того, чтобы ввести программу, скомпилировать и запустить ее на выполнение с помощью среды разработки Microsoft Visual Studio.

В.1. Запуск программы

Для того чтобы запустить программу, вам необходимо как-то собрать файлы вместе (чтобы, ссылаясь друг на друга — например, исходный файл на заголовочный, — они могли найти друг друга). Затем необходимо вызвать компилятор и редактор связей (если даже не потребуется сделать что-то еще, он позволит по крайней мере связать программу со стандартной библиотекой языка C++) и запустить (выполнить) программу. Существует несколько способов решения этой задачи, причем в разных операционных системах (например, в Windows и Linux) приняты разные соглашения и предусмотрены разные наборы инструментов. Тем не менее все примеры, приведенные в книге, можно выполнить во всех основных системах, используя один из распространенных наборов инструментов. В этом приложении показано, как это сделать в одной из наиболее широко распространенных систем — Microsoft Visual Studio.

Лично мы, реализуя некоторые примеры, испытали то же чувство разочарования, которое испытывает любой программист, приступая к работе с новой и странной системой. В этом случае стоит обратиться за помощью. Однако, обращаясь за помощью, постарайтесь, чтобы ваш советчик научил вас решать задачу, а не решил ее вместо вас.

В.2. Инсталляция Visual Studio

Visual Studio — это интерактивная среда разработки программ (IDE — interactive development environment) для операционной системы Windows. Если она не установлена на вашем компьютере, можете купить ее и следовать приложенным инструкциям или загрузить и инсталлировать свободно распространяемую версию Visual C++ Express с веб-страницы `www.microsoft.com/express/download`. Описание, приведенное здесь, основано на версии Visual Studio 2010. Другие версии могут немного от нее отличаться.

B.3. Создание и запуск программ

Создание и запуск программ состоит из следующих шагов.

1. Создание нового проекта.
2. Добавление в проект исходного файла на языке C++.
3. Ввод исходного текста.
4. Создание выполнимого файла.
5. Выполнение программы.
6. Сохранение программы.

B.3.1. Создание нового проекта

В среде Visual Studio *проектом* называется совокупность файлов, участвующих в создании и выполнении программы (называемой также приложением) в операционной системе Windows.

1. Откройте Visual C++, щелкнув на пиктограмме Microsoft Visual Studio 2010 или выполнив команду Пуск⇨Программы⇨Microsoft Visual Studio 2010⇨Microsoft Visual Studio 2010.
2. Выберите команду меню Файл⇨Создать⇨Проект...
3. На вкладке Установленные шаблоны выберите Visual C++.
4. В разделе шаблонов выберите Консольное приложение Win32.
5. В поле ввода Имя введите имя вашего проекта, например **`Hello, World!`**
6. Выберите каталог для своего проекта. По умолчанию предлагается путь C:\Documents and Settings\Your name\My Documents\Visual Studio 2010\Projects.
7. Щелкните на кнопке OK.
8. После этого должно открыться окно Мастера приложений Win32.
9. В левой части диалогового окна выберите пункт Параметры приложения.
10. В разделе Дополнительные параметры выберите Пустой проект.
11. Щелкните на кнопке Готово. Теперь для вашего консольного проекта будут инициализированы все установки компилятора.

B.3.2. Использование заголовочного файла `std_lib_facilities.h`

Для вашей первой программы мы настоятельно рекомендуем использовать заголовочный файл **`std_lib_facilities.h`**, который можно найти по адресу **`www.stroustrup.com/Programming/std_lib_facilities.h`**. Скопируй-

те его в каталог, выбранный в разделе В.3.1 в п. 6. (*Примечание*: сохраните этот файл как текстовый, а не как HTML-файл.) Для того чтобы использовать этот файл, вставьте в свою программу строку

```
#include "../../std_lib_facilities.h"
```

Символы "../../" сообщат компилятору, что вы разместили этот файл в каталоге C:\Documents and Settings\Your name\My Documents\Visual Studio 2010\Projects, где его смогут найти другие проекты, а не просто рядом со своим исходным файлом, так как в этом случае вам придется повторно копировать его в каталог каждого нового проекта.

В.3.3. Добавление в проект исходного файла на языке C++

Ваша программа должна состоять как минимум из одного исходного файла (хотя часто программы состоят из нескольких файлов).

1. Щелкните на пиктограмме Добавить новый элемент в строке меню (обычно вторая слева). Перед вами откроется диалоговое окно Добавление нового элемента. Выберите в категории Visual C++ пункт Код.
2. Выберите в окне шаблонов пункт Файл C++ (.cpp). Введите имя файла своей программы (`Hello, World!`) в поле Имя и щелкните на кнопке Добавить.

Итак, вы создали пустой исходный файл. Теперь вы готовы набрать текст своей программы.

В.3.4. Ввод исходного кода

В этом пункте вы можете либо ввести исходный код, набрав его непосредственно в среде разработки, либо скопировать и вставить его из другого источника.

В.3.5. Создание выполнимой программы

Если вы уверены, что правильно набрали текст исходного кода своей программы, выберите команду меню Построение⇨Построить решение или щелкните на треугольной пиктограмме справа от остальных пиктограмм верхней части окна среды разработки. Среда разработки попытается скомпилировать код и отредактировать связи вашей программы. Если этот процесс завершится успешно, вы получите в окне Вывод сообщение

```
Построение: успешно:1, с ошибками:0, без изменений:0, пропущено:0
```

В противном случае в окне Вывод появится много сообщений об ошибках. Отладьте программу, чтобы исправить ошибки, и вновь выполните команду Построить решение.

Если вы использовали треугольную пиктограмму, то программа автоматически начнет выполняться (если в ней нет ошибок). Если же вы использовали пункт меню Построить решение, вы можете явно запустить программу, как описано в разделе В.3.6.

В.3.6. Выполнение программы

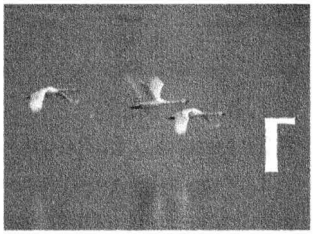
После устранения всех ошибок выполните программу, выбрав команду меню Отладка⇨Запуск без отладки.

В.3.7. Сохранение программы

Находясь в меню Файл, щелкните на пункте Сохранить все. Если вы забудете об этом и попытаетесь закрыть среду разработки без сохранения, она напомнит вам об этом.

В.4. Что дальше

Среда разработки имеет бесконечное множество свойств и возможностей. Не беспокойтесь об этом, иначе вы полностью запутаетесь. Если ваш проект станет вести себя странно, попросите опытного друга помочь вам или создайте новый проект "с нуля". Со временем потихоньку начинайте экспериментировать со свойствами и возможностями Visual Studio.



Установка FLTK

Если код и комментарии противоречат
друг другу, то, вероятно, неверны они оба.

— Норм Шрайер (Norm Schreyer)

В этом приложении описано, как загрузить, установить и использовать библиотеку FLTK.

Г.1. Введение

Мы выбрали библиотеку FLTK (Fast Light Tool Kit) как основу для нашего представления графики и решения задач, связанных с созданием графического пользовательского интерфейса, потому что она является переносимой, относительно простой, относительно широко распространенной и относительно легко устанавливаемой. Мы покажем, как установить библиотеку FLTK в среде Microsoft Visual Studio, потому что именно это интересует большинство наших студентов и вызывает у них наибольшие затруднения. Если вы используете какую-то другую систему (как некоторые из наших студентов), просто поищите в главном каталоге загружаемых файлов (раздел Г.3) инструкции, касающиеся вашей системы.

Если вы используете библиотеку, не являющуюся частью стандарта ISO C++, вам приходится ее загрузить, установить и правильно использовать в своем коде. Эта задача редко бывает тривиальной, так что установка библиотеки FLTK — неплохое упражнение, потому что загрузка и установка даже самой хорошей библиотеки часто вызывают трудности, если вы никогда не делали этого раньше. Не стесняйтесь спрашивать совета у более опытных людей, но не перекладывайте на них свои проблемы, а учитесь у них.

Отметим, что в реальности файлы и процедуры могут немного отличаться от того, что мы описываем. Например, может появиться новая версия библиотеки FLTK или измениться версия Visual Studio или вы вообще перейдете к совершенно иной среде.

Г.2. Загрузка библиотеки FLTK

Перед тем как начать работу, сначала проверьте, не установлена ли библиотека FLTK на вашем компьютере (см. раздел Г.5). Если нет, то загрузите файлы библиотеки.

1. Зайдите на веб-страницу `http://fltk.org`. (Если это почему-то у вас не получится, можете скопировать файлы с веб-сайта, посвященного этой книге (`www.stroustrup.com/Programming/FLTK`).
2. Щелкните на кнопке Download в меню навигации.
3. Выберите в выпадающем меню пункт FLTK 1.1.x и щелкните на кнопке Show Download Locations.
4. Выберите место, откуда вы будете загружать файл, и загрузите файл с расширением `.zip`.

Полученный вами файл записан в формате zip. Это формат архивации, удобный для передачи файлов по сети. Для того чтобы разархивировать файлы и привести их к обычному виду, вам нужна специальная программа, например в системе Windows примерами таких программ являются WinZip и 7-Zip.

Г.3. Установка библиотеки FLTK

При выполнении инструкций может возникнуть одна из двух проблем: за время, прошедшее с момента выхода нашей книги, что-то изменилось (это случается) или вы не понимаете терминологию (в этом случае мы ничем не можем вам помочь; извините). В последнем случае позовите друга, который вам все объяснит.

1. Распакуйте загруженный файл и откройте главный каталог, `fltk-1.1.?`. В папке Visual C++ (например, `vc2005` или `vcnet`) откройте файл `fltk.dsw`. Если вас попросят подтвердить обновление старых проектов, выберите `Yes to All`.
2. В меню Построение выберите команду Построить решение. Это может занять несколько минут. Исходный код компилируется в статические библиотеки (static link libraries), поэтому вам не придется компилировать исходный код библиотеки FLTK каждый раз при создании нового проекта. После завершения процесса закройте среду Visual Studio.
3. В главном каталоге библиотеки FLTK откройте подкаталог `lib`. Скопируйте (а не просто переместите или перетащите) все файлы с расширением `.lib`, за исключением файла README.lib (их должно быть семь) в каталог `C:\Program Files\Microsoft Visual Studio\Vc\lib`.
4. Вернитесь в главный каталог библиотеки FLTK и скопируйте подкаталог FL в каталог `C:\Program Files\Microsoft Visual Studio\Vc\include`.

Эксперты скажут вам, что есть лучшие способы установки библиотеки по сравнению с копированием файлов в каталоги `C:\Program Files\Microsoft Visual Studio\Vc\lib` и `C:\Program Files\ Microsoft Visual`

Studio\Vc\include. Они правы, но мы не стремимся сделать вас экспертами по среде Visual Studio. Если эксперты будут настаивать, попросите их продемонстрировать, как это делается.

Г.4. Использование библиотеки FLTK в среде Visual Studio

1. Создайте новый проект в среде Visual Studio, внеся одно изменение в обычной процедуре: выбирая тип проекта, выберите Проект Win32, а не Консольное приложение Win32. Убедитесь, что вы создаете Пустой проект; в противном случае мастер добавит в ваш проект много лишнего кода, который вы не поймете и вряд ли будете использовать.
2. Находясь в среде Visual Studio, выберите в главном меню команду Проект⇨Свойства.
3. В левом меню окна свойств щелкните на пункте Компоновщик. В открывающемся подменю выберите команду Ввод. В поле Дополнительные зависимости, находящемся справа, введите следующий текст:

   ```
   fltkd.lib wsock32.lib comctl32.lib fltkjpegd.lib fltkimagesd.lib
   ```

   (Следующий шаг может оказаться ненужным, поскольку в настоящее время он выполняется по умолчанию.) В поле Игнорировать все стандартные библиотеки введите следующий текст:

   ```
   libcd.lib
   ```

4. (Этот шаг может оказаться ненужным, так как в настоящее время опция /MDd включается по умолчанию.) В левом меню того же самого окна Свойства выберите команду C/C++, чтобы открыть другое подменю. Открыв подменю, выберите пункт Создание кода. В правом меню измените настройку Библиотека времени выполнения на Многопоточная отладка DLL Debug DLL (/MDd). Щелкните на кнопке OK, чтобы закрыть окно Свойства.

Г.5. Тестирование, все ли работает

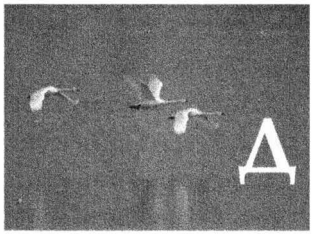
Создайте новый файл с расширением **.cpp** в новом проекте и введите следующий исходный текст. Он должен скомпилироваться без проблем.

```
#include <FL/Fl.h>
#include <FL/Fl_Box.h>
#include <FL/Fl_Window.h>
int main()
{
     Fl_Window window(200, 200, "Window title");
     Fl_Box box(0,0,200,200,"Hey, I mean, Hello, World!");
     window.show();
     return Fl::run();
}
```

Если что-то не работает, выполните следующее.

- Если вы получили сообщение компилятора, утверждающее, что файл с расширением `.lib` невозможно найти, то, возможно, вы сделали что-то не так при установке библиотеки. Внимательно проверьте п. 3, в котором указан путь для сохранения библиотечных файлов (`.lib`) на вашем компьютере.
- Если вы получили сообщение компилятора, утверждающее, что невозможно открыть файл с расширением `.h`, значит, скорее всего, вы ошиблись при установке. Внимательно проверьте п. 3, в котором указан путь для сохранения заголовочных файлов (`.h`) на вашем компьютере.
- Если вы получили сообщение редактора связей, упоминающее о неразрешенных внешних ссылках, то проблема таится в свойствах проекта.

Если наши советы вам не помогли, зовите друга...



Реализация графического пользовательского интерфейса

Когда вы, наконец, поймете, что делаете, все пойдет правильно.
— Билл Фэйрбэнк (Bill Fairbank)

В этом приложении представлены детали реализации обратных вызовов, а также классов **`Window`**, **`Widget`** и **`Vector_ref`**. В главе 16 мы не требовали от читателей знаний об указателях и операторах приведения типа, поэтому вынесли подробные объяснения в приложение.

Д.1. Реализация обратных вызовов

Обратные вызовы реализованы примерно следующим образом:

```
void Simple_window::cb_next(Address, Address addr)
    // Вызов функции Simple_window::next() для окна,
    // расположенного по адресу addr
{
    reference_to<Simple_window>(addr).next();
}
```

Раз уж вы прочитали главу 17, то вам должно быть очевидно, что аргумент **Address** должен иметь тип **void***. И разумеется, функция **reference_to<Simple_window>(addr)** должна каким-то образом создавать ссылку на объект класса **Simple_window** из указателя **addr**, имеющего тип **void***. Однако если у вас нет опыта программирования, то для вас ничто не "очевидно" и не "разумеется", пока вы не прочтете главу 17, поэтому рассмотрим использование адресов подробнее.

Как описано в разделе А.17, язык C++ предлагает способ для указания имени типа. Рассмотрим пример.

```
typedef void* Address; // Address  -  это синоним типа void*
```

Это значит, что мы можем использовать имя **Address** вместо **void***. В данном случае, используя имя **Address**, мы хотим подчеркнуть, что передаем адрес, и скрыть тот факт, что **void*** — это имя типа указателя на объект неизвестного типа.

Таким образом, функция **cb_next()** получает указатель типа **void*** с именем **addr** в качестве аргумента и — каким-то образом — немедленно преобразовывает его в ссылку **Simple_window&**:

```
reference_to<Simple_window>(addr)
```

Функция **reference_to** является шаблонной (раздел А.13).

```
template<typename W>
W& reference_to(Address pw)
    // Преобразует адрес в ссылку на объект класса W
```

```
{
    return *static_cast<W*>(pw);
}
```

Здесь мы использовали шаблонную функцию для того, чтобы самостоятельно написать операции, действующие как приведение типа **void*** к типу **Simple_window&**. Приведение **static_cast** описано в разделе 17.8.

Компилятор не имеет возможности проверить наши предположения о том, что аргумент **addr** указывает на объект класса **Simple_window**, но правила языка требуют, чтобы компилятор в этом вопросе доверял программисту. К счастью, мы оказались правы. Об этом свидетельствует тот факт, что система FLTK возвращает нам указатель, который мы ей передавали. Поскольку, передавая указатель системе FLTK, мы знали его тип, можно использовать функцию **reference_to**, чтобы "получить его обратно". Все это немного запутанно, непроверяемо и больше характерно для низкоуровневого программирования.

Получив ссылку на объект класса **Simple_window**, мы можем использовать ее для вызова функции-члена класса **Simple_window**. Рассмотрим пример (раздел 16.3).

```
void Simple_window::cb_next(Address, Address pw)
    // Вызов функции Simple_window::next() для окна,
    // расположенного по адресу pw
{
    reference_to<Simple_window>(pw).next();
}
```

Мы использовали функцию обратного вызова **cb_next()**, просто чтобы согласовать типы, необходимые для вызова совершенно обычной функции-члена **next()**.

Д.2. Реализация класса Widget

Наш интерфейсный класс **Widget** выглядит следующим образом.

```
class Widget {
    // Класс Widget - это дескриптор класса Fl_widget,
    // а не сам класс Fl_widget. Мы пытаемся не смешивать
    // наши интерфейсные классы с FLTK
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb)
            :loc(xy), width(w), height(h), label(s), do_it(cb)
    {}
    virtual ~Widget() { } // Деструктор

    virtual void move(int dx,int dy)
        { hide(); pw->position(loc.x+=dx, loc.y+=dy); show(); }
```

```
    virtual void hide() { pw->hide(); }
    virtual void show() { pw->show(); }

    virtual void attach(Window&) = 0; // Каждый объект класса
         // Widget определяет хотя бы одно действие над окном

    Point loc;
    int width;
    int height;
    string label;

    Callback do_it;

protected:

    Window* own;     // Каждый объект класса Widget
                     // принадлежит объекту класса Window
    Fl_Widget* pw;   // Каждый объект класса Widget "знает"
                     // о "своем" классе Fl_Widget
};
```

Обратите внимание на то, что наш класс **Widget** следит за "своим" компонентом библиотеки FLTK и классом **Window**, с которыми он связан. Кроме того, заметим, что для этого нам необходимы указатели, поскольку объект класса **Widget** на протяжении времени своего существования может быть связан с разными объектами класса **Window**. Ссылки или именованного объекта для этого недостаточно. (Объясните, почему.)

Объект класса **Widget** имеет местоположение (**loc**), прямоугольную форму (**width** и **height**), а также метку (**label**). Интересно, что он также имеет функцию обратного вызова (**do_it**), т.е. связывает образ объекта класса **Widget** на экране с фрагментом нашего кода. Смысл операций **move()**, **show()**, **hide()** и **attach()** должен быть очевидным.

Класс **Widget** выглядит незаконченным. Он спроектирован как класс реализации, который пользователи не должны видеть слишком часто. Его стоило бы переделать. Мы подозреваем, что все эти открытые члены и "очевидные" операции содержат подводные камни.

Класс **Widget** имеет виртуальные функции и может быть использован как базовый класс, поэтому в нем предусмотрен виртуальный деструктор (см. раздел 17.5.2).

Д.3. Реализация класса Window

Когда следует использовать указатели, а когда ссылки? Мы обсудили этот общий вопрос в разделе 8.5.6. Здесь мы лишь отметим, что некоторые программисты любят указатели и что нам нужны указатели, когда мы хотим обращаться к разным объектам в разные моменты времени.

До сих пор мы скрывали главный класс в нашей графической библиотеке — класс **Window**. Основная причина заключалась в том, что он использует указатели, а его реализация с помощью библиотеки FLTK опирается на использование динамической памяти. Вот как описан этот класса в заголовочном файле **Window.h**.

```
class Window : public Fl_Window {
public:
    // Позволяет системе выбрать место на экране:
    Window(int w, int h, const string& title);
    // Верхний левый угол в точке xy:
    Window(Point xy, int w, int h, const string& title);

    virtual ~Window() { }

    int x_max() const { return w; }
    int y_max() const { return h; }

    void resize(int ww, int hh) { w=ww, h=hh; size(ww,hh); }

    void set_label(const string& s) { label(s.c_str()); }

    void attach(Shape& s) { shapes.push_back(&s); }

    void attach(Widget&);
    void detach(Shape& s);      // Удаляет s из списка фигур
    void detach(Widget& w);     // Удаляет элемент w из окна
                                // (отключает обратные вызовы)

    void put_on_top(Shape& p); // Помещает объект p поверх
                               // всех других фигур
protected:
    void draw();
private:
    vector<Shape*> shapes;      // Фигуры, связанные с окном
    int w,h;                    // Размер окна

    void init();
};
```

Таким образом, когда мы связываем фигуру с окном, используя функцию **attach()**, мы храним указатель в векторе **shapes**, чтобы окно **Window** могло выводить соответствующую фигуру. Поскольку впоследствии мы можем отсоединить фигуру от окна с помощью функции **detach()**, нам нужен указатель. По сути, присоединенной фигурой владеет ее код; мы просто передаем объекту класса **Window** ссылку на нее. Функция **Window::attach()** преобразует свой аргумент в указатель, чтобы его можно было сохранить. Как показано выше, функция **attach()** является тривиальной; функция **detach()** немного сложнее. Открыв файл **Window.cpp**, мы видим следующее.

```
void Window::detach(Shape& s)
    // Предполагает, что первой будет удаляться
    // последняя присоединенная фигура
{
    for (vector<Shape*>::size_type i = shapes.size(); 0<i; --i)
        if (shapes[i-1]==&s)
            shapes.erase(shapes.begin()+(i-1));
}
```

Функция-член `erase()` удаляет ("стирает") значение из вектора, уменьшая размер последнего на единицу (раздел 20.7.1).

Класс `Window` используется как базовый, поэтому его деструктор — виртуальный (раздел 17.5.2).

Д.4. Реализация класса `Vector_ref`

По сути, класс `Vector_ref` имитирует вектор ссылок. Мы можем инициализировать его ссылками или указателями.

- Если объект передается объекту класса `Vector_ref` с помощью ссылки, то предполагается, что он принадлежит вызывающей функции, которая управляет его временем жизни (например, объект — это переменная, находящаяся в определенной области видимости).
- Если объект передается объекту класса `Vector_ref` с помощью указателя, то предполагается, что он размещен в памяти с помощью оператора `new`, и ответственность за его удаление несет класс `Vector_ref`.

Элемент хранится в объекте класса `Vector_ref` в виде указателя, а не как копия объекта, и имеет семантику ссылок. Например, можно поместить в вектор класса `Vector_ref<Shape>` объект класса `Circle`, не подвергаясь опасности срезки.

```
template<class T> class Vector_ref {
    vector<T*> v;
    vector<T*> owned;
public:
    Vector_ref() {}
    Vector_ref(T* a, T* b = 0, T* c = 0, T* d = 0);

    ~Vector_ref() {
        for (int i=0; i<owned.size(); ++i)
            delete owned[i];
    }

    void push_back(T& s) { v.push_back(&s); }
    void push_back(T* p) { v.push_back(p); owned.push_back(p); }

    T& operator[](int i) { return *v[i]; }
```

```
    const T& operator[](int i) const { return *v[i]; }

    int size() const { return v.size(); }
};
```

Деструктор класса `Vector_ref` удаляет каждый объект, переданный ему как указатель.

Д.5. Пример: работа с объектами `Widget`

Это законченная программа. Она демонстрирует многие из свойств классов `Widget/Window`. Мы поместили в нее минимальное количество комментариев. К сожалению, такое недостаточное комментирование программ — довольно распространенное явление. Попытайтесь выполнить эту программу и объяснить, как она работает.

По сути, при выполнении она определяет четыре кнопки.

```
#include "../GUI.h"
using namespace Graph_lib;

class W7 : public Window {
    // Четыре способа продемонстрировать, что кнопка может
    // передвигаться: показать/скрыть, изменить местоположение,
    // создать новую и присоединить/отсоединить
public:
    W7(int w, int h, const string& t);

    Button* p1;   // Показать/скрыть
    Button* p2;
    bool sh_left;

    Button* mvp;  // Переместить
    bool mv_left;

    Button* cdp;  // Создать/уничтожить
    bool cd_left;

    Button* adp1; // Активировать/деактивировать
    Button* adp2;
    bool ad_left;

    void sh();    // Действия
    void mv();
    void cd();
    void ad();
    static void cb_sh(Address, Address addr) // Обратные вызовы
        { reference_to<W7>(addr).sh(); }
    static void cb_mv(Address, Address addr)
        { reference_to<W7>(addr).mv(); }
    static void cb_cd(Address, Address addr)
```

```
        { reference_to<W7>(addr).cd(); }
    static void cb_ad(Address, Address addr)
        { reference_to<W7>(addr).ad(); }
};
```

Однако объект класса `W7` (эксперимент с объектом класса `Window` номер 7) на самом деле содержит шесть кнопок: просто две из них он скрывает.

```
W7::W7(int w, int h, const string& t)
    :Window(w,h,t),
    sh_left{true}, mv_left{true}, cd_left{true}, ad_left{true}
{
    p1 = new Button{Point{100,100},50,20,"show",cb_sh};
    p2 = new Button{Point{200,100},50,20,"hide",cb_sh};

    mvp = new Button{Point{100,200},50,20,"move",cb_mv};

    cdp = new Button{Point{100,300},50,20,"create",cb_cd};

    adp1 = new Button{Point{100,400},50,20,"activate",cb_ad};
    adp2 = new Button{Point{200,400},80,20,"deactivate",cb_ad};

    attach(*p1);
    attach(*p2);
    attach(*mvp);
    attach(*cdp);
    p2->hide();
    attach(*adp1);
}
```

В этом классе имеется четыре обратных вызова. Каждый из них проявляется в том, что нажатая кнопка исчезает и вместо нее появляется новая. Однако это достигается четырьмя разными способами.

```
void W7::sh()  // Скрывает кнопку, показывает следующую
{
    if (sh_left) {
        p1->hide();
        p2->show();
    }
    else {
        p1->show();
        p2->hide();
    }
    sh_left = !sh_left;
}

void W7::mv() // Перемещает кнопку
{
    if (mv_left) {
        mvp->move(100,0);
    }
```

```
    else {
        mvp->move(-100,0);
    }
    mv_left = !mv_left;
}

void W7::cd() // Удаляет кнопку и создает новую
{
    cdp->hide();
    delete cdp;
    string lab = "create";
    int x = 100;
    if (cd_left) {
        lab = "delete";
        x = 200;
    }
    cdp = new Button{Point{x,300}, 50, 20, lab, cb_cd};
    attach(*cdp);
    cd_left = !cd_left;
}

void W7::ad() // Отсоединяет кнопку от окна и
              // устанавливает связь с ее заменой
{
    if (ad_left) {
        detach(*adp1);
        attach(*adp2);
    }
    else {
        detach(*adp2);
        attach(*adp1);
    }
    ad_left = !ad_left;
}

int main()
{
    W7 w{400,500,"move"};
    return gui_main();
}
```

Эта программа демонстрирует основные способы добавления и удаления элементов окна, которые проявляются в их исчезновении и появлении.

Глоссарий

Часто точно выбранные слова стоят тысячи рисунков.

— Аноним

Глоссарий — это краткое объяснение слов, использованных в тексте. Ниже приведен относительно краткий словарь терминов, которые мы считаем наиболее важными, особенно на ранних этапах изучения программирования. Предметный указатель и раздел "Термины" в конце каждой главы также могут помочь читателям в этом. Более подробный и широкий словарь терминов, тесно связанных с языком C++, можно найти на веб-странице `www.stroustrup.com/glossary.html`. Кроме того, в вебе существует невероятно много специализированных глоссариев (очень разного качества). Пожалуйста, имейте в виду, что термины могут иметь несколько разных значений (некоторые из них мы указываем), причем большинство перечисленных нами терминов в другом контексте могут иметь иной смысл; например, мы не определяем слово *абстрактный* (abstract) как прилагательное, относящееся к современной живописи, юридической практике или философии.

Абстрактный класс (abstract class). Класс, который невозможно непосредственно использовать для создания объектов; часто используется для определения интерфейсов производных классов. Класс является абстрактным, если содержит чисто виртуальную функцию или защищенный конструктор.

Абстракция (abstraction). Описание сущности, которая вольно или невольно игнорирует (скрывает) детали (например, детали реализации); селективное незнание.

Адрес (address). Значение, позволяющее найти объект в памяти компьютера.

Алгоритм (algorithm). Процедура или формула для решения проблемы; конечная последовательность вычислительных шагов, приводящих к результату.

Аргумент (argument). Значение, передаваемое функции (или шаблону), в которой доступ осуществляется через параметр.

Базовый класс (base class). Класс, используемый как основание иерархии классов. Обычно базовый класс содержит одну или несколько виртуальных функций.

Байт (byte). Основная единица адресации в большинстве компьютеров. Как правило, байт состоит из восьми битов.

Бесконечная рекурсия (infinite recursion). Рекурсия, которая никогда не заканчивается, пока компьютер не исчерпает память, необходимую для хранения вызовов. На практике такая рекурсия никогда не бывает бесконечной, а прекращается в результате ошибки аппаратного обеспечения.

Бесконечный цикл (infinite loop). Цикл, в котором условие выхода из него никогда не выполняется. См. **итерация** (iteration).

Библиотека (library). Совокупность типов, функций, классов и т.п., реализованных в виде набора средств (абстракций), которые могут использоваться во многих программах.

Бит (bit). Основная единица измерения количества информации в компьютере. Бит может принимать значение 0 или 1.

Ввод (input). Значения, используемые для вычисления (например, аргументы функции или символы, набранные на клавиатуре).

Виртуальная функция (virtual function). Функция-член, которую можно перекрыть в производном классе.

Владелец (owner). Объект, ответственный за освобождение ресурса.

Время жизни (lifetime). Время, прошедшее между моментом инициализации и моментом, в который объект становится неиспользуемым (выходит из области видимости, уничтожается или прекращает существовать из-за прекращения работы программы).

Вывод (output). Значения, созданные в результате вычислений (например, результат работы функции или строка символов, выведенная на экран).

Выполнимый код (executable). Программа, готовая к выполнению на компьютере.

Вычисление (computation). Выполнение некоего кода, обычно получающего входную информацию и создающего результат.

Данные (data). Значения, используемые для вычислений.

Дескриптор (handle). Класс, обеспечивающий доступ к другому классу через указатель- или ссылку-член. См. также **копирование**, **перемещение**, **ресурс**.

Деструктор (destructor). Операция, неявно вызываемая для уничтожения объекта (например, в конце области видимости). Обычно освобождает ресурсы.

Дефект (bug). Ошибка в программе.

Диапазон (range). Последовательность значений, которую можно описать, задав начальную и конечную точки. Например, диапазон [0,5) означает значения 0, 1, 2, 3 и 4.

Единица (unit). 1) Стандартная мера, придающая значению смысл (например, км для расстояния). 2) Различимая (т.е. именованная) часть целого.

Заголовочный файл (header). Файл, содержащий объявления, используемые для распределения интерфейсов между частями программы.

Значение (value). Совокупность битов в памяти, интерпретируемая в соответствии с типом.

Идеал (ideal). Совершенный вариант того, к чему мы стремимся. Обычно мы вынуждены соглашаться на компромисс и довольствоваться лишь приближением к идеалу.

Идиома RAII (Resource Acquisition Is Initialization). Основная технология управления ресурсами, основанная на концепции области видимости.

Изменяемый (mutable). Сущность, допускающая изменение своего состояния, в противоположность неизменяемым объектам, константам и переменным.

Инвариант (invariant). Условие, которое всегда должно выполняться в заданной точке (или точках) программы; обычно используется для описания состояния (набора значений) объекта или цикла перед входом в повторяющуюся инструкцию.

Инициализация (initialize). Присваивание объекту первого (начального) значения.

Инкапсуляция (encapsulation). Защита деталей реализации от несанкционированного доступа.

Интерфейс (interface). Объявление или набор объявлений, определяющих способ вызова фрагмента кода (например, функции или класса).

Исходный код (source code). Код, созданный программистом и (в принципе) пригодный для чтения другими программистами.

Исходный файл (source file). Файл, содержащий исходный код.

Итератор (iterator). Объект, идентифицирующий элемент последовательности.

Итерация (iteration). Повторяющееся выполнение фрагмента кода; см. **рекурсия**.

Класс (class). Определенный пользователем тип, который может содержать данные-члены, функции-члены и типы-члены.

Код (code). Программа или часть программы; термин используется как для исходного, так и для объектного кода.

Компилятор (compiler). Программа, превращающая исходный код в объектный.

Компромисс (trade-off). Результат согласования нескольких принципов проектирования и реализации.

Конкретный класс (concrete class). Класс, объекты которого можно создать.

Константа (constant). Значение, которое невозможно изменить (в заданной области видимости); неизменное значение.

Конструктор (constructor). Операция, инициализирующая (конструирующая) объект. Обычно конструктор устанавливает инвариант и часто запрашивает ресурсы, необходимые для использования объектов (которые обычно освобождаются деструктором).

Контейнер (container). Объект, содержащий элементы (другие объекты).

Концепция (concept). 1) Понятие, идея. 2) Набор требований, обычно для аргумента шаблона.

Копирование (copy). Операция, которая делает два объекта имеющими значения, которые при сравнении оказываются равными. См. также **перемещение**.

Корректность (correctness). Программа (или фрагмент программы) считается правильной (корректной), если она соответствует своим спецификациям. К сожалению, спецификация может быть неполной или противоречивой или не соответствовать разумным ожиданиям пользователя. Таким образом, для того чтобы создать приемлемый код, мы иногда должны сделать больше, чем просто следовать формальной спецификации.

Литерал (literal). Обозначение, которое непосредственно задает число, например литерал 12 задает целое число, равное "двенадцать".

Массив (array). Однородная последовательность элементов, обычно нумерованная, например `[0,max)`.

Нагромождение возможностей (feature creep). Стремление добавлять избыточные функциональные возможности в программу "на всякий случай".

Неинициализированный (uninitialized). (Неопределенное) состояние объекта до его инициализации.

Область видимости (scope). Область текста программы (исходного кода), в которой можно ссылаться на имя сущности.

Обобщенное программирование (generic programming). Стиль программирования, нацеленный на проектирование и эффективную реализацию алгоритмов. Обобщенный алгоритм должен работать с

аргументами любого типа, соответствующими его требованиям. В языке C++ обобщенное программирование обычно использует шаблоны.

Объект (object). 1) Инициализированная область памяти известного типа, в которой записано какое-то значение данного типа. 2) Область памяти.

Объектно-ориентированное программирование (object-oriented programming). Стиль программирования, нацеленный на проектирование и использование классов и иерархий классов.

Объектный код (object code). Результат работы компилятора, представляющий собой входную информацию для редактора связей, который, в свою очередь, создает выполняемый код.

Объектный файл (object file). Файл, содержащий объектный код.

Объявление (declaration). Спецификация имени с типом.

Округление (rounding). Преобразование значения в ближайшее менее точное значение по математическим правилам.

Операция (operation). Нечто, выполняющее какое-то действие, например функция или оператор.

Определение (definition). Объявление сущности, содержащее всю необходимую информацию для его использования в программе. Упрощенное определение: объявление, выделяющее память.

Отладка (debugging). Поиск и удаление ошибок из программы; обычно имеет менее систематический характер, чем тестирование.

Ошибка (error). Несоответствие между разумными ожиданиями относительно поведения программы (часто выраженными в виде требований или руководства пользователя) и тем, что программа делает на самом деле.

Парадигма (paradigm). Несколько претенциозное название стиля проектирования или программирования. Часто (ошибочно) считают, что существует парадигма, превосходящая все остальные.

Параметр (parameter). Объявление явной входной информации для функции или шаблона. При вызове функция может обращаться к аргументам по именам своих параметров.

Перегрузка (overload). Определение двух функций или операторов с одинаковыми именами, но разными типами аргументов (операндов).

Перекрытие (override). Определение в производном классе функции, имя и типы аргументов которой совпадают с именем и типами аргументов виртуальной функции из базового класса; в результате эту функцию можно вызывать через интерфейс базового класса.

Переменная (variable). Именованный объект заданного типа; содержит значение, если был инициализирован.

Перемещение (move). Операция, которая передает значение из одного объекта в другой, оставляя в первом значение, представляющее "пустое состояние". См. также **копирование**.

Переполнение (overflow). Создание значения, которое невозможно сохранить в предназначенной для него области памяти.

Подтип (subtype). Производный тип; тип, обладающий всеми свойствами базового типа и, возможно, дополнительными возможностями.

Последовательность (sequence). Совокупность элементов, которую можно обойти последовательно.

Постусловие (post-condition). Условие, которое должно выполняться при выходе из фрагмента кода, например, функции или цикла.

Предусловие (pre-condition). Условие, которое должно выполняться при входе в фрагмент кода, например в функцию или цикл.

Приближение (approximation). Нечто (например, число или проект), близкое к совершенству или идеалу (числу или проекту). Часто аппроксимация является результатом компромисса между идеалами.

Приложение (application). Программа или коллекция программ, рассматриваемая пользователями как единая сущность.

Программирование (programming). Искусство выражения решений задач в виде кода.

Программное обеспечение (software). Совокупность фрагментов кода и связанных с ними данных; часто используется как синоним слова "**программа**".

Программный код (program code) (возможно, вместе со связанными с ним данными). То, что полностью готово к выполнению на компьютере.

Проект (design). Общее описание того, как должно работать программное обеспечение, чтобы соответствовать своей спецификации.

Производный класс (derived class). Класс, являющийся наследником одного или нескольких базовых классов.

Псевдокод (pseudo code). Описание вычислений, записанное с помощью неформальных обозначений, а не языка программирования.

Псевдоним (alias). Альтернативный способ обращения к объекту; часто имя, указатель или ссылка.

Реализация (implementation). 1) Действие, означающее написание и тестирование кода. 2) Код, реализующий программу.

Регулярное выражение (regular expression). Запись шаблонов в символьных строках.

Редактор связей (linker). Программа, объединяющая файлы объектного кода с библиотеками в выполнимый модуль.

Рекурсия (recursion). Вызов функцией самой себя; см. также **итерация**.

Ресурс (resource). Нечто, чем можно завладеть и что впоследствии следует освободить, например дескрипторы файлов, блокировка или память. См. также **дескриптор**, **владелец**.

Система (system). 1) Программа или набор программ для выполнения определенной задачи на компьютере. 2) Сокращение от словосочетания "операционная система", т.е. базовая среда для выполнения программ и инструменты компьютера.

Слово (word). Основная единица памяти компьютера, обычно используется для хранения целых чисел.

Сложность (complexity). С трудом поддающееся точному определению понятие, представляющее собой некую меру трудности процесса поиска решения задачи или свойство самого решения. Иногда под *сложностью* понимают просто оценку количества операций, необходимых для выполнения алгоритма.

Сокрытие (hiding). Действие, предотвращающее непосредственный доступ к информации. Например, имя из вложенной (внутренней) области видимости, совпадающее с именем из охватывающей (внешней) области видимости, может быть недоступно для непосредственного использования.

Сокрытие информации (information hiding). Действие, связанное с отделением интерфейса и реализации друг от друга, в результате которого детали реализации остаются за пределами внимания пользователя и возникает абстракция.

Состояние (state). Набор значений.

Спецификация (specification). Описание того, что должен делать фрагмент кода.

Ссылка (reference). 1) Значение, описывающее расположение в памяти типизированного значения. 2) Переменная, содержащая такое значение.

Стандарт (standard). Официально согласованное определение чего-либо, например языка программирования.

Стиль (style). Совокупность методов программирования, обеспечивающая согласованное использование возможностей языка. Иногда используется в очень ограниченном смысле, касающемся правил выбора имен и внешнего вида текста программы.

Стоимость (cost). Затраты (например, время работы программиста, время выполнения программы или объем памяти), связанные с производством программы или ее выполнением. В идеале стоимость должна быть функцией сложности.

Строка (string). Последовательность символов.

Супертип (supertype). Базовый тип; тип, обладающий подмножеством свойств производного типа.

Сценарий использования (use case). Конкретный (как правило, простой) пример использования программы, предназначенный для ее тестирования и демонстрации возможностей.

Тестирование (testing). Систематический поиск ошибок в программе.

Тип (type). То, что определяет набор возможных значений и допустимых операций над объектом.

Требование (requirement). 1) Описание желательного поведения программы или части программы. 2) Описание предположений об аргументах функции или шаблона.

Указатель (pointer). 1) Значение, используемое для идентификации в памяти объекта, имеющего тип. 2) Переменная, содержащая такое значение.

Усечение (truncation). Потеря информации в результате преобразования одного типа в другой тип, который не может точно представить преобразованное значение.

Утверждение (assertion). Утверждение, вставленное в программу для того, чтобы указать, какое условие всегда должно выполняться в данной точке программы.

Файл (file). Контейнер, содержащий информацию в постоянной памяти компьютера.

Функция (function). Именованный фрагмент кода, который можно активизировать (вызвать) из разных частей программы; логическая единица вычислений.

Целое число (integer). Целое число в математическом смысле, например 42 или –99.

Цикл (loop). Фрагмент кода, выполняющийся повторно; в языке C++ циклы, как правило, реализуются инструкцией **for** или **while**.

Число с десятичной точкой (floating-point number). Компьютерное приближение действительного числа, например 7.93 и 10.78e–3.

Чисто виртуальная функция (pure virtual function). Виртуальная функция, которая должна быть перекрыта в производном классе.

Шаблон (template). Класс (или функция), параметризованный одним или несколькими типами или значениями времени компиляции; основная конструкция в языке C++, поддерживающая обобщенное программирование.

Язык программирования (programming language). Язык для выражения программ.

Библиография

1. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools. Second Edition* (обычно именуемая "Книгой Дракона"). Addison-Wesley, 2006. ISBN 0321486811. (Русский перевод — Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. *Компиляторы: принципы, технологии и инструментарий.* — М: Издательский дом "Вильямс", 2008.)
2. Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services.* Addison-Wesley, 2006. ISBN 0321369440.
3. Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages, Volume 2.* Addison-Wesley, 1996. ISBN 0201895021.
4. Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4.* Prentice Hall, 2006. ISBN 0131872494.
5. Boost.org. "A Repository for Libraries Meant to Work Well with the C++ Standard Library." **www.boost.org**.
6. Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, ...)." **http://swtch.com/~rsc/regexp/regexp1.html**.
7. dmoz.org. **http://dmoz.org/Computers/Programming/Languages**.
8. Freeman, T. L., and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall, 1992. ISBN 0136515975.
9. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. ISBN 0201633612.
10. Goldthwaite, Lois, ed. *Technical Report on C++ Performance.* ISO/IEC PDTR 18015. **www.stroustrup.com/performanceTR.pdf**.
11. Gullberg, Jan. *Mathematics — From the Birth of Numbers.* W. W. Norton, 1996. ISBN 039304002X.
12. Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III).* San Diego, CA, 2007. **http://portal.acm.org/toc.cfm?id=1238844**.
13. ISO/IEC 9899:2011. *Programming Languages — C.* The C standard.
14. ISO/IEC 14882:2011. *Programming Languages — C++.* The C++ standard.
15. Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988. ISBN 0131103628.
16. Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition.* Addison-Wesley, 1997. ISBN 0201896842. (Русский перевод — Кнут Дональд Эрвин. *Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.* — М.: Издательский дом "Вильямс", 2000.)

17. Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example.* Addison-Wesley, 2000. ISBN 020170353X. (Русский перевод — Кёниг Эндрю, Му Барбара. *Эффективное программирование на C++.* — М.: Издательский дом "Вильямс", 2002.)

18. Langer, Angelika, and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference.* Addison-Wesley, 2000. ISBN 0321585585.

19. Lippman, Stanley B., Jos Lajoie, and Barbara E. Moo. *The C++ Primer, Fifth Edition.* Addison-Wesley, 2005. ISBN 0321714113. (Читайте только пятое издание.)

20. Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program." Document Number 2RDU00001 Rev C. December 2005. Общеизвестен как "JSF++." **www.stroustrup.com/JSF-AV-rules.pdf.**

21. Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts — The Programmers Who Created the Software Revolution.* Basic Books, 2002. ISBN 978-0465042265.

22. Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Addison-Wesley, 2001. ISBN 0201749629.

23. Meyers, Scott. *Effective C++: 55 Specifi c Ways to Improve Your Programs and Designs, Third Edition.* Addison-Wesley, 2005. ISBN 0321334876.

24. Programming Research. *High-Integrity C++ Coding Standard Manual Version 2.4.* **www.programmingresearch.com.**

25. Richards, Martin. *BCPL — The Language and Its Compiler.* Cambridge University Press, 1980. ISBN 0521219655.

26. Ritchie, Dennis. "The Development of the C Programming Language." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2). ACM SIGPLAN Notices,* Vol. 28 No. 3, 1993.

27. Salus, Peter H. *A Quarter Century of UNIX.* Addison-Wesley, 1994. ISBN 0201547775.

28. Sammet, Jean E. *Programming Languages: History and Fundamentals.* Prentice Hall, 1969. ISBN 0137299885.

29. Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns.* Addison-Wesley, 2002. ISBN 0201604647.

30. Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks.* Addison-Wesley, 2003. ISBN 0201795256.

31. Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition.* O'Reilly, 2005. ISBN 0596101058.

32. Scott, Michael L. *Programming Language Pragmatics.* Morgan Kaufmann, 2000. ISBN 1558604421.

33. Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition.* Addison-Wesley, 2003. ISBN 0321193628.

34. Shepherd, Simon. "The Tiny Encryption Algorithm (TEA)." `www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf` и `http://143.53.36.235:8080/tea.htm`.

35. Stepanov, Alexander. `www.stepanovpapers.com`.

36. Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.

37. Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.

38. Stroustrup, Bjarne. "A History of C++: 1979–1991." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2). ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.

39. Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.

40. Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.

41. Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *The C/C++ Users Journal*, July, Aug., and Sept. 2002.

42. Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991–2006." *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. `http://portal.acm.org/toc.cfm?id=1238844`.

43. Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.

44. Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.

45. Stroustrup, Bjarne. Author's home page, `www.stroustrup.com`.

46. Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622. (Русский перевод — Саттер, Герб. *Решение сложных задач на C++*. — М.: Издательский дом "Вильямс", 2002.)

47. Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586. (Русский перевод — Саттер Герб, Александреску Андрей. *Стандарты программирования на C++*. — М.: Издательский дом "Вильямс", 2005.)

48. University of St. Andrews. The MacTutor History of Mathematics archive. `http://www-gap.dcs.st-and.ac.uk/~history`.

49. Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.

50. Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002. ISBN 0201796198.

51. Wood, Alastair. *Introduction to Numerical Analysis*. Addison-Wesley, 2000. ISBN 020134291X.

Предметный указатель

Б

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

P

С

Т

У

Ф

Фотографии

Раздел	Фотография
0.7	Бьярне Страуструп, 2005. Источник: Bjarne Stroustrup
0.7	Лоуренс "Пит" Петерсен, 2006. Источник: Dept. pf Computer Science, Texas A&M Unversity
1.5.1	Цифровые часы фирмы Casio. Источник: `www.casio.com`
1.5.1	Аналоговые часы фирмы Casio. Источник: `www.casio.com`
1.5.2	Корабельный дизельный двигатель MAN 12K98ME. MAN Burgmeister & Waine. Источник: MAN Diesel A/S, Copenhagen, Denmark
1.5.2	Emma Maersk; крупнейший контейнеровоз в мире; порт приписки Århus. Denmark. Источник: Getty Images
1.5.3	Цифровой телефонный коммутатор. Источник: Alamy Images
1.5.3	Мобильный телефон Sony-Ericsson W-920 с музыкальной системой, телефонными функциями и возможностью выхода в веб. Источник: `www.sonyericsson.com`
1.5.3	Операционный зал Нью-Йоркской фондовой биржи на Уолл-стрит. Источник: Alamy Images
1.5.3	Графическое представление частей, из которых состоит Интернет, созданное Стивеном Айком (Stephen G. Eick). Источник: S. G. Eick
1.5.4	Сканер CAT. Источник: Alamy Images
1.5.4	Компьютерная хирургия. Источник: Da Vinci Surgical Systems, `www.intuitivesurgical.com`
1.5.5	Обычная комплектация компьютера (экран слева соединен с настольной системой под управлением системы Unix, а справа — с ноутбуком, на котором установлена операционная система Windows). Источник: Bjarne Stroustrup
1.5.5	Стойка компьютеров на серверном узле. Источник: Istockphoto
1.5.6	Вид с марсохода. Источник: NASA, `www.nasa.gov`
22.2.1	Команда EDCAS в 1949 году. В центре Морис Уилкс (Maurice Wilkes), без галстука — Дэвид Уилер (David Wheeler). Источник: The Cambridge University Computer Laboratory
22.2.1	Дэвид Уилер читает лекцию (примерно в 1974 году). Источник: The Cambridge University Computer Laboratory
22.2.2.1	Джон Бэкус (John Backus, 1996). Copyright: Louis Fabian Bachrach. Коллекцию фотографий пионеров компьютерной эры смотрите в книге Christopher Morgan, *Wizards and their wonders: portraits in computing*, ACM Press Press, 1997. ISBN 0-89791-960-2

22.2.2.2 Грейс Мюррей Хоппер (Grace Murray Hopper). Источник: Computer History Museum

22.2.2.2 Жучок Грейс Мюррей Хоппер (Grace Murray Hopper). Источник: Computer History Museum

22.2.2.3 Джон Маккарти (John C. McCarty) в 1967 году, Станфорд. Источник: Stanford Unversity

22.2.2.3 Джон Маккарти в 1996 году. Источник: Louis Fabian Bachrach

22.2.3.1 Фотография Питера Наура (Peter Naur), сделанная Брайаном Ранделлом (Brian Randell) в Мюнхене в 1968 году, где они вместе редактировали отчет, давший старт развитию компьютерных наук. Воспроизводится с разрешения Брайана Ранделла

22.2.3.1 Питер Наур, портрет маслом, написанный Дуо Дуо Жангом (Duo Duo Zhuang) в 1995 году. Воспроизводится с разрешения Эрика Фрокьяера (Erik Frøkjær)

22.2.3.1 Эдсгер Дейкстра (Edsger Dijkstra). Источник: Wikimedia Commons

22.2.3.2 Никлаус Вирт (Nisklaus Wirth). Источник: N. Wirth

22.2.3.2 Никлаус Вирт (Nisklaus Wirth). Источник: N. Wirth

22.2.3.3 Жан Ишби (Jean Ichbiah). Источник: Ada Information Clearinghouse

22.2.3.3 Леди Лавлейс (Lady Lovelace), 1838. Старинная фотография. Источник: Ada Information Clearinghouse

22.2.4 Кристен Нюгард (Kristen Nygaard) и Оле-Йохан Даль (Ole-Johan Dal), примерно 1968 год. Источник: University of Oslo

22.2.4 Кристен Нюгард, примерно 1996 год. Источник: University of Oslo

22.2.4 Оле-Йохан Даль, 2002. Источник: University of Oslo

22.2.5 Деннис Ритчи (Dennis M. Ritchie) и Кен Томпсон (Ken Thompson), примерно 1978 год. Copyright: AT&T Bell Labs

22.2.5 Деннис Ритчи, 1996. Источник: Louis Fabian Bachrach

22.2.5 Дуг Мак-Илрой (Doug McIlroy), примерно 1990 год. Источник: GerardHolzmann

22.2.5 Брайан Керниган (Brian W. Kernigan), примерно 2004 год. Источник: B.W. Kernigan

22.2.6 Бьярне Страуструп, 1996. Источник: Bjarne Stroustrup

22.2.6 Алекс Степанов (Alex Stepanov), 2003. Источник: Bjarne Stroustrup

25.1 Дизельный двигатель. Источник: Mogens Hansen, MAN B&W, Copenhagen

27.1 AT&T Bell Labs' Murray Hill Research Center, примерно 1990 год. Источник: AT&T Bell Labs

Введение в программирование от создателя языка C++

- ***Подготовка к созданию реальных программ***
  Автор книги предполагает, что читатели в конце концов начнут писать нетривиальные программы либо в качестве профессиональных разработчиков программного обеспечения, либо в качестве программистов, работающих в других областях науки и техники.
- ***Упор на основные концепции и методы***
  Основные концепции и методы программирования в книге излагаются глубже, чем это принято в традиционных вводных курсах. Этот подход служит основательным фундаментом для разработки полезных, правильных, понятных и эффективных программ.
- ***Программирование на современном языке C++ (C++11 и C++14)***
  Книга представляет собой введение в программирование, включая объектно-ориентированное и обобщенное программирование. Одновременно она представляет собой введение в язык C++, один из широко применяющихся языков программирования в современном мире. В книге описаны современные методы программирования на C++, включая стандартную библиотеку и возможности C++11 и C++14, позволяющие упростить программирование.
- ***Для начинающих программистов и всех, кто хочет научиться программировать***
  Книга предназначена в основном для людей, никогда ранее не программировавших, и опробована на более чем тысяче студентов университета. Однако и опытные программисты, и студенты, уже изучившие основы программирования, найдут в книге много полезной информации, которая позволит им перейти на еще более высокий уровень мастерства.
- ***Широкий охват тем***
  Первая половина книги охватывает широкий спектр основных понятий, методов проектирования и программирования, свойств языка C++ и его библиотек. Это позволит читателям писать программы, выполняющие ввод и вывод данных, вычисления и построение простых графических изображений. Во второй половине рассматриваются более специализированные темы (такие, как обработка текста, тестирование и язык C). В книге содержится много справочного материала. Исходные тексты программ и иные материалы читатели могут найти на веб-сайте автора.

ОБ АВТОРЕ

Бьярне Страуструп — разработчик языка программирования C++ и первой его реализации; он также является автором известной книги *The C++ Programming Language*. Сейчас он работает управляющим директором в технологическом отделе Morgan Stanley; приглашенным профессором в Колумбийском университете и является почетным профессором Университета Техаса. Страуструп — член Национальной академии технических наук США (U.S. National Academy of Engineering). До перехода на академическую работу он много лет работал в исследовательском центре AT&T Bell Labs. Он также является одним из членов-учредителей международного комитета по стандартизации языка C++ (ISO C++).

ISBN 978-5-8459-1949-6

ВИЛЬЯМС
Издательский дом "Вильямс"
http://www.williamspublishing.com

Изображение на обложке: © ssguy/Shutterstock.com

Addison-Wesley

ALWAYS LEARNING

PEARSON