Опыт не требуется

Книги этой серии наглядно свидетельствуют, что новичка можно научить языку и хорошему стилю программирования, не повергая его в тоску и уныние.
– Лу Гринзоу, обозреватель Dr. Dobb's Journal.

Исходные тексты всех игр находятся на CD

Дирк Хенкеманс, Марк Ли

Программирование на C++

Premier Press
СИМВОЛ

C++ Programming

for the Absolute Beginner

Dirk Henkemans and Mark Lee

Premier Press

Опыт не требуется

Программирование на C++

Дирк Хенкеманс и Марк Ли

Символ

Санкт-Петербург
2005

Серия «Опыт не требуется»

Дирк Хенкеманс, Марк Ли

Программирование на C++

Перевод М. Зислиса

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Лосев*
Художник	*В. Гренда*
Корректура	*С. Беляева*
Верстка	*Н. Гриценко*

Хенкеманс Д., Ли М.
Программирование на C++. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 416 с., ил.
ISBN 5-93286-050-2

Для тех, кто мало знаком с программированием, но ищет хороший учебник по C++, эта книга станет идеальным выбором. Написанная профессиональными разработчиками и отличающаяся легким стилем изложения, она обучает принципам программирования на примерах создания простых игр. Прочитав ее, вы приобретете навыки, необходимые для создания более сложных программ на C++, и узнаете, как использовать их в реальных приложениях. Изучите многочисленные приемы, которые применимы не только к C++, но и к программированию в целом, поэтому полученные знания будут вам полезны при освоении других языков программирования.

Вы узнаете, что такое переменные и управляющие операторы, функции и объектно-ориентированное программирование, пространства имен и массивы. Научитесь программировать для Windows, создавать программы шифрования, отлаживать ошибки и грамотно обрабатывать исключения, эффективно использовать потоки и файлы, а также разрабатывать игры с помощью библиотеки DirectX.

ISBN 5-93286-050-2
ISBN 1-93184-143-8 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 19.11.2004. Формат 70x100 1/16 . Печать офсетная.
Объем 26 печ. л. Доп. тираж 1000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Всем детям двадцать первого века –
вы способны осуществить все,
что можете представить в своих мечтах.

Предисловие

Индустрия видеоигр является уникальной: она регулярно вбирает все новшества основных направлений развития компьютерных наук от трехмерной графики и искусственного интеллекта до теории операционных систем и проектирования баз данных. Если вы разрабатываете коммерческую игру, то рано или поздно столкнетесь с задачами, принадлежащими каждой из этих областей. Некоторые из задач могут потребовать применения специальных языков, но в конечном итоге есть только два языка, столь же привычных для игровой индустрии, как пицца, кофеиносодержащие напитки и критические дни в нашей жизни. Несколько коммерческих игр было написано и на Java (языке, весьма похожем на C++), но практически каждая игра, в которую вам приходилось играть, написана на C или C++. Неважно, работает она на PC, игровой консоли или вовсе на игровом автомате – все шансы за то, что ее сердцем является код C или C++. Даже в случаях, когда требования к производительности диктуют необходимость создания подпрограммы на ассемблере с целью повышения скорости работы, как правило, первый вариант этой подпрограммы пишется на C или C++.

За многие годы работы в этой индустрии я провел более сотни собеседований с претендентами на вакансии, связанные с программированием, и, кроме того, прочитал более тысячи резюме. При отборе я непрестанно ищу у кандидатов комбинацию трех качеств. Первое качество – умение решать проблемы, постоянным источником которых служат непрекращающиеся изменения в технологиях и жесткая конкуренция в среде разработчиков игр. Как следствие, отточенные навыки разрешения проблем – не только роскошь, но и необходимость. Во-вторых, кандидат на должность должен иметь опыт работы во всем спектре компьютерных наук. Ведь будучи хорошим специалистом в одной области, всегда можно столкнуться с проблемой, решение которой лежит за пределами компетенции. И наконец, я требую отличного владения C/C++. Для программиста эти языки равноценны кистям и краскам для художника. Это орудия труда, и значит, они должны быть идеально отточены.

Сегодня C++ повсеместно используется для обучения программированию, но так было не всегда. Я до сих пор помню, как познакомился с программированием на языке C. До этого момента мой опыт в программировании сводился к языкам Basic (на котором я написал свою первую игру), Pascal и Fortran. Но я слышал о C, и, по слухам, этот язык стоило изучать. Я с нетерпением ожидал начала следующего

курса по информатике: «Введение в языки программирования». Я полагал, что в этом курсе меня научат программировать на C, и ошибся. Единственная ссылка на язык C в этом курсе выглядела так: «Вот задание. Напишите программу на C. Сдайте ее в среду». Ну ничего, подумал я. Есть учебник по языку C. Однако выяснилось, что он был посвящен доступу к информации ОС UNIX из программ на языке C. В книге рассказывалось о том, как получать идентификаторы процессов и выполнять команды интерпретатора, и значит, она была для меня бесполезна, т. к. в ней не объяснялось, как прочитать файл или создать функцию.

Кое-как я ухитрился выполнить задание и даже приобрести знания в процессе. Это был не лучший способ изучить новый язык, но моя первая встреча с C++ выглядела еще хуже. Окончив университет, я поступил на работу. В мои задачи входило создание программного обеспечения для исследовательских проектов спортивного факультета. Один из проектов, доставшихся мне от предшественника, был завершен лишь наполовину и написан на языке C++. И снова мне пришлось учиться плавать в боевых условиях. На этот раз у меня был доступ к справочнику по функциям, в котором был описан синтаксис языка, но не рассматривались способы его применения. В то время я готов был пойти на преступление ради книги, которую вы держите в руках. Конечно, я преувеличиваю, но невозможно переоценить достоинства иного метода изучения C++, последовательного и доступного. Читая эту книгу, вспоминайте с сочувствием тех из нас, у кого не было столь замечательного учебника.

Scott Greig
Director of Programming
Bio Ware Corp.

Благодарности

В процессе создания книги участвуют многие замечательные люди, и эта книга не стала исключением. Хотя нам трудно осознать объемы времени и сил, вложенные в эту книгу, мы знаем, что эти объемы были внушительными.

Прежде всего, спасибо нашим родителям за то, что они нас вырастили, и за поддержку, которую мы всегда в них находим.

Спасибо нашему издательству, Premier Press, за эту книгу. Отдельной благодарности заслуживает Мелоди Лейн (Melody Layne), менеджер по работе с авторами, которая поверила в нашу идею и поддержала ее. Мелоди прекрасно известно, что именно эту книгу мы всегда хотели увидеть на полке.

Мы выражаем искреннюю благодарность Грегу Перри (Greg Perry), координирующему редактору и техническому рецензенту. Спасибо за великолепные отклики, Грег, и за тщательную проверку кода в нашей книге.

Мельба Хоппер (Melba Hopper), выпускающий и литературный редактор, заслуживает отдельной страницы благодарностей. Ее рука касалась всех строк этой книги, изменяя и улучшая их. Мельба, ты замечательно с нами ладила, постоянно объясняла, что следует исправить в рукописи, и служила неиссякаемым источником информации и поддержки, в которых мы нуждались, чтобы продолжать работу. И самое главное, ты сделала процесс работы над книгой увлекательным. Спасибо!

Отдельное спасибо всем остальным, кто участвовал в подготовке публикации этой книги. Вот эти люди: Энди Харрис (Andy Harris), редактор серии «Для начинающих», Эрли Хартман (Arlie Hartman), автор компакт-диска; Шон Морнингстар (Shawn Morningstar), дизайнер-верстальщик; художники из Argosy, которые превратили наши наброски в нечто удобоваримое; Дженни Смит (Jeannie Smith), корректор; а также Джонна ВанХуз Динс (Johnna VanHoose Dinse), автор указателя. Все вы сыграли важную роль в достижении полученного результата.

Мы восхваляем Скотта Грейга (Scott Greig), ведущего программиста BioWare Corp. и автора предисловия к этой книге. Скотт, ты наш кумир. Не будь тебя, кем бы мы стремились стать?

И наконец, отдельное спасибо Нолану Барду (Nolan Bard), который в четыре утра помогал нам закончить книгу в срок, а также Джеки Нэги (Jackie Nagy) за его поддержку и за то, что не оставил Дирка, когда тот писал книгу.

Об авторах

Дирк Хенкеманс – создатель любительских руководств по разработке игр и автор веб-сайта EastCoastGames.com. Он также является одним из основателей FireStorm Studios, растущей компании, специализирующейся на мультимедиа-приложениях.

Марк Ли – второй основатель FireStorm Studios, работал компьютерным консультантом и оператором текстовой пользовательской сети. Бегло говорит на C, Java, C++, Visual Basic, разнообразных диалектах ассемблера и систем управления базами данных.

Введение

C++ является одним из наиболее широко применяемых языков программирования, индустриальным стандартом для создания приложений всевозможного рода. Кроме того, это очень рациональный язык, позволяющий использовать ресурсы более эффективно, чем Visual Basic или Delphi. По большому счету, благодаря функциональности и стилю C++ может оказаться единственным из языков, не ориентированных на работу с веб-средой, который вам когда-либо понадобится.

Мы решили обучать читателей C++ на примерах создания игр прежде всего потому, что для многих людей первое знакомство с компьютером связано с играми. А самое главное, это замечательный способ научиться программировать – игры учат отображать интерфейс на экране, обрабатывать команды пользователя и информацию. В конечном итоге они сочетают в себе искусство и науку, проникая в умы творческие и логичные, служат источником визуальных, звуковых и душевных переживаний для программистов и пользователей.

Читая книгу, вы изучите многочисленные приемы программирования, которые применимы не только к C++, но и к программированию в целом. Эти распространенные приемы упростят изучение других языков и создание разнообразных приложений (не только игр).

Структура книги

Сложность материала книги возрастает постепенно – от обычных текстовых программ к играм с полноценной графикой. Новичкам в программировании рекомендуем читать главы в естественной последовательности. Читатели, уже имеющие опыт написания программ, могут бегло пролистать первые шесть глав, которые посвящены основам, и перейти сразу к более сложным темам.

Концептуально книга разбита на четыре части (хотя это деление не соответствует последовательности глав). Первая часть (с главы 1 «Путешествие начинается» по главу 6 «Сложные типы данных») дает базовые знания, необходимые для программирования на C++. Темы в этих главах изложены в определенной последовательности, так что их рекомендуется читать подряд. Так, перед прочтением главы 5 «Боевые качества ООП», скорее всего, придется изучить главу 4 «Пишем функции».

Вторая часть книги (с главы 7 «Градостроение и пространства имен» по главу 11 «Ошибки и обработка исключений») содержит сложные темы C++. Эти главы можно читать в любом порядке.

Третью часть составляют главы с 12 по 14. Здесь читателям предстоит применить все знания, полученные ранее, сначала для разработки Windows-приложений (глава 12 «Программирование для Windows»), затем для работы с DirectX (глава 13 «DirectX») и наконец, для создания потрясающей игры о пиратах с помощью стандартных методов отрасли (глава 14 «Создаем пиратское приключение»).

Четвертая часть содержит приложения с дополнительной информацией, которая будет полезна для читателей.

В каком бы порядке вы ни читали книгу, помните, что существенную часть изучения C++ составляет собственный опыт написания программ. Чем больше программируешь, тем больше приобретаешь навыков в решении задач (очень важное умение в программировании, как станет понятно) и обнаружении ошибок в своем коде. И не исключено, что после многочисленных тренировок вы даже сможете вычислить значение числа π до миллионной цифры после запятой... в уме (хотя авторы книги не дают никаких гарантий этого)!

По мере чтения глав будут встречаться небольшие фрагменты кода, иллюстрирующие понятия, о которых идет речь. В конце каждой главы приводится полноценная игра, демонстрирующая ключевые идеи этой главы, а также резюме главы и набор упражнений, позволяющих испытать приобретенные знания. Мы надеемся, что читатели не поленятся испробовать эти игры и выполнить упражнения, поскольку они существенным образом помогут развить хватку в программировании. Ответы к заданиям приведены в приложении A, а копии всех полноценных программ содержатся на компакт-диске. Но мы настоятельно советуем хотя бы пытаться самостоятельно выполнять задания, не заглядывая в ответы (даже если нужна помощь). Задания достаточно короткие, так что их можно выполнять в компиляторе (опять же, это отличный способ набраться опыта).

Самое необходимое

Изучение программирования – великолепный способ воспользоваться вычислительной мощью компьютера. Но прежде чем вы начнете писать программы, вам понадобится следующее:

- Персональный компьютер с тактовой частотой процессора не менее 75 МГц.
- Операционная система, совместимая с DirectX: Microsoft Windows 95/98/2000, Windows ME/XP.
- Минимум 16 Мбайт оперативной памяти.

- По меньшей мере 125 Мбайт дискового пространства.
- Компилятор (например, CodeWarrior от MetroWerks).
- Устройство для чтения компакт-дисков.
- Знание, время и терпение. Информация, представленная в этой книге, позволит вам эффективно овладеть C++ и компилятором CodeWarrior (да и любым другим).

Специальные обозначения

Помимо полноценных игр и заданий в конце каждой главы книга содержит ряд специальных обозначений:

Примечания. Содержат дополнительную информацию по сложным темам.

Ловушки. Предупреждают об ошибках, которых следует избегать.

Приемы. Содержат советы, облегчающие и совершенствующие программирование.

Истории из жизни

Эти врезки содержат истории о программировании и информацию непосредственно с «передовой».

1

Путешествие начинается

Мысль о программировании может быть пугающей, но волноваться не стоит. Мы написали эту главу так, что вам не придется погружаться в тонкости программирования, чтобы начать писать программы. Глава начинается с рассказа о *CodeWarrior*, компиляторе C++. Затем мы перейдем к основам создания программ, а затем поиграем со строками и числами. С нашей помощью, используя свою изобретательность, вы очень скоро начнете писать собственные программы. Позже они будут постепенно становиться все более сложными, но, чтобы отправиться в путешествие, нужно с чего-то начать. *Ваше* приключение начинается здесь и сейчас! И его началом станут следующие этапы:

- Работа с компилятором CodeWarrior
- Создание кода
- Создание самой первой программы
- Изучение цикла разработки
- Работа с текстом
- Работа с числами

Работа с компилятором CodeWarrior

В этом разделе вы узнаете, как использовать CodeWarrior для создания программы на основе готового шаблона исходного текста. *Исходный текст* – это текст, представляющий собой определенный набор инструкций, которые будут выполняться компьютером. Исходный текст пишется не на русском языке, а на языке программирования. И хотя языков программирования существует великое множество, эта книга научит вас применять C++. Позже в этой главе мы расскажем, как писать исходные тексты программ (вернее, изменять текст, предоставляемый компилятором CodeWarrior).

CodeWarrior облегчает создание программ при помощи *интегрированной среды разработки* (Integrated Development Environment, IDE). Среда разработки является общим графическим интерфейсом для компилятора, навигации в каталогах, изменения настроек, а также *редактора исходных текстов* (то есть окна, в котором происходит редактирование и просмотр текстов программ). Когда мы начинали программировать, то пользовались бесплатным компилятором C++, и все настройки приходилось вводить в приглашении командной строки DOS. На это уходила масса времени. Среда разработки сама позаботится о настройках проектов и файлов, ускоряя и облегчая процесс создания программ.

Мы писали эту книгу, исходя из предположения, что читатели пользуются компилятором CodeWarrior Professional 5.0 (разработанным Metrowerks). Если это не так, ничего страшного. Большая часть сведений из этой и других глав будет полезна независимо от того, какой компилятор используется.

Пора начать наши поиски приключений в мире программирования. В процессе чтения разделов, посвященных созданию проекта в CodeWarrior и написанию кода, пробуйте воспроизводить наши действия на своем компьютере. Практика позволяет привыкнуть к незнакомым вещам.

Создаем новый проект

При первом запуске CodeWarrior выглядит так, как показано на рис. 1.1. Как видите, никакого волшебства (ну разве что совсем немного). CodeWarrior – это просто приложение, такое же как Microsoft Word и Netscape Navigator, лишь с той разницей, что CodeWarrior позволяет создавать другие приложения.

Чтобы создать новый проект C++, имея запущенный CodeWarrior, выполните следующие шаги (помните, что элементы меню, диалоговые окна и параметры могут отличаться для вашего компилятора):

1. В строке главного меню CodeWarrior щелкните по элементу File.
2. В появившемся меню выберите пункт New. Откроется диалоговое окно (рис. 1.2), позволяющее создать практически любой тип приложения.
3. На вкладке Project выберите строку «Win32 C/C++ Application Stationery» (Приложение C/C++ для платформы Win32).
4. В поле имени проекта (Project name) наберите имя **Hello**.
5. Нажмите OK. Откроется диалоговое окно нового проекта (рис. 1.3). Оно позволяет выбрать тип *среды времени выполнения* (*run-time environment*), в которой предполагается использовать готовое приложение. Среда времени выполнения определяется условиями, в которых работает программа. Чаще всего эти условия определяют

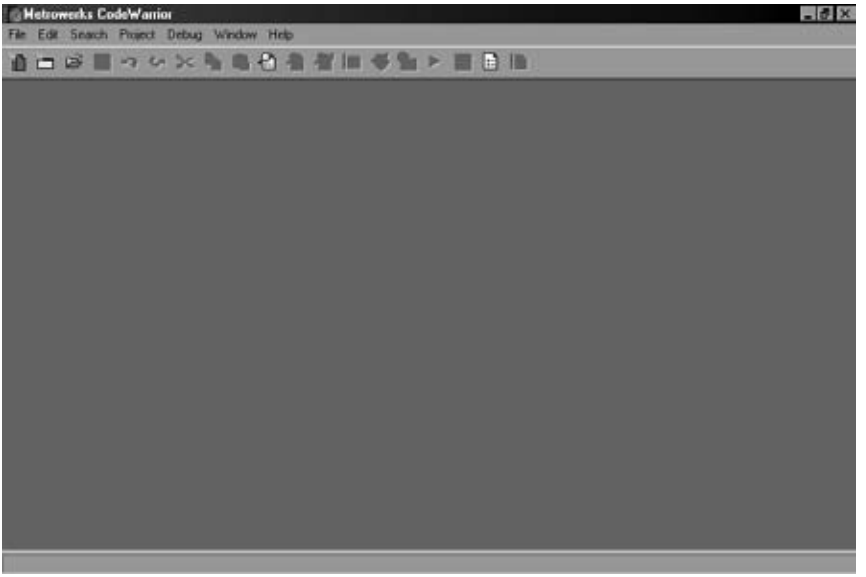


Рис. 1.1. *Так выглядит CodeWarrior при первом запуске*

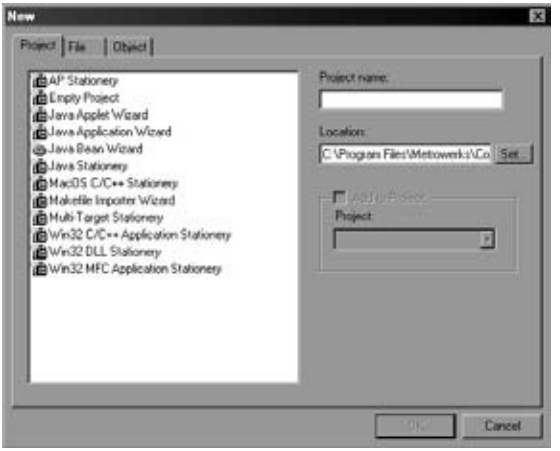


Рис. 1.2. *В окне New выберите тип создаваемого проекта*

лишь операционную систему, для которой будут компилироваться исходные тексты. Например, программы для DOS работают в среде DOS, а приложениям Win32 требуется 32-разрядная среда Windows. Компилятор оптимизирует программу для работы в конкретной среде. В конечном итоге это сокращает размер файлов и повышает скорость работы. Консольные приложения (C++ Console) являются приложениями Windows, работающими в окне, похожем на то, что используется в DOS для выполнения команд. Среда для работы таких приложений является вариантом среды DOS.

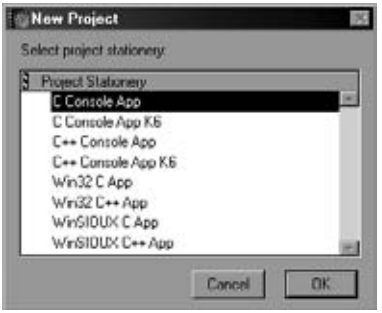


Рис. 1.3. *Диалоговое окно New Project позволяет оптимизировать программы*

6. Выберите из списка в диалоговом окне нового проекта элемент C++ Console App и нажмите OK. Как только CodeWarrior закончит создавать настройки для выбранного проекта, откроется новое окно. Это окно носит имя проекта – hello.mcp. Расширение .mcp связано с файлами проектов CodeWarrior. Файл проекта хранит все настройки проекта, а кроме того содержит перечень всех исходных файлов, составляющих проект. *Исходный файл* имеет текстовый формат и предназначен для хранения исходных текстов. Исходные файлы имеют расширение .cpp.

7. Чтобы раскрыть папку, щелкните по расположенному рядом значку +. В раскрывшейся папке проекта находится файл hello.cpp (расширение .cpp связано с исходными файлами C++).

8. Чтобы открыть файл, дважды щелкните по его имени. На экране появится окно, показанное на рис. 1.4.

Рис. 1.4. *В этом редакторе вы будете работать с исходными текстами*

Итак, вы добрались до сердца подземелий. В следующем разделе мы расскажем о загадочных письменах на рис. 1.4.

Истории из жизни

В начале 1980 года Бьерн Страуструп (Bjarne Stroustrup), работавший в AT&T Bell Laboratories, начал трудиться над языком C++. C++ официально получил свое имя в конце 1983 года, причем имя, подчеркивающее родство с предшественником, языком C. В октябре 1985 года появилась первая коммерческая версия языка и первое издание книги Страуструпа «Язык программирования C++» (The C++ Programming Language).[1]

В восьмидесятые годы язык C++ проходил доработку и в итоге приобрел индивидуальность, сохранив, по большей части, совместимость с языком C и его наиболее важные характеристики. В C++ по-прежнему доступны классические методологии структурного программирования, но дополнительно существует функциональность *объектно-ориентированного программирования* (или ООП; более подробно тема рассмотрена в главе 5 «Боевые качества ООП»). Своим происхождением C++ обязан и другим языкам – BCPL, Simula67, Algol68, Ada, Clu и ML, – из которых он позаимствовал некоторые свои достоинства. К счастью, C++ предоставляет все преимущества этих языков, а потому нет надобности изучать их.

В 1990 году комитет ANSI (*American National Standards Institute*, Американский национальный институт стандартов) под названием *X3J16* начал разработку стандарта C++. До публикации его окончательного варианта в ноябре 1997 года язык C++ успел получить широкое распространение и теперь является самым популярным языком для разработки приложений.

---

[1] Б. Страуструп, «Язык программирования C++». – Пер. с англ. – СПб: Невский Диалект, 2001 г.

Что такое исходный текст?

Программирование заключается в передаче компьютеру инструкций, заключенных в исходном коде. Синтаксис исходного кода зависит от того, какой язык программирования используется; в нашем случае это синтаксис языка C++. (*Синтаксис* – это набор правил, определяющих структуру языка.)

Почему нельзя на русском языке объяснить компьютеру, что следует делать? Дело в том, что русский язык очень сложный, и компьютер не сможет понять, что вы пытаетесь сказать. C++ является в некотором роде упрощенным вариантом английского языка, который способен понять CodeWarrior. В следующем разделе вы узнаете, что для непосредственного понимания компьютером слишком сложен даже язык C++. CodeWarrior дол-

*жен перевести код на C++ в **машинный код**. А пока просто сфокусируйтесь на создании исходного кода и существующих для этого правилах.*

C++ очень четко определяет структуру исходных текстов. К примеру, знаки препинания и порядок следования очень важны. Даже регистр символов важен, поскольку C++ *чувствителен к регистру*. Это означает, что компилятор различает прописные и строчные буквы (буквы *K* и *k* с его точки зрения являются различными).

Создание программы начинается с набора кода в редакторе исходных текстов. Затем компилятор преобразует исходный код в язык, на котором говорит компьютер (машинный код). Компилятор и редактор интегрированы в среду CodeWarrior IDE. Компилятор имеет первостепенное значение, и поэтому интегрированные среды разработки часто так и называют – *компиляторами*.

Каждая из строк кода отвечает за определенное действие, примерно как ингредиенты в рецепте. Компилятор разбивает строки кода на инструкции, которые называются *командами*. Команда является атомарной инструкцией для компьютера.

Ранее мы уже говорили, что CodeWarrior представляет собой интегрированную среду разработки и выполняет за вас многие задачи. Текст, который вы видите в окне hello.cpp, свидетельствует об этом. Этот текст – автоматически созданный код, который послужит основой для любой программы. О программе в целом можно думать как о мосте, который вы строите. Сгенерированные строки кода являются опорами. Опоры нужны любому мосту, но мост, состоящий из одних опор, бесполезен.

Взгляните на сгенерированный код: эта программа печатает на экране строку `This is a test` («Это проверка»):

```
#include <iostream>
using namespace std;       //introduces namespace std
int main( void )
{
     cout << "This is a test" ;
     return 0;
}
```

Эти строки содержат некоторые из существующих инструкций, которые могут передаваться компьютеру. Код можно изменять и совершенствовать (либо просто стереть и начать с нуля) с целью создания собственной программы. Но пока что будем считать, что это программа, которую вы написали. В этом случае для создания работающей программы необходимо скомпилировать и выполнить код.

Компиляция

Чтобы запустить программу, следует преобразовать написанный на C++ код в язык, который понимает компьютер. Именно на этом этапе в игру вступает компилятор. Вообразите, что вы эльф, а компьютер – гном. Чтобы компьютер понял ваши указания, необходимо преодолеть языковой барьер. Нужен переводчик, который говорит на языке гномов и на языке эльфов. В мире компьютеров таким переводчиком является *компилятор*. Как мы уже говорили, компилятор преобразует язык программирования в *машинный код*, на котором говорит компьютер. Однако перевод возможен только в одну сторону: компилятор не умеет переводить машинные коды в исходный текст.

Чтобы произвести компиляцию в CodeWarrior, выполните следующие шаги:

1. Из главного меню выберите Project (Проект) и затем пункт Compile (Компилировать). Откроется окно, озаглавленное Building Hello.mcp. (Этот шаг иногда занимает до нескольких минут, поэтому необходимо дождаться завершения работы компилятора.)

   Когда это окно активно, происходит преобразование файла в машинный код и проверка того, что автор программы не нарушил правила языка C++. Если вы изменили текст, созданный средой CodeWarrior, может появиться сообщение об ошибке. Если вы не меняли текст либо внесенные изменения не содержат ошибок, компилятор закроет окно Building Hello.mcp после завершения перевода программы в машинный код. Программа готова к запуску.

2. Из главного меню Project (Проект) выберите пункт Run (Выполнить).

   Откроется окно, в котором отображается вывод программы (рис. 1.5). Вы должны увидеть надпись: `This is a test`.

3. Нажмите любую клавишу, чтобы закрыть окно.

Рис. 1.5. *Вот что вы увидите после компиляции и выполнения программы*

Пишем первую программу

Постойте! Не торопитесь выключать компьютер, все будет гораздо интереснее. Сейчас вы напишете свою первую программу. Она будет печатать на экране строку `Здравствуй, мир!`. Этот проект поможет вам понять язык C++ и механизмы его работы. Но прежде следует выяснить, что уже сделал CodeWarrior.

Начинаем с типовой программы

В начале нового проекта CodeWarrior самостоятельно создает некий код. Этот код является основой для развития программы. Его можно удалить полностью или частично в зависимости от потребностей, но в большинстве случаев сгенерированный код помогает вам начать. Этот код называется *типовой программой*. Он должен выглядеть так:

```
#include <iostream>
using namespace std;       //introduces namespace std
int main( void )
{
     cout << "This is a test" ;
     return 0;
}
```

Рассмотрим строки по очереди. Первая строка сообщает компилятору, что мы будем пользоваться командами из библиотеки iostream. Это существующая библиотека, которая поставляется в составе CodeWarrior и всех остальных компиляторов C++. Она является частью стандартной библиотеки C++. Прежде чем продолжить, нам придется объяснить несколько дополнительных терминов:

- **Команды.** Общее название для строк кода, набираемых в редакторе исходных текстов и представляющих собой инструкции для компьютера. Комментарии не являются командами.
- **Директивы включения.** Это строки, которые начинаются с инструкции `#include`. Они нужны для включения исходных файлов, созданных вами или другими программистами, в текст программы. За инструкцией `#include` следует имя файла, заключенное в символы < и > (для стандартных библиотечных файлов) либо в двойные кавычки (для всех остальных файлов). По странному стечению обстоятельств у файлов стандартной библиотеки нет расширений. Именно поэтому включается iostream, а не iostream.cpp. Однако у большинства включаемых файлов расширения есть, и про них следует помнить. Обычно директивы включения расположены в начале файла. (Пусть вас не беспокоят принципы работы директив включения. На данном этапе достаточно знать, что они используются для включения в программы внешнего кода.)
- **Библиотеки.** Законченные фрагменты кода, для удобства включенные в язык C++. (С различными компиляторами поставляются раз-

личные наборы библиотек. Найти нужную библиотеку можно в Интернете. Один из замечательных ресурсов по бесплатным библиотекам C++ расположен по адресу *http://www.cnet.com*.) Грубо говоря, библиотеки – это скомпилированные включаемые файлы. К примеру, случайное число можно получить с помощью генератора случайных чисел, функции `rand`, поскольку она входит в состав библиотеки cstdlib (C Standard Library, стандартная библиотека C). Для этого необходимо поместить строку `#include <cstdlib>` в начало программы. Эта инструкция подсказывает компилятору, какая библиотека используется. Включаемые файлы библиотек обычно имеют расширение .h (header, файл заголовка), за исключением файлов стандартных библиотек C++, у которых нет расширения.

Вторая строка приведенного кода, `using namespace std`, заставляет стандартные библиотеки работать так, как нам нужно. Подробности, связанные с этой строкой, входят в состав более сложных тем, которые мы изучим в главе 7 «Градостроение и пространства имен». Вторая половина строки, `//introduces namespace std`, является *комментарием*. Комментарий не влияет на выполнение программы. Пара прямых слэшей (//) предписывает компилятору игнорировать оставшуюся часть строки. Назначение комментариев – делать код более понятным. Комментарии можно писать двумя способами. Однострочные комментарии (как предыдущий) занимают только одну строку. Все символы после // в этой строке игнорируются:

```
// В моей армии один человек
```

Второй способ позволяет растянуть одну фразу либо длинный комментарий на произвольное количество строк:

```
/* драконы правят миром */
```

или

```
/* драконы правят
    миром */
```

И хотя это обычно снижает читаемость, можно размещать многострочные комментарии практически в любой точке кода, как показано здесь:

```
using namespace /*introduces namespace std*/ std;
```

Однако не стоит пользоваться возможностью только потому, что она существует. Такой стиль использования комментариев может быстро сделать код совершенно нечитаемым. Основное правило таково: пользуйтесь комментариями, чтобы делать код более понятным. Если комментарий не приносит пользы, удаляйте его.

Повторимся: содержание комментариев не влияет на код программы. Применяйте их только для пояснения сложных или крупных частей программы, используя обычный русский язык. Компилятор игнорирует комментарии, когда переводит программу в машинный код.

Истории из жизни

Комментарии полезны, поскольку облегчают понимание кода не только для других, но и для самих авторов.

Некоторое время назад мы занимались воскрешением классической игры для Nintendo, которая называлась «Iron Tank» (Железный танк). Через некоторое время после начала проекта у нас было более 30 страниц кода. Затем нам пришлось на некоторое время прервать разработку. Когда мы снова вернулись к этой игре, то потратили почти неделю, пытаясь разобраться, что делает код. Если бы мы сразу озаботились написанием комментариев, эта работа заняла бы максимум один день.

Следующая строка, `int main( void )`, отмечает начало функции `main`. Большая часть кода содержится в функции `main`. В главе 4 «Пишем функции» мы расскажем о том, как помещать код в другие функции, но на данном этапе практически каждая написанная строка кода будет располагаться в теле главной функции. Функция `main` начинается символом `{` и заканчивается символом `}`. Внутри фигурных скобок может располагаться практически любая строка кода. В каждой программе должна быть функция `main` (и только одна).

Следующая строка отмечает начало кода функции `main` открывающейся фигурной скобкой (`{`).

Следующая строка печатает сообщение `This is a test`. Слово `cout` позволяет отобразить текст на экране. И поскольку это слово не встроено в язык, а является частью библиотеки iostream, чтобы воспользоваться им, необходимо включить эту библиотеку в код. Подробнее мы рассмотрим оператор `cout` в разделе «Работа с текстом» позже в этой главе. (*Операторы* – это атомарные мысли или команды; считайте их эквивалентами предложений. Обычно оператор заканчивается точкой с запятой.)

`return 0;` сообщает компьютеру, что все задачи в функции `main` выполнены и следует завершить ее работу.

Закрывающаяся фигурная скобка в последней строке (`}`) сообщает компилятору, что в функции `main` больше нет строк кода. `return 0;` – это исполняемый оператор, который завершает работу функции, а закрывающаяся скобка показывает, что в функции больше нет исполняемых операторов.

Не забывайте о точках с запятыми – они являются частью кода. Большинство операторов *должны* заканчиваться точкой с запятой. В главе 3 «Принимайте командование: управляющие операторы» мы расскажем об операторах, которые не требуют точки с запятой, но до тех пор каждый из встретившихся нам операторов должен завершать-

ся этим символом. Точное правило звучит так: каждый исполняемый оператор должен заканчиваться точкой с запятой. Она отмечает конец оператора, а не конец строки, поскольку в одной строке может присутствовать несколько операторов.

Теперь, когда первое препятствие преодолено, ваше приключение станет еще проще.

Здороваемся с миром

Итак, вы готовы создать свою самую первую программу, которую мы предлагаем назвать «Здравствуй, мир!» (хотя вы можете выбрать любое другое название). Программа будет выводить на экран сообщение `Здравствуй, мир!`. В процессе создания этой программы вы научитесь редактировать исходный текст и поближе познакомитесь с работой кода, сгенерированного средой CodeWarrior. Прежде всего, замените строку

```
cout << "This is a test";
```

строкой

```
cout << "Здравствуй, мир!";
```

Если теперь скомпилировать и выполнить программу, мы увидим на экране сообщение `Здравствуй, мир!`.

Ниже приведен законченный код. (Обратите внимание, мы добавили в начало программы комментарий, который содержит название программы и имя автора. Такие комментарии не являются обязательными, но лишними они тоже не будут.)

```
//1.1 - Здравствуй, мир - Дирк Хенкеманс - Premier Press
#include <iostream>
using namespace std;     //introduces namespace std
int main( void )
{
    cout << "Здравствуй, мир!" ;
    return 0;
}
```

Повторите действия, которые уже выполняли для компиляции кода, сгенерированного CodeWarrior, либо (что работает в большинстве компиляторов) просто нажмите <F5>, чтобы скомпилировать и выполнить программу. Программа выводит сообщение `Здравствуй, мир!`.

Цикл разработки

Цикл разработки – это процесс, через который необходимо пройти при создания программы. Рано или поздно вы поймете, что процесс этот незатейлив и довольно прост. Блок-схема на рис. 1.6 иллюстрирует цикл разработки.

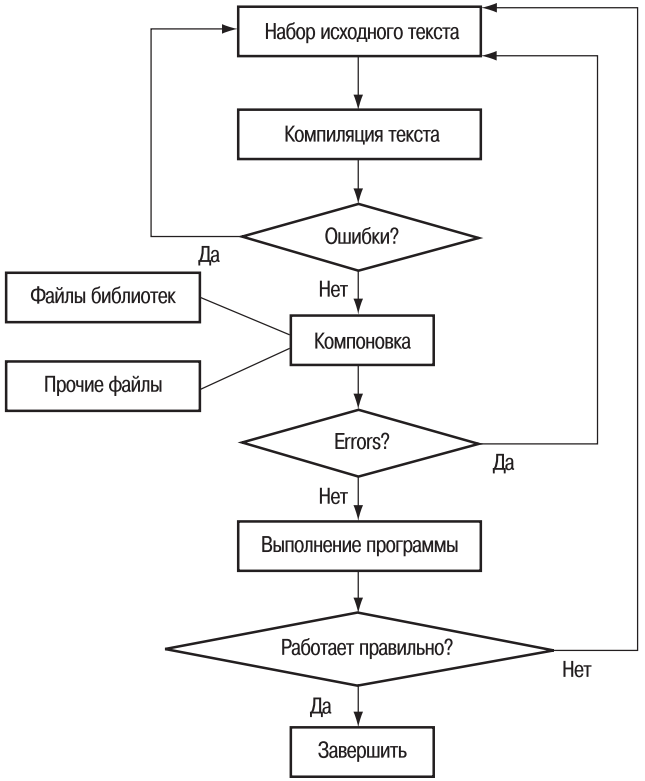


Рис. 1.6. *Схема процесса создания работающей программы*

В общем случае создание программы включает следующие шаги:

1. Набор кода в редакторе исходных текстов (CodeWarrior или другой редактор). В предыдущем разделе «Пишем первую программу» вы немного этим занимались, а до конца книги еще вдоволь попрактикуетесь.

2. Компиляция кода. Если обнаружены ошибки, следует вернуться к предыдущему шагу и исправить их. Пошаговые инструкции по компиляции кода в CodeWarrior даны выше по тексту в разделе «Компиляция».

3. *Компоновка* кода. Это процесс проверки его работоспособности со всеми включенными файлами. Если возникнет ошибка, следует вернуться к шагу 1 и исправить ее. CodeWarrior выполняет компоновку автоматически, и она вас не должна интересовать, пока не возникнет ошибка.

4. Тестирование программы. Следует убедиться, что программа работает правильно. Другими словами, тестирование предназначено для обнаружения семантических ошибок. *Семантические ошибки*

связаны с результатами, которые показывает программа. Если программа компилируется и выполняется, но не делает то, что должна делать, речь обычно идет о семантической ошибке. Допустим, вы написали программу для вывода на экран слова «Привет», а вместо этого выводится слово «Пока». Это семантическая ошибка, которую следует исправить. Тем не менее, наличие семантических ошибок не мешает компиляции кода.

Если программа успешно прошла через все шаги, цикл разработки завершен. Тестирование программ и исправление ошибок описано в главе 5.

Работа с текстом

До появления графики основой всех программ был текст. Текстовые приключения и текстовые электронные доски объявлений (BBS) – вот наши первые опыты программирования. Текст по-прежнему является очень важной составляющей программирования. В этом разделе вы узнаете, как создавать и хранить его.

Текст – самое простое средство вывода результатов работы программ. В первых главах книги используется только текст, поскольку в большинстве случаев его достаточно для отображения выходных данных. Этот подход также избавляет нас от необходимости вдаваться в сложности, связанные с графикой, и отложить их до более поздних глав.

В мире компьютеров у текста есть техническое название – *строка* (*string*). Так, в нашей первой программе текст `Здравствуй, мир!` является строкой.

Компоновка строк

Строка – это набор символов. Обычно мы считаем *символом* знак, который можно набрать с клавиатуры (включая пробелы). Компьютер считает строку, например `"За короля!!!"`, последовательностью символов, каждый из которых занимает одну позицию (`'3'` – первый символ, `'a'` – второй и т. д.). Строки заключаются в двойные кавычки, а отдельные символы – в одинарные. Скажем, `"a"`, `" "` (пробел), `"4"` и `"%"` – это строки, и каждая состоит из одного символа. Однако `'a'`, `' '`, `'4'` и `'%'` – это отдельные символы. За редким исключением символы можно использовать везде, где используются строки.

Но символы не так просты, как кажется. Не все символы можно набрать при помощи клавиатуры. Существует 256 различных символов.

Доступ ко многим другим символам можно получить, удерживая клавишу <Alt> и набирая их численные значения. К примеру, численным значением буквы *A* является 65. В приложении C «Стандартная таблица символов ASCII» содержится справочник по стандартизированным символам и их численным значениям.

Очень важно понимать разницу между строками и символами. Строки состоят из символов, но значительно отличаются от них. Кроме того, символ можно легко преобразовать в строку, но он не является строкой.

Тема может показаться сложной, но на деле строки очень легко создавать. Ниже приведены примеры строк. Как видите, строка – это просто текст, заключенный в двойные кавычки.

```
"За короля!!!"
"Передайте мне мой меч."
"Кто положил мой посох на повозку?"
```

Хранение строк

Строки можно хранить в компьютере, чтобы их не приходилось набирать многократно. Точнее, строки хранятся в *памяти* (о памяти мы подробно поговорим в главе 2 «Продолжаем погружение: переменные»). Код для работы со строками хранится в подходящим образом названной *библиотеке строк*. Чтобы хранить строки, необходимо включить библиотеку в текст программы. Это можно сделать, добавив в начало исходного текста такую директиву:

```
#include <string>
```

Когда вы решаете сохранить строку в памяти, то должны придумать для нее имя, чтобы в будущем можно было объяснить компьютеру, о какой из строк идет речь. К примеру, мы сохраним строку `"Дракон приближается"` в памяти и назовем ее `yell`.

Существует и небольшая сложность. Компьютер умеет хранить в памяти различную информацию, не только строки, и различного рода информация хранится по-разному. Число хранится иначе, чем строка. Поэтому необходимо указать компьютеру, какого рода информацию мы намереваемся хранить. Вот что мы должны указать:

- Тип информации, которую необходимо хранить (в нашем случае – строка `string`)
- Символы, составляющие строку
- Имя строки

И хотя C++ не обеспечивает возможности хранить строки, такую возможность обеспечивает стандартная библиотека. Чтобы воспользоваться ею, необходимо включить в текст программы библиотеку `<string>`.

После включения библиотеки строку можно сохранить так:

```
string yell = "Дракон приближается";
```

Прежде всего указывается тип хранимой информации: строка. Затем идет имя строки: `yell`. Знак равенства сообщает компьютеру, что строка `"Дракон приближается"` должна быть сохранена в `yell`. Последний символ, точка с запятой, сообщает компьютеру, что команда закончена и за ней начинается следующая.

Вывод строк: cout

Теперь вы знаете, как записать строку в память, и вам наверняка захочется ее отобразить. В этом разделе мы расскажем, как печатать строки разнообразными способами.

Вывод строк производится с помощью команды `cout`. Команда `cout` принадлежит файлу iostream стандартной библиотеки C++. Вот команда `cout`, которую мы встречали раньше:

```
cout<< "Здравствуй, мир!";
```

При помощи `cout` можно отобразить любую строку:

```
cout<< "Вперед, на подвиги!";
```

Во-первых, необходимо набрать команду `cout`, затем два знака «меньше» (`<<`), а затем строку для печати и за ней точку с запятой (которая, как вы, вероятно, помните, сообщает компилятору, что команда закончилась).

Вывод нескольких строк: cout

Вы овладели основами `cout`, но на этом разговор о `cout` и строках не заканчивается. Еще эта команда позволяет, например, выводить на печать несколько строк одновременно. Предположим, мы хотим отобразить два слова, но они хранятся в различных строках. С учетом полученных знаний мы могли бы написать примерно такой код:

```
cout<<"Красный";
cout<<" Дракон";
```

Вывод:

```
Красный Дракон
```

Но есть и более простой способ. Можно отобразить любое число строк, разместив их рядом и разделив парами символов `<<`. Вот эквивалент предыдущего фрагмента кода:

```
cout<<"Красный"<<" Дракон";
```

Вывод:

```
Красный Дракон
```

Вывод в обоих случаях одинаковый, но второй фрагмент кода немного проще. Структура языка C++ позволяет записывать команду в несколько строк. Последний фрагмент кода и приведенный ниже эквивалентны:

```
cout
<<"Красный"
<<" Дракон";
```

Вывод:

```
Красный Дракон
```

Разумеется, пример тривиальный, поскольку в данном случае мы могли бы воспользоваться единственной строкой (`"Красный Дракон"`).

Рекомендуется структурировать текст программы так, чтобы его было легко читать. В начале каждой фразы следует использовать отдельную команду `cout`. К примеру, чтобы напечатать предложение о драконах и предложение об эльфах, следует поместить всю информацию о драконах в одну команду `cout`, а всю информацию об эльфах – во вторую команду. Таким образом, код будет более организованным и легким для восприятия.

Специальные символы

Некоторые символы, такие как двойные кавычки (`"`) и разрыв строки, невозможно выразить в строке. Если включить двойные кавычки в строку, компилятор посчитает этот символ признаком конца этой строки. К примеру, чтобы создать строку с цитатой, мы могли бы написать:

```
"Он сказал: "Это цитата"."
```

Но компилятор интерпретирует нашу строку как две самостоятельные строки `"Он сказал: "` и `"."`, а поскольку слова `Это цитата` не будут сочтены строкой, мы получим ошибку синтаксиса. Сходная проблема связана с разрывом строк. Текст строки должен располагаться в одной строке программы, и разрыв строки в любом месте также приведет к возникновению ошибки синтаксиса. К счастью, существует решение всех этих проблем.

Речь идет о специальных символах. *Специальные символы* (или метасимволы, escape-последовательности) используются для представления символов, которые не могут быть явно включены в текст строки. Специальный символ является комбинацией обратного слэша (`\`) и конкретного символа. К примеру, специальным символом для двойных кавычек будет `\"`. Вот так это выглядит в коде:

```
"Он сказал: \"Это и впрямь цитата\"."
```

Такая строка приведет к получению желаемого результата. Если вывести строку на печать, мы увидим на экране текст `Он сказал: "Это и впрямь цитата".`. Существуют и другие специальные символы, а самые важные из них перечислены в табл. 1.1.

Мы советуем поближе познакомиться с большинством из этих специальных символов, потому что они бывают полезными, а некоторые из них, скажем, разрыв строки, являются жизненно необходимыми.

Таблица 1.1. Специальные символы

Название	Символ
Новая строка	\n
Горизонтальная табуляция	\t
Забой (backspace)	\b
Звуковой сигнал	\a
Обратный слэш	\\
Знак вопроса	\?
Одинарная кавычка	\'
Двойная кавычка	\"

Вывод хранимых строк

До этого момента мы выводили на печать непосредственно символьные строки. Теперь мы готовы к отображению хранимых строк. Чтобы отобразить хранимую строку, следует использовать вместо символьной строки имя хранимой строки. Напомним определение строки, с которым мы уже встречались:

```
string yell = "Приближается дракон";
```

Чтобы отобразить строку, воспользуемся ее именем, `yell`, а не хранимым текстом:

```
cout<<yell.c_str();
```

Эта строка делает то же, что и следующая, которую мы использовали в разделе «Хранение строк».

```
cout<<"Приближается дракон";
```

Не беспокойтесь пока о суффиксе `.c_str()` после имени `yell`: это просто «волшебный» код, позволяющий правильно отобразить строку с текстом. Подробнее о волшебном коде мы расскажем в главе 6 «Сложные типы данных».

Программа «Глашатай»

Вы узнали, как создавать строки и выводить их на экран, пользуясь `cout`. Настало время испытать знания, создав программу «Глашатай».

Представьте, что к деревне приближается дракон и необходимо прокричать (yell) предупреждение всем жителям. Если вы прокричите предупреждение в четыре раза быстрее обычного, деревня будет спасена. Очевидно, это работа для наших героев... хранимых строк и команды `cout` – только они смогут спасти деревню. Таким образом, ваше задание заключается в создании программы, которая выводит текст предупреждения четыре раза подряд.

Ниже приводится один из вариантов готовой программы, но мы предлагаем обращаться к нему только в самом крайнем случае – если вы будете испытывать сложности с созданием собственной программы:

```
//1.2 - Программа "Глашатай" - Дирк Хенкеманс - Premier Press
#include<iostream>
#include<string>
using namespace std;      //introduces namespace std

string yell = "Приближается дракон, спасайтесь!!!";

int main(void)
{
    cout<< yell.c_str() <<endl
        << yell.c_str() <<endl
        << yell.c_str() <<endl
        << yell.c_str() <<endl;

    return 0;
}
```

Вывод:

```
Приближается дракон, спасайтесь!!!
Приближается дракон, спасайтесь!!!
Приближается дракон, спасайтесь!!!
Приближается дракон, спасайтесь!!!
```

Применение cin

При разработке программ часто бывает необходимо предоставить пользователям возможность вводить и хранить информацию. Например, набирая свои имена, пользователи должны иметь возможность обращаться к ним впоследствии. Следовательно, необходим способ получения и применения информации от пользователя в процессе работы программы (на этапе ее выполнения), а не в процессе написания (разработки) программы, поскольку невозможно предсказать ввод пользователей. В этом разделе мы расскажем, как получать ввод пользователя и сохранять его для последующего применения.

Сохраняем строки с помощью cin

Сохранение строк с помощью `cin` во многом похоже на то, как мы сохраняли строки раньше, с одним исключением: сначала объявляется имя строки, а значение присваивается позже. Для присвоения значения используется объект `cin`. (*Объект* – это конструктивный элемент программирования, представляющий собой сущность или понятие. Объект `cin` представляет клавиатуру или иное устройство ввода, а объект `cout` – экран или иное устройство вывода. (Подробнее мы расскажем об объектах в главе 5.) Вот пример сохранения строки, введенной пользователем:

```
string name;
cin>>name;
```

Обратите внимание, применение `cin` во многом сходно с применением `cout`. Знаки «меньше» сменились на знаки «больше». Они означают, что компьютер читает данные, а не печатает их.

Вот пример полноценной программы, в которой используется `cin`:

```
//1.3 - Программа "Приветствие" - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
using namespace std;      //introduces namespace std
string name = ""; //  "" означает пустую строку
int main( void )
{
    cout<< "Как тебя зовут?";
    cin>>name;
    cout <<endl<<  "Привет, " << name.c_str() ;
    return 0;
}
```

Вывод (ввод пользователя выделен):

```
Как тебя зовут?
Джеки
Привет, Джеки
```

В некоторых ситуациях и для отдельных компиляторов система Windows закрывает окно программы раньше, чем будет отображена последняя информация. Действительно, как только система Windows поймет, что осталось лишь отобразить текст, после его отображения она немедленно закрывает окно. Чтобы увидеть окончание вывода программы, можно добавить в конец программы оператор `cin` *(что мы и сделаем, когда чуть позже будем писать игру «Пираты и мушкетеры»). Windows не закроет окно, пока вы не нажмете клавишу <Enter> по завершении программы.*

В пятой строке приведенного кода `string name` сообщает компьютеру, что следует выделить в памяти место для хранения строки по имени `name`. Помните, что для работы со строками необходимо включить библиотеку `string`.

В строке 8 оператор `cout<< "Как тебя зовут?";` отображает для пользователя приглашение, в котором предлагается набрать имя.

В строке 9 оператор `cin<<name;` предписывает компьютеру остановиться, чтобы пользователь мог набрать информацию. Когда пользователь нажимает клавишу <Enter>, компьютер делает весь текст, набранный до нажатия <Enter>, значением строки `name`.

В строке 10 оператор `cout<< endl << "Привет, " << name.c_str();` начинается с указания компьютеру начать вывод с новой строки. Затем происходит вывод строки `"Привет, "` и имени пользователя. Если в строке 9 пользователь ввел `Джо` или `Джейн` в качестве своего имени, строка 10

приведет к отображению `Привет, Джо` или `Привет, Джейн`. Между словом `Привет,` и именем присутствует пробел, потому что он включен в конец строки `"Привет, "` (до закрывающей двойной кавычки).

И хотя о тексте можно еще многое рассказать, базовую информацию по использованию текста вы уже получили, и теперь мы обратим свое внимание на числа.

Работа с числами

Компьютеры работают только благодаря числам. Даже текст, с которым мы работали ранее, состоит из них (если прибавить единицу к букве *B*, мы получим букву *C*). Числа являются основой всего, что происходит в компьютере, и хорошее их понимание просто необходимо. В этом разделе вы узнаете об основах математики, операторе взятия остатка и о целых числах (дополнительная информация по целым числам содержится в главе 2).

Знакомьтесь, целые числа

Компьютеры хранят информацию разнообразными способами. Тем не менее, мы остановимся пока лишь на базовых понятиях целых чисел и их применении. *Целые числа* – это все недробные числа, как положительные, так и отрицательные, включая нуль. Числа 5, 0 и –100 являются целыми, а число 0,5 – нет. Если вы попытаетесь сохранить дробное число в качестве целого, компьютер удалит остаток, то есть все, что находится после десятичной запятой.

Выполнение действий с помощью операторов

В общем случае *оператор* – это любой символ или пара символов (скажем `<=`), а в некоторых случаях даже термин вроде `sizeof()`, который предписывает компьютеру выполнить определенное действие. Для выполнения сложения, вычитания, умножения и деления применяются операторы. К примеру, когда требуется, чтобы компьютер сложил два числа, следует использовать оператор сложения (`+`), что, вообще говоря, совершенно естественно. Все мы знаем, что 2 + 2 = 4. Вот как можно выполнить ту же операцию в коде C++:

```
cout<<2 + 2;
```

Эта строка выводит на экран цифру `4`.

Четыре основных оператора абсолютно прозрачны – они делают именно то, что вы, вероятно, предполагаете, но потратьте несколько секунд на изучение символов, соответствующих каждому из них:

Сложение `+`

Вычитание `-`

Умножение	*
Деление	/

Как и в математике, эти операторы выполняются не в порядке следования. Операторы умножения и деления выполняются до операторов сложения и вычитания. Например:

```
1 + 3 * 2
```

Сначала вычисляется `3 * 2`, затем добавляется цифра `1`, и в результате получается число `7`.

Приоритет вычислений можно изменять с помощью простых скобок. Если добавить в предыдущую формулу скобки, как показано здесь:

```
(1 + 3) * 2
```

то сначала будет вычислена сумма `1 + 3` с результатом `4`, и этот результат будет умножен на `2`. Окончательно получаем `8`.

Используйте скобки в больших количествах. Это значительно облегчает отладку. Основное правило таково: если формула может выиграть от применения скобок, используйте их. Следование этому правилу сделает код более понятным и легким для восприятия.

Оператор взятия остатка

Помните, в начальных классах вы выполняли письменное деление столбиком и всегда получали целочисленные ответы? Примерно в то же время вы начали работать с остатками, потому что еще не изучали десятичные дроби. Иногда очень полезно знать остаток, полученный при делении числа. Эту информацию можно извлечь с помощью оператора *взятия остатка*. Он возвращает остаток от деления x на y (`x % y`). Чтобы найти остаток от деления 5 на 2, можно воспользоваться таким кодом:

```
5 % 2
```

Эта строка возвращает цифру `1`. Вот еще один простой пример. Представьте, что пять пиратов пытаются поделить 16 сверкающих золотых монет. Пиратам необходимо выяснить, можно ли разделить сокровище поровну или придется устраивать большую пьяную драку, что вплотную подводит нас к следующей игре. (Впрочем, надо заметить, что пираты в любом случае устроят пьяную драку.)

Пишем игру «Пираты и мушкетеры»

Пришла пора испытать приобретенные навыки. Эта программа доступна на компакт-диске, но мы настоятельно рекомендуем написать ее на компьютере самостоятельно. Это будет первая *полная* програм-

ма, которая позволит проверить приобретенные знания о числах и тексте. Удачной потасовки!

```
//1.4 - Игра "Пираты и мушкетеры" - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
using namespace std;      //introduces namespace std

int main( void )
//расскажет историю о пиратах
{
     int buddies;
     int afterBattle;
     string exit;

     cout<< "Ты пират, и разгуливаешь"
          << " по кишащему преступниками " << endl
          << "городу Гаване  (год 1789).  "
          << "Сколько с тобой  "<<endl
          <<"дружков-пиратов? (побольше)"<<endl;
     //записываем число спутников игрока
     cin>>buddies;
     //вычисляем число пиратов, выживших после схватки.
     afterBattle = 1 + buddies - 10;
     cout<< "Внезапно из ближайшей таверны "
          << "выбегают 10 мушкетеров и " <<endl
          << "обнажают свои шпаги. "
          << "10 мушкетеров и 10 пиратов погибают в " <<endl
          << "схватке. Осталось лишь "
          <<(buddies + 1 - 10)<< " пиратов." <<endl
          <<endl;
    cout<< "Состояние убитых насчитывает 107 золотых монет. Это по "
          <<(107 / afterBattle)
           << " золотых монет на каждого." <<endl;
     cout<< "Пираты устраивают большую пьяную драку из-за оставшихся "
          <<(107 % afterBattle)<< " монет.";
     //делаем паузу, чтобы игрок мог оценить результаты
     cin>>exit;

     return 0;
}
```

Резюме

Самый простой способ выводить текст и числа на экран, а также принимать ввод пользователя – воспользоваться библиотекой iostream. Для обработки ввода пользователя можно применять `cin`, а для вывода информации – `cout`. Чтобы воспользоваться ими, необходимо включить библиотеку `<iostream>` в начале программы. Можно отображать целые числа и сохранять их для последующего использования. Хранение данных позволяет сокращать длину программ и делать их более

эффективными. Кроме того, помните, что в одном операторе `cout` можно выводить сразу несколько строк и целых чисел, но при этом их следует разделять парами символов <<. Вот и все на сегодня; приключение продолжится в главе 2.

Задания

1. Напишите программу, которая выводит изображение домика, похожее на то, что представлено на рис. 1.7.

2. Что выводит следующая программа?

```
#include <iostream>
using namespace std;       //introduces namespace std
int x = 25;
string str2 = "Это проверка";

int main( void )
{
    cout<<"Проверка"<<1<<2<<"3";
    cout<<25 %7<<endl<<str2.c_str();
    return 0;
}
```

3. Напишите программу, которая запрашивает у пользователя его имя, приветствует его, затем запрашивает два числа и отображает их сумму.
4. Что произойдет, если сохранить 10,3 в качестве целого числа? А если 0,6? Можно ли сохранить число –101,8 в качестве целого?
5. Напишите код, умножающий исходное число на 2, если оно принадлежит интервалу от 1 до 100 (включительно) и делится нацело на 3; в противном случае умножает на три, если число принадлежит интервалу от 1 до 100, но не делится нацело на три; и наконец, умножает число на остаток от его деления на 100, если число не принадлежит интервалу от 1 до 100. (Подсказка: используйте вложенные операторы `if`.)

2

Продолжаем погружение: переменные

Для начинающего программиста речи о переменных могут звучать пугающе. Но в этой главе мы устраним все сложности и надеемся, что приведенная информация послужит для читателей факелом в царстве тьмы. К концу главы вы будете уверенно разбираться в переменных и применять их в своих программах.

В этой главе вы узнаете:

- Что такое переменные
- Как хранить данные
- Как объявлять переменные и присваивать им значения
- Об основных типах данных
- Как определять размер переменной
- Как использовать `typedef`
- Как преобразовывать шестнадцатеричные числа в десятичные
- Как выполнять приведение типов
- Как пользоваться константами

Что такое переменная

Помните, еще в школе вам давались формулы вроде $3x + 5$? В них x мог быть любым числом. Часто накладывалось сопутствующее ограничение: x мог быть целым числом, рациональным числом и т. д. Это ограничение определяло спектр возможных значений x. То же справедливо и для переменных.

Переменная – это символ, который представляет численные значения, строковые (текстовые) значения или логические значения (истина и ложь). Определение переменной содержит ее тип (например, целочисленный), и переменная может представлять только числа этого типа.

К примеру, если x – целое число, цифра 2 будет допустимым значением x, а число 2,5 – нет.

Каждой переменной одного из основных типов (см. раздел «Знакомьтесь, основные типы переменных» позже в этой главе) сопутствует три фрагмента информации: *идентификатор*, или имя, по которому к переменной обращается программист; *размер*, который сообщает компьютеру объем хранимых данных; и собственно *данные*. Имя и размер присваиваются переменной в зависимости от ее предназначения.

В мире компьютеров переменная – это определенный сегмент компьютерной памяти. Данные, связанные с конкретной переменной, хранятся в присвоенном ей сегменте. Представьте себе, что несколько коробок равного размера выстроены в ряд и пронумерованы последовательно (первая имеет номер 1, вторая – номер 2 и т. д.). Так работают переменные. Каждая переменная занимает одну или несколько коробок, в которых и содержатся данные, хранимые в переменной. Этими коробками являются ячейки памяти компьютера. В них хранятся данные переменных. Присвоив переменной имя, мы получаем удобный доступ к самой переменной и связанной с ней памяти (рис. 2.1).

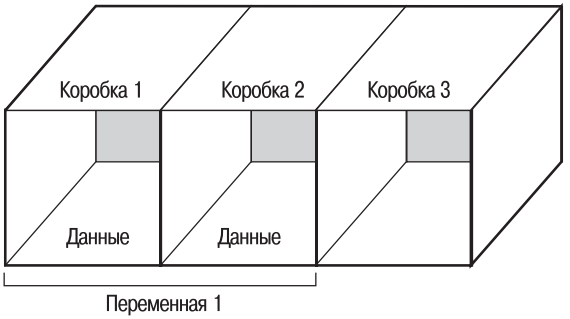


Рис. 2.1. *Первые две коробки диаграммы (байты) принадлежат памяти, отведенной под хранение Переменной 1. Третья коробка – свободное пространство, поскольку она не зарезервирована под хранение данных других переменных*

Разбираемся в отношениях переменных и памяти

По мере накопления опыта вы будете все отчетливее понимать, что большая часть жизни программиста связана с обработкой данных и принятием решений по их обработке. Данные, информация – это сердце мира компьютеров. Если вы знаете, как манипулировать данными, вы умеете программировать.

В целом, ссылаться на данные можно двумя способами: как на литералы или как на основные типы данных. *Литералы* – это буквальные значения данных. Например, число `2` и слово `Привет` являются литера-

лами. В главе 1 «Путешествие начинается» вы встретили много литералов. "Приближаются драконы!" – это строковый литерал, а цифра 8 – целочисленный литерал. Если вы говорите о данных в традиционной манере, то, вероятнее всего, используете литералы.

С другой стороны, *основные типы данных* являются уникальными для программирования. Идея основных типов заключается в обращении к данным без использования литералов. Этот подход является более выгодным, поскольку создает уровень абстракции между кодом и данными, с которыми он работает; это означает, что строка кода может при каждом вызове обрабатывать различные данные.

Основные типы разделены на четыре категории, каждая из которых соответствует определенному виду данных: *логический тип*, *символьные типы*, *целочисленные типы* и *типы с плавающей точкой*. С помощью этих четырех категорий можно представить любые данные в программе.

Но прежде чем продолжить изучение переменных, вы должны узнать немного больше о памяти и о хранении данных.

Существует два базовых типа компьютерной памяти: *оперативная память* (Random-Access Memory, RAM) и *постоянная память* (Read-Only Memory, ROM). Оперативная память состоит из микросхем памяти, расположенных на материнской плате и периферийных картах (графических, звуковых и т. д.). Оперативная память является временным хранилищем данных. Это память, которая используется для работы приложений, и именно этот вид памяти более всех других интересует программистов. ROM-память является *энергонезависимой* и не может изменять состояние; в ней хранятся инструкции загрузки и автоматического тестирования, которые выполняются при включении компьютера. Наш разговор ограничится оперативной памятью (RAM).

*Другим видом памяти является **дисковая память**. Дисковыми накопителями являются жесткие и гибкие диски, компакт-диски и другие устройства временного и долговременного хранения данных. Дисковые носители сохраняют записанную информацию при выключении компьютера.*

Компьютерная память состоит из миллионов крохотных электрических ключей, которые называются битами. (*Бит* – самая маленькая единица хранения информации на компьютере.) Каждый ключ, или бит, имеет два состояния для различных уровней напряжения (например, 0 и +5 вольт). Таким образом, каждый бит может представлять одно из двух значений – назовем их 0 и 1 или истиной и ложью. Следовательно, один ключ хранит один бит (0 или 1). Сочетание двух битов дает четыре варианта комбинаций значений (00, 01, 10 и 11). Так рождается *двоичная* система счисления, или система счисления по основанию два.

Наиболее распространенной системой счисления является десятичная (или система счисления по основанию десять). В ней каждая цифра

имеет одно из десяти возможных состояний (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). В табл. 2.1 приведены числа от 0 до 10, их двоичные и десятичные варианты.

Таблица 2.1. Двоичная и десятичная системы счисления

Десятичное	Двоичное
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Пусть вас не слишком пугает двоичная система. Старые добрые числа остались на месте; просто в двоичной системе они представлены по-другому. Преобразование числа из двоичной системы в десятичную выполняется достаточно легко. Возьмите первую цифру (крайнюю правую цифру двоичного числа, в нашем случае 1) и умножьте на 2^0 (где *x^a* означает *x* в степени *a*). Затем возьмите следующую цифру и умножьте на 2^1. Повторяйте процесс, пока не закончатся двоичные цифры. Для числа 0110101 преобразование выглядит так:

```
1 x 2^0    =    1
0 x 2^1    =    0
1 x 2^2    =    4
0 x 2^3    =    0
1 x 2^4    =   16
1 x 2^5    =   32
0 x 2^6    =    0
            ------
           =   53
```

К сожалению, преобразование из десятичной системы в двоичную несколько сложнее. Прежде всего, разделите число на 2^(n), где *n* – колонка (в числе 101 три колонки), с которой вы работаете. Затем отнимите от числа полученный при делении остаток. Чтобы найти значение двоичной цифры в колонке, разделите новое число на 2^(n–1). Повторяйте эти шаги, пока при вычитании остатка из числа не получится 0. Вот шаги преобразования числа 9 в двоичную форму:

Колонка номер 1:

```
9  / 2^1  =  4 Остаток: 1
Двоичная цифра = 1 / 2^0 = 1
Двоичное число на данном этапе: 1
9 - 1 = 8
```

Колонка номер 2:

```
8  / 2^2  =  2 Остаток: 0
Двоичная цифра = 0/2^1 = 0
Двоичное число на данном этапе: 01
8 - 0 = 8
```

Колонка номер 3:

```
8  / 2^3  =  1 Остаток: 0
Двоичная цифра = 0/2^2 = 0
Двоичное число на данном этапе: 001
8 - 0 =  8
```

Колонка номер 4:

```
8  / 2^4  =  0 Остаток: 8
Двоичная цифра = 8/2^3 = 1
Двоичное число на данном этапе: 1001
8 - 8 =  0
```

Как мы уже говорили, электрический ключ называется битом и может иметь два значения, 0 или 1. Последовательность из восьми битов называется *байтом*. Байт может иметь 256 различных значений, от 0 до 255. Вспомним аналогию с коробками из начала главы: ряды коробок представляют память. Каждая коробка – один байт памяти. Байт – минимальный объем информации, с которым работает компьютер. Если необходимо хранить более одного байта, компьютер задействует соответствующее число коробок. *Килобайт* (Кбайт) содержит 1024 байта. Помните, не 1000 байт, а именно 1024. Память не до конца подчиняется метрической системе. 1024 килобайта составляют мегабайт (Мбайт). 1024 мегабайта – гигабайт (Гбайт). Эта информация сведена в табл. 2.2.

Таблица 2.2. Единицы измерения памяти

Единица измерения памяти	Число битов	Число значений
бит	1	2
байт	8	256
килобайт (Кбайт)	8192	2^8192
мегабайт (Мбайт)	8388608	2^8388608
гигабайт (Гбайт)	8589934592	2^8589934592
терабайт (Тбайт)	8.796093022 * 10^12	2^(8.796093022 * 10^12)

Считайте микросхему оперативной памяти последовательностью байтов. Каждый байт имеет уникальный адрес. *Адрес* – это число (например, 10345), которое позволяет компьютеру определить, о каком из

байтов идет речь. Однако число, которое выдает компьютер, представлено в *шестнадцатеричной* системе счисления, т. е. в системе с основанием 16. Числа на коробках являются адресами памяти.

Итак, узнав чуть больше о хранении и представлении данных компьютером, вы готовы продолжить исследование переменных.

Идентификаторы переменных

Идентификаторы – имена, которые присваиваются переменным и используются для обращения к ним. Например, переменная, в которой хранится ввод пользователя, может иметь имя `input`. Правила именования переменных таковы:

- Имя должно начинаться с буквы или символа подчеркивания (_).
- Последующие символы могут быть буквами, цифрами или символами подчеркивания.
- Идентификаторы могут быть длинными (более 200 символов).
- Ключевые слова языка запрещено использовать в качестве имен переменных.
- Язык C++ чувствителен к регистру символов (имена *aVariable* и *aVARIABLE* являются различными).

Допустимыми идентификаторами являются `_file5G`, `String7F4`, `input`, `__my_variable` и т. д. Для сравнения приведем примеры недопустимых идентификаторов: `9variable` и `&variable`. Двойное подчеркивание в начале идентификатора – идея не очень хорошая, поскольку такие имена часто резервируются для специальных системных переменных. Кроме того, помните, что идентификатор должен быть не только допустимым, но и полезным. Идентификатор должен описывать информацию, с которой связан, чтобы читая свой код три недели спустя, вы могли четко понять, для чего нужна та или иная переменная.

Избегайте даже случайного использования ключевых слов в качестве имен переменных. (Возможно, вы помните из главы 1, что ключевые слова – это основной набор команд и функций, применяемых в программировании.) Такое использование ключевых слов может приводить к непредсказуемым результатам и ненужной головной боли. Например слово `int` *зарезервировано для представления целых чисел и его нельзя использовать в качестве имени переменной – компилятор выдаст ошибку в процессе сборки программы.*

Объявления переменных и присвоение значений

Прежде чем использовать переменную, ее необходимо объявить. Объявление переменной является инструкцией компьютеру зарезервиро-

вать определенный объем памяти под хранение данных и присвоить этому объему имя. Различным типам переменных требуются различные объемы памяти. Например, для объявления целочисленной переменной следует использовать такую строку:

```
int x;
```

Общий синтаксис объявления переменных таков:

```
тип_переменной идентификатор;
```

Для присвоения значений переменным используется оператор присваивания (=). *Операторы* – это символы или слова, которые предписывают выполнение... да-да, *операций*. Операторы могут быть математическими, операторами отношений и др. В этой главе мы рассмотрим только математические операторы и оператор присваивания. Оператор присваивания помещает значения из правой части в значения, указанные в левой части. Пример:

```
x = 5;
```

Переменная x теперь хранит значение 5, то есть память, зарезервированная под переменную x, заполнена числом 5. Не забывайте об отличиях знака равенства в математике и программировании. В данном случае число из правой части равенства присваивается в качестве значения переменной из левой части. Эта операция не имеет ничего общего с проверкой равенства частей. Единичный знак равенства *не* имеет значения «равно». Потратьте лишнюю минуту на повторное прочтение этой информации, поскольку она очень важна.

В правой части оператора присваивания также могут фигурировать переменные:

```
x = y;
```

Переменная x получает значение, хранимое в переменной y.

Помните, что в левой части оператора присваивания не могут фигурировать литералы:

```
5 = x;
```

Это недопустимый оператор. Он предписывает сохранить значение, хранимое переменной x, в числе 5, что не имеет смысла.

Чтобы присвоить переменной значение при объявлении, воспользуйтесь такой конструкцией:

```
тип переменной идентификатор = значение;
```

Это называется *инициализацией переменной*, поскольку она получает начальное значение, инициализируется.

Научившись объявлять переменные и присваивать им значения, мы готовы перейти к изучению различных типов переменных.

Знакомьтесь, основные типы данных

Существует довольно много типов переменных, но в этой главе мы рассмотрим лишь четыре из них. Тип переменной определяет объем резервируемой под нее памяти, а значит, и предельные значения хранимых чисел (или объем хранимых данных). В этой главе мы изучим логические (булевые), символьные, целочисленные типы и типы с плавающей точкой. Некоторые из существующих типов переменных представлены в табл. 2.3.

Таблица 2.3. Типы переменных

Тип	Размер	Значения
bool	1 байт	true (1) или false (0)
char	1 байт	от 'a' до 'z', от 'A' до 'Z', от '0' до '9', пробел, табуляция и т. д.
int	4 байта	от –2 147 483 648 до 2 147 483 647
short	2 байта	от –32 768 до 32 767
long	4 байта	от –2 147 483 648 до 2 147 483 647
float	4 байта	+–(1.2 x 10^–38 до 3.4 x 10^38)
double	8 байт	+–(2.3 x 10^–308 до 1.7 x 10^308)

Логический тип

Логический (или булевый) тип является простейшим типом данных. Его размер равен одному байту (минимальный объем данных, с которым работает компьютер). Он может иметь два значения, true (истина) и false (ложь). Переменные этого типа используются для случаев, когда есть лишь два возможных значения. К примеру, может существовать переменная end, которая имеет значение false, пока не закончится программа. Эту переменную можно использовать, чтобы определить, достигнут ли конец программы. Логические переменные объявляются при помощи ключевого слова bool:

```
bool myBool = true;
```

Символьные типы

Переменная *символьного типа* может хранить один из 256 различных символов, перечисленных в приложении С «Стандартная таблица символов ASCII».

Символьный литерал (буквальное представление данных, не сохраненных в переменной) заключается в одинарные кавычки. Примеры символьных литералов: 'a', '5', '%', 'W'.

Переменные символьного типа объявляются при помощи ключевого слова `char`:

```
char myChar = 'a';
cout<< "буква " << myChar;
```

Целочисленные типы

Существует три целочисленных типа: `int`, `short int` (или `short`) и `long int` (или `long`).

С типом `int` мы познакомились в главе 1. Его длина составляет 4 байта (на большинстве компьютеров). Он может хранить числа в диапазоне от –2 147 483 648 до 2 147 483 647. Этот тип данных применяется чаще всего. Переменные целочисленных типов не могут хранить десятичные и рациональные дроби (например, 2.1 или 1/3). Объявление переменных этого типа происходит при помощи ключевого слова `int`:

```
int myInt = -3;
```

Тип `short` в два раза меньше типа `int`, то есть его длина – 2 байта. Он может хранить числа диапазона от –32 768 до 32 767. Этот тип весьма полезен для хранения небольших значений и вдвойне эффективен, поскольку занимает в два раза меньше памяти. Переменные этого типа объявляются при помощи ключевого слова `short` (или `short int`).

```
short myShort = -56;
```

Последняя строка может быть переписана следующим образом:

```
short int myShortInt = -56;
```

Следующим в этом замечательном путешествии по типам данных будет тип `long`. Он имеет ту же длину, что и `int`, 4 байта. Диапазоны хранимых значений также совпадают: от –2 147 483 648 до 2 147 483 647. Читатели уже, вероятно, догадываются, что переменные этого типа объявляются при помощи ключевого слова `long` (или `long int`). Этот тип является синонимом для `int`:

```
long myLong = 32056;
```

абсолютно равнозначно

```
long int myLong = 32056;
```

Чтобы ограничить диапазон значений целочисленной переменной только положительными числами, можно предварить ее объявление ключевым словом `unsigned` (беззнаковое). Тип `unsigned short` ограничен диапазоном от 0 до 65 535, а `unsigned long` или `int` – диапазоном от 0 до 4 294 967 295. Беззнаковые целые занимают столько же памяти, сколько и обычные (знаковые) целые. Чтобы гарантировать возмож-

ность хранения отрицательных значений, можно использовать ключевое слово signed, но в этом нет необходимости, поскольку знаковость является стандартом для объявленных переменных.

```
unsigned int anUnsignedInt; // от 0 до 4 294 967 295
unsigned short anUnsignedShort; // от 0 до 65535
signed long aSignedLong; // от -2 147 483 648 до 2 147 483 647
int anInt; // от -2 147 483 648 до 2 147 483 647
```

Циклический возврат целых чисел

Если целое число выходит за пределы допустимого диапазона, происходит особенное событие. Отсчет начинается с начала. Например, если присвоить переменной типа unsigned integer значение 4 294 967 295 (которое на единицу больше максимально допустимого), в действительности будет записано нулевое значение (0). Для беззнаковых целых отсчет начинается с цифры 0, а для знаковых – с нижней границы диапазона. Поясним изложенное таким примером:

```
//2.1 - Циклический возврат целых чисел - Дирк Хенкеманс и Марк Ли - Premier
Press
#include <iostream>
using namespace std;

//демонстрация циклического возврата для целых чисел
int main()
{
     unsigned int unInt = 4294967295;
     signed short aShort = 32767;
     cout << "Значение беззнакового int: " << unInt << endl;
     cout << "Значение short: " << aShort << endl;
     unInt = unInt + 1;
     aShort = aShort + 1;
     cout << endl << "Новое значение беззнакового int: "
          << unInt << endl;
     cout << "Новое значение short: " << aShort << endl;
     return 0;
}
```

Вывод:

```
Значение беззнакового int: 4294967295
Значение short: 32767
Новое значение беззнакового int: 0
Новое значение short: -32768
```

В строках 8 и 9, соответственно, переменным unInt и aShort присваиваются максимальные значения. В строках 12 и 13 значения переменных увеличиваются на единицу. Как можно видеть, произошел циклический переход. Это второстепенный момент, но о нем следует помнить на всякий случай.

Оператор приращения

Чтобы увеличить значение, хранимое целочисленной переменной, ровно на единицу, можно воспользоваться оператором приращения (++), который позволяет существенно сократить код. *Оператор инкремента* является математическим оператором, который показывает, что значение переменной увеличивается на единицу относительно текущего. (*Математическим оператором* является любой, позволяющий производить вычисления.) Итак, вместо строки

```
count = count + 1;
```

можно написать

```
count++;
```

Аналогичным образом можно использовать *оператор декремента* (--) для уменьшения целочисленного значения на единицу. Оператор декремента вычитает единицу из значения числа. Вместо

```
count = count - 1;
```

можно написать

```
count--;
```

Типы с плавающей точкой

Для хранения десятичных или рациональных дробей следует использовать типы данных с плавающей точкой, `float` и `double`.

Тип данных `float` (или тип данных с плавающей точкой одинарной точности) имеет длину 4 байта. Он может хранить положительные и отрицательные целые и действительные значения с максимальной и минимальной точностью в 1.2×10^{-38} и 3.4×10^{38}. Помните, что эти числа могут быть положительными или отрицательными.

Программисты часто записывают действительные числа в *экспоненциальном представлении*. Экспоненциальное представление позволяет емко выражать очень маленькие или очень большие числа.

Экспоненциальное представление – это просто метод записи действительных чисел, с которыми становится неудобно работать. Например, число 0.0000000000000000002. Вот некоторые числа в экспоненциальном представлении:

```
5.6543e17     -4.02934e5     -17.204e-10     2.0e-19
```

Правило преобразования экспоненциальной записи в традиционный формат: сдвиньте десятичную дробь на число позиций, определенное цифрой справа от буквы `e` (она называется *экспонентой*). Чтобы преобразовать запись 5.6543e17 в традиционный формат:

1. Выделите число (17) справа от буквы e.
2. Сдвиньте десятичную запятую на 17 десятичных позиций вправо. Направление однозначно определяется по знаку числа 17 (в данном случае – положительный). Заполните образовавшиеся пустые позиции нулями.

   В результате получаем 563430000000000000.0, что представляет собой то же, что и 5.6543e17. Последнее число гораздо удобнее набирать, чем 563430000000000000.0.

Если экспонента числа отрицательная, скажем, 5.6543e–17, десятичную запятую следует сдвинуть на 17 позиций влево; в результате получим 0.00000000000000056343.

При хранении все числа преобразуются в экспоненциальную запись, а затем округляются до ближайшей позиции 10^–38. 38 знаков – это уровень точности для этого типа данных. Например, если сохранить число π (3.14159265358979323846...) с точностью в два десятичных знака, число будет округлено до 3.14.

Переменные с плавающей точкой одинарной точности объявляются при помощи ключевого слова `float`:

```
float aFloat = 3.156;
float negativeFloat = -678.876;
```

Чтобы запомнить рациональную дробь, компьютер должен привести ее из формата дробей a/b к десятичной форме. При следующем обращении к рациональной дроби компьютер выдаст наиболее точное из достижимых для него десятичных представлений этой дроби.

Тип данных `double` имеет длину 8 байт. Он может хранить положительные значения из интервала от 2.2×10^{-308} до 1.7×10^{308} и отрицательные значения из интервала от -2.3×10^{-308} до -1.7×10^{308}. Таким образом, значения `double` округляются до ближайшей позиции 10^–308. Это невероятно высокая точность. В обычных условиях переменные `double` способны сохранить любое значение. Объявление переменных производится при помощи ключевого слова `double`. Запись *<число_a>*e*<число_x>* означает *<число_a>**10^*<число_x>*.

В следующем примере число 2.2e–308 абсолютно эквивалентно 2.2*10^–308:

```
double aDouble = 2.2e-308;
double negDouble = -2.3e-308;
```

Итак, вы познакомились с основными типами данных. Привыкните к тому, чтобы использовать наиболее короткие из подходящих типов. Это сэкономит память и сделает ваши программы более быстрыми.

Оператор sizeof()

Этот относительно простой оператор является частью языка C++. Для любой переменной он возвращает объем зарезервированной для нее памяти в байтах. Формат вызова:

```
sizeof(идентификатор);
```

Чтобы вывести размер переменной типа `double` на экран, воспользуйтесь следующим фрагментом кода:

```
double aDouble;
cout << "Размер переменной типа double: " << sizeof(aDouble); // равен 8
```

Поскольку оператор возвращает целое число (*количество* байтов), его можно использовать в любом контексте, где может быть использовано целочисленное значение.

Кроме того, этот оператор можно выполнить не только для идентификатора, но и для ключевого слова типа, как показано ниже:

```
cout<< "Размер двух чисел типа int равен " << 2 * sizeof(int); // равен 4
```

Игра «Типы данных»

На своем неблизком пути вы встретили поселение эльфов. Тебе повезло, о смелый путешественник, потому что выпал идеальный шанс испытать приобретенные умения. Деревню атакует полчище драконов! Чтобы защитить ее, необходимо вызвать из мистического леса воинов типов данных, которые смогут спасти эльфов.

Для спасения деревни придется использовать знания о целых числах и числах с плавающей точкой различной точности, приобретенные в предыдущем разделе. Готовы принять вызов? Тогда читайте дальше:

```
//2.2 - Игра "Типы данных"- Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;   //используем пространство имен std

//в главных ролях - типы данных
int main( void )
{
     int intWarriors;
     double doubleWarriors;
     float floatWarriors;

     cout << "Поселение эльфов атаковали "
          <<" драконы."<< " Чтобы спасти жителей,  ";
     cout << "необходимо создать воинов каждого типа данных "
          <<"и защитить город." << endl << endl;
     cout << "Сколько воинов типа int послать в атаку?";
     cin >> intWarriors;
```

```
    cout << endl << "К счастью, каждый воин обладает силой"
         << sizeof(intWarriors) << ", " << endl
        << "и этого почти достаточно, чтобы победить синих драконов." << endl;

    cout << endl << "Скорее! Сколько воинов типа double "
          << "послать в атаку?";
    cin >> doubleWarriors;
    cout << endl << doubleWarriors;
     cout << " воинов типа double атакуют оставшихся синих "
           << "драконов" << endl << "Они убили "
           << sizeof(doubleWarriors) << " синих драконов."
          << " Все синие драконы мертвы."
           << endl << endl;

    cout << "Сколько воинов типа  float послать в атаку?";
    cin >> floatWarriors;
    cout << endl << "Каждый из "<< floatWarriors
           <<" воинов типа float выпускает ";
    cout << sizeof(floatWarriors) << " стрел." << endl;
    cout << "Этого как раз достаточно, чтобы убить зеленых драконов."
            << endl << "Поздравляем, ты спас "
           << "эльфов!";
}
```

typedef облегчает жизнь

Периодически становится весьма неудобно раз за разом пользоваться такими ключевыми словами, как `unsigned short int`. Оператор `typedef` позволяет переименовать тип переменной. Можно заменить `unsigned short` на `USHORT` или на еще более удобный вариант. Формат `typedef` следующий:

```
typedef тип_переменной новое_имя;
```

Переименовать `float` в `f` можно так:

```
typedef float f;
```

А теперь объявляем переменную типа `float` с помощью нового ключевого слова:

```
f radius = 4.7639;
```

Помните, что новое имя типа должно быть не только удобным, но и содержательным.

Приведение типов

Иногда бывает удобно произвести преобразование из одного типа в другой. Такое преобразование получило название приведения. (*Приве-*

дение, называемое еще *приведением типов*, – это процесс преобразования данных из одного типа в другой с сохранением почти такого же значения.) При работе с некоторыми библиотеками может требоваться применение конкретных типов данных (это часто происходит при работе с DirectX). Приведение выполняется следующим образом:

```
float pi = 3.14;
int roundedPi = (int) pi;
```

В этом фрагменте кода `roundedPi` получает значение `3`. Дробная часть отсекается. Обратите внимание, что в данном случае не происходит округление до ближайшего целого – дробная часть просто удаляется.

Применение констант

В C++ существует два вида констант. С первым из них вы уже знакомы, это *литеральные константы*. Литеральная константа и литерал – это одно и то же (более подробно о литералах рассказано в разделе «Разбираемся в отношениях переменных и памяти» ранее в этой главе). В качестве примеров литеральных констант можно привести число `2` и строку `"Hello"`. Они являются константами по определению, ведь их значения невозможно изменить.

Второй тип констант – *символьные константы*. Символьная константа во многом похожа на переменную, но ее значение нельзя менять. В начале программы можно создать константу `PI` со значением `3.14`. При компиляции кода каждое вхождение идентификатора `PI` будет заменено числом `3.14`, и это означает, что по крайней мере для компьютера нет разницы между символьной константой и литеральной константой. Но прежде чем вы начнете задаваться вопросом, зачем и когда следует использовать символьные константы, необходимо научиться их использовать.

Существует два способа объявления констант. Первый связан с директивой `#define`. (*Директива* – это строка кода, которая исполняется компилятором непосредственно перед компиляцией оставшейся части кода.) Эта директива имеет следующий синтаксис:

```
#define ИМЯКОНСТАНТЫ значение
```

`ИМЯКОНСТАНТЫ` – это идентификатор, по которому можно будет ссылаться на константу, а `значение` – значение константы. Обратите внимание, как и в случае с директивой `#include`, отсутствует точка с запятой в конце строки. Директива не является исполняемым оператором C++; она лишь предписывает компилятору выполнить определенные действия перед компиляцией. Директива `#define` предписывает компилятору заменить в программе все вхождения идентификатора `ИМЯКОНСТАНТЫ` значением этой константы. Компилятор, выполнив подстановку, переходит к компиляции кода. В программе при этом можно использо-

вать ИМЯКОНСТАНТЫ вместо литеральных констант. К примеру, можно объявить константу для числа π:

```
#define PI 3.14159
```

И в коде вместо подобных конструкций:

```
y = 3.14159 * x;
```

использовать такие:

```
y = PI * x;
```

Первое и самое заметное преимущество использования констант – повышение читаемости исходного текста. Число 3.14159 само по себе не подразумевает значение π; но если создать константу с именем PI, смысл происходящего становится предельно ясным.

Второй способ определения констант связан с ключевым словом const. Синтаксис почти такой же, как для #define:

```
const типКонстанты ИМЯКОНСТАНТЫ значение;
```

Ключевое различие заключается в том, что const определяет тип константы, и в качестве такового может использоваться любой существующий тип данных. Это отличие говорит в пользу const, поскольку хорошо сочетается с проверкой типов. Компилятор имеет возможность убедиться, что константа используется в контексте, где ее тип не вызовет конфликта. К примеру, если создать строковую константу

```
const string HELLO "Привет";
```

или так:

```
#define HELLO "Привет"
```

именем константы можно пользоваться в коде вместо строки "Привет".

Но следует помнить, что const string HELLO дает возможность проверки типа, и константу HELLO можно будет использовать только там, где можно использовать обычные строки. Однако #define HELLO не имеет подобных ограничений, и это может приводить к странным результатам:

```
int x = HELLO;
```

Для константы, определенной посредством const, возникнет ошибка в процессе компиляции, но этого не произойдет для константы #define. Переложив поиск возможных проблем на компилятор, вы избавитесь от необходимости самостоятельно искать ошибку. А это очень существенное преимущество при разработке больших приложений.

Определим константу PI вторым способом:

```
const float PI 3.14159;
```

Не забывайте инициализировать константы при их создании. После создания ее значение невозможно изменить. Если не инициализировать ее при объявлении, возникнет ошибка при попытке установить значение позже.

Применение констант имеет много положительных сторон. Представьте, что в своей программе вы набирали `3.14` всякий раз, когда требовалось значение числа π. И вдруг вы осознали, что для работы приложения требуется более высокая точность. Придется заменить каждое вхождение числа 3.14 более точным значением, скажем, 3.14159. Использование директивы для создания константы `PI` позволило бы менять точность легким изменением `#define` и перекомпиляцией программы.

Истории из жизни

Вообразите, что являетесь руководителем крупной корпорации, в которой каждый сотрудник получает $10.78 в час, и решили повысить оплату до $11.78 в час. Очень затруднительно будет редактировать тысячи записей в программе расчета зарплаты. В случае применения константы для хранения информации о ставке понадобится изменить лишь эту константу, что сэкономит вам лично и всей корпорации массу времени и усилий.

Если речь идет об очень длинных числах (скажем, π с точностью до 22 знаков после запятой), весьма утомительно набирать их при каждой необходимости; а если вы похожи на нас, то запросто можете сделать ошибку при наборе. Константы позволяют избежать подобных затруднений.

Игра «Круги»

Ваше домашнее задание по математике на завтра: вычислить площадь круга и длину окружности. Не исключено, что мы сможем посодействовать. В следующей программе мы выполним обе задачи с помощью констант. Воспользуемся константой для хранения значения числа π (3.141592). Пользователю останется лишь ввести радиус круга, а программа вычислит площадь круга и длину окружности. Обратите внимание: ввод пользователя должен храниться в переменной типа `float`, чтобы пользователь мог указывать дробные значения радиуса. Отлично, пора писать код:

```
//2.3 - Игра "Круги" - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;

//Вычисляет площадь круга и длину окружности
int main()
{
```

```
    typedef float f;
    const f PI = 3.141592;
    f radius, circumference, area;

    cout << "Добро пожаловать в программу создания кругов!" << endl;
    cout << "Какой изволите определить "<<endl
        <<"радиус для круга? ";
    cin >> radius;

    area = PI * radius * radius;
    circumference = PI * (radius * 2);
    cout << "Площадь круга: " << area << endl;
    cout << "Длина окружности: "
        << circumference << endl;
    cout << "Спасибо, что играли в создание кругов!"
         << endl;

    return 0;
}
```

Повторяем синтаксис

К этому моменту мы изучили разнообразные типы слов – ключевые, идентификаторы, директивы и т. д., – а также многие виды операторов – присваивания, включения и пр. В этом разделе мы повторим уже изученные конструкции синтаксиса языка C++. (*Синтаксис* – это грамматические правила языка.)

Начнем с ключевых слов – фундамента языка C++. Эти слова служат основой для создания программ. Компилятор C++ понимает только эти слова, пока программист не «научит» его другим.

В табл. 2.4 перечислены уже изученные нами ключевые слова, описана их функциональность и синтаксис.

Повторимся, ключевые слова – это сердце языка C++. Каждое из них имеет особое значение и смысл. Познав эти слова, вы окажетесь на верном пути к вершинам мастерства в C++.

Следом идут идентификаторы. Вам известно три типа идентификаторов: литералы, идентификаторы констант и идентификаторы переменных. В табл. 2.5 указаны их применения и форматы.

Идентификаторы литералов выбирать не приходится: им даются имена, соответствующие значениям. В идентификаторах констант обычно все буквы заглавные, хотя это необязательно. В идентификаторах переменных обычно все буквы строчные, хотя слова в составных именах выделяются (к примеру, `variableName`). Старайтесь делать идентификаторы как можно более наглядными; это повысит прозрачность кода.

Таблица 2.4. Ключевые слова C++ и их назначение

Ключевое слово	Назначение	Синтаксис
`const`	Объявление констант	`const тип_константы имя_константы значение;`
`int`	Объявление переменных типа int	`int имя_переменной;`
`short`	Объявление переменных типа short	`short имя_переменной;`
`long`	Объявление переменных типа long	`long имя_переменной;`
`float`	Объявление переменных типа float	`float имя_переменной;`
`double`	Объявление переменных типа double	`double имя_переменной;`
`bool`	Объявление логических переменных	`bool имя_переменной;`
`string`	Объявление переменных типа string	`string имя_переменной;`
`unsigned`	Ограничение целочисленных переменных положительными значениями	`unsigned целочисленный_тип имя_переменной;`
`return`	Необходимо в конце функции `main()`	`return 0;`
`sizeof()`	Возвращает размер переменной	`sizeof(тип_переменной)`
`void`	Необходимо для объявления функции `main()`	`int main (void)`
`main`	Начало основной программы	`int main (void) { }`
`typedef`	Переименование типов переменных	`typedef тип_переменной новое_имя;`

Таблица 2.5. Применения и форматы идентификаторов

Идентификатор	Применение	Формат
Литерал	Хранение буквальных значений (3, 2)	Нет правил форматирования (просто используйте значение)
Переменная	Хранение изменяющихся значений	Присвоить литерал идентификатору переменной с помощью оператора присваивания (=)
Константа	Хранение констант	Правило для переменных; присваивание возможно только в момент создания константы

Мы еще не рассматривали в подробностях директивы препроцессора, но две из них вы уже знаете. Речь идет о `#include` и `#define`. (Как мы уже говорили, директивы являются предписаниями компилятору выполнять определенные действия в процессе компиляции.) `#include` сообщает компилятору, что следует добавить внешний файл к тому, который компилируется. Скажем, в программе «Здравствуй, мир» строка

```
#include <iostream>
```

сообщает компилятору, что необходимо добавить библиотеку `iostream` в файл `hello.cpp` при компиляции. Эта библиотека содержит довольно большой объем кода, включая и реализацию оператора `cout`.

`#define` предписывает компилятору заменять каждое вхождение имени определяемой константы ее значением в процессе компиляции. Операторы директив препроцессора не заканчиваются точкой с запятой.

Как мы уже рассказывали, операторы – это символы или слова, выполняющие определенные операции. Вы уже изучили девять операторов. Математические – сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`) и взятие остатка (`%`); оператор приращения (`++`), оператор присваивания (`=`), оператор `sizeof()` и оператор приведения типов (`( )`). В табл. 2.6 перечислены эти операторы и их назначение.

Научились ли вы обращаться с переменными свободно? Если нет, советуем повторно прочитать разделы этой главы и попрактиковаться с примерами кода. Практика поможет решить все проблемы.

Таблица 2.6. Применение и синтаксис операторов

Оператор	Применение	Синтаксис
Сложения (`+`)	Сложение двух чисел	`число1 + число2`
Вычитания (`-`)	Вычитание для двух чисел	`число1 - число2`
Умножения (`*`)	Умножение двух чисел	`число1 * число2`
Деления (`/`)	Деление двух чисел	`число1 / число2`
Взятия остатка (`%`)	Взятие остатка от деления двух чисел	`число1 % число2`
Приращения (`++`)	Увеличение целого числа на единицу	`целое++`
Присваивания (`=`)	Присвоение значения	`переменная1 = число1`
`sizeof()`	Вычисление размера типа данных	`sizeof(идентификатор)`
Приведения типов (`( )`)	Изменение типа значения переменной	`(тип)Переменная`

Пишем игру «Оружейный магазин»

Пробираясь через темный лес, где-то у черта на рогах вы наткнулись на загадочный оружейный магазин. Тебе повезло, о достойнейший из путешественников, ибо это отличная возможность проверить приобретенные в этой главе знания – включая константы, приведение, операторы, типы данных и прочее, прочее... Чтобы посетить оружейный магазин, придется скомпилировать и выполнить эту программу:

```
//2.4  - Игра "Оружейный магазин" - Дирк Хенкеманс и Марк Ли
//Premier Press
#include <iostream>
#include <string>
using namespace std;

//код игры "Оружейный магазин".
int main (void)
{
     string name;
     cout << "Добро пожаловать в оружейный магазин, благородный рыцарь."
          << " Снова настала пора экипировать армию?" <<endl
          << "Как твое имя? ";
     cin >> name;
     cout << "Что ж, сэр " << name.c_str()
           << ", вперед, за покупками!" << endl;

     float gold = 50;
     int silver = 8;
     const float SILVERPERGOLD = 6.7;
     const float BROADSWORDCOST = 3.6;
     unsigned short broadswords;

     cout << "У тебя " << gold << " золотых монет и "
          << silver << " серебряных." <<endl<< "Это будет   ";
     gold += silver / SILVERPERGOLD;
     cout << gold << " золотых монет." << endl;

     cout<< "Сколько палашей ты желаешь приобрести?"
          <<" (каждый стоит 3.6 золотых)";
     cin >> broadswords;
     gold = gold - broadswords * BROADSWORDCOST;
     cout << "\nСпасибо за покупку. У тебя осталось " << gold << ".";
     silver = (gold - (int)gold) * SILVERPERGOLD;
     gold = (int)(gold);
     cout << "Это будет " << gold << " золотых и "
          << silver << " серебряных монет. " << endl
          << "Спасибо, что посетили Оружейный магазин. "
          << "Всего хорошего, сэр " << name.c_str();
     return 0;
}
```

Резюме

В этой главе мы изучили много всего. Вы узнали, как компьютер хранит информацию в памяти, как временно сохранять данные для последующего использования, а также о различных типах данных. Вы узнали о константах, определениях `typedef`, операторе присваивания и `sizeof()`. Кроме того, мы повторили уже изученный синтаксис языка C++.

Вы проделали большой путь и можете без опаски называть себя программистами. Это повод для гордости; очень немногие забираются так далеко.

В следующей главе мы познакомим вас с управляющими операторами, с которых начинается истинная власть над компьютером. Расслабьтесь на минутку, чтобы улеглась полученная информация, а затем экипируйтесь, прыгайте в кресло и не забудьте пристегнуть ремни – впереди изумительное путешествие!

Задания

1. Выберите подходящий тип переменной для хранения следующей информации:

   Число книг на книжной полке

   Стоимость этой книги

   Число людей в мире

   Слово *Привет*

2. Придумайте содержательные имена для переменных из первого задания.
3. Приведите две причины использования констант вместо литералов.
4. Напишите программу, которая вычисляет и отображает размеры всех основных типов.
5. Выясните, что происходит, когда символьный тип объявляется с модификатором `unsigned`. Соответствуют ли результаты ожиданиям? Сформулируйте причину соответствия или несоответствия.

3

Принимайте командование: управляющие операторы

В предшествующих главах мы создавали прямолинейные программы. Каждый оператор выполнялся лишь единожды, причем в обязательном порядке. Но такая линейность приемлема только для самых примитивных программ.

Чтобы создавать сложные, динамические программы, необходимо овладеть *управляющими операторами*, с помощью которых можно переходить к нужному фрагменту кода и выполнять фрагменты произвольное число раз. В этой главе вы изучите:

- Булеву логику и логические операторы
- Операторы выбора
- Операторы цикла
- Операторы ветвления
- Случайные числа

Логические операторы

В главе 2 мы отметили, что логическая переменная может иметь одно из двух значений: `true` (`1`) и `false` (`0`), а все условные операторы вычисляются в логические значения. Несколько операндов могут определить логическое значение условия (то есть является ли условие истинным или ложным); эти операнды называются *логическими операндами*. Их можно использовать для формирования *условий* (выражений, которые вычисляются в истинное или ложное значение). В этом разделе мы рассмотрим каждую из операций, существующих для таких операндов, и научим читателей по необходимости применять их.

Оператор равенства

Простейшим логическим оператором является оператор равенства. Оператор равенства (==, два символа равенства) является *бинарным* (или двухместным); это означает, что он работает для двух операндов, по одному с каждой стороны. Вот общий синтаксис оператора:

```
операнд1 == операнд2
```

Значение оператора является истинным, если значения двух операндов (вернее, их структуры и наполнение) совпадают. Поясним на примере. Допустим, объявлены переменные:

```
long elves = 8;
int dwarves = 8;
```

Их можно использовать в таких условиях:

```
if (elves == dwarves)  //true
if (dwarves == 8)      //true
if (dwarves == 0)      //false
if (dwarves == elves)  //true
```

Все перечисленные условия вычисляются в значение `true`, кроме третьего. Минуточку! Можно заметить, что эльфы и гномы представлены различными типами данных (целым и длинным целым), так что их битовые образы также различны – и это будет справедливо. Но дело в том, что язык C++ достаточно умен, чтобы учесть это обстоятельство. Во *время выполнения программы* для первого условия компьютер преобразует (*приводит*) переменную `dwarves` к типу `long` (самому длинному из типов данных, участвующих в выражении) и лишь затем сравнивает значения. Например, при сравнении типов `double` (8 байт) и `int` (4 байта) компьютер сначала приведет значение `int` к типу `float`, а затем выполнит сравнение. Третье условие вычисляется в значение `false`, поскольку значение переменной `dwarves` – `8`, а не `0`.

Оператор равенства является *левоассоциативным*; это означает, что при наличии нескольких операторах равенства в одном операторе их вычисление происходит в порядке следования, слева направо. Например, в условии вроде этого:

```
if (dwarves == elves == dragons)
```

прежде всего будет выполнено сравнение `dwarves` и `elves`. Полученное для этого сравнения значение (`true` или `false`) будет сравниваться с переменной `dragons`. Это последнее сравнение и определит результат вычисления всего выражения.

Не путайте оператор равенства (==) с оператором присваивания, состоящим из одного символа равенства (=). Присваивание назначает переменной в левой части оператора значение, указанное в его правой части. Оператор равенства проверяет операнды на равенство, не изменяя их. Нович-

ки часто путают эти операторы. Иначе говоря, не позволяйте тому, что вы уже знаете об этих символах, мешать изучению нового.

Знакомьтесь, оператор неравенства

Противоположностью оператору равенства является оператор неравенства (`!=`). Этот оператор также является бинарным и левоассоциативным. Его синтаксис практически идентичен синтаксису оператора равенства:

```
выражение1 != выражение2
```

Выражения-операнды должны быть допустимыми выражениями. Выражение, сформированное этим оператором, является условием. Условие истинно, если *выражение1* и *выражение2* не равны; условие ложно, если выражения равны. К примеру, для объявленных переменных

```
int plasmaGun = 50;
int rifle = 10;
```

можно создать следующие условия неравенства:

```
if (plasmaGun != rifle) //true
if (plasmaGun != 50)    //false
if (rifle != 0)         //true
```

Операторы «меньше» и «больше»

Знаки «меньше» (`<`) и «больше» (`>`) для компьютеров имеют практически такое же значение, как в курсе математики четвертого класса. Например, выражение `3 < 4` вычисляется в значение `true`, поскольку `3` меньше `4`. Аналогично для знака «больше»: условие `4 > 3` истинно, поскольку `4` больше `3`. Взгляните на следующие переменные и результаты сравнений:

```
int elves = 4;
int dwarves = 5;

if (elves<dwarves)      //true
if (dwarves < elves)    //false
if (dwarves > (2 / 3))  //true
if (elves < (23 *117))  //true
```

Оба оператора являются бинарными и левоассоциативными. Например, следующий код может работать не так, как вы ожидаете:

```
if (0 < x < 99)
```

Если вы еще не успели забыть математику, то можете подумать, что это проверка принадлежности `x` интервалу от `0` до `99`. Это не так. Прежде всего, `x` сравнивается с числом `0`. Результат этого сравнения (`true` или `false`) сравнивается с числом `99`. Поскольку как `true` (`1`), так и `false`

(0) – меньше, чем 99, это условие всегда будет истинным независимо от значения x.

Сочетания с оператором равенства

Возможно, из математики вы припоминаете и опыты со знаками «меньше или равно» и «больше или равно». В C++ эти знаки тоже существуют, хотя и записываются несколько иначе: вместо дополнительной черты под знаком «больше» или «меньше» за ним следует знак равенства. Соответственно, получаем оператор «больше или равно» (>=) и оператор «меньше или равно» (<=).

Эти операторы работают в точности как операторы «меньше» и «больше», с той разницей, что они возвращают значение true и в том случае, когда операнды равны. Вот несколько примеров:

```
if (5 <= 5) //true
if (6 >= 7) //false
if (1 <= 8) //true
```

Операторы «больше или равно» и «меньше или равно» также являются бинарными и левоассоциативными.

Логическое «или»

Логический оператор «или» (||) является бинарным. Он вычисляется в значение true, если хотя бы один из операндов имеет значение true. Прежде чем будет вычислено значение оператора, должны быть вычислены логические значения (true или false) обоих операндов.

Табл. 3.1 является первой из *таблиц истинности* (в ней приведены все возможные значения операндов и результаты вычислений оператора для этих значений). Первая колонка содержит значения первого операнда (который для удобства мы назовем *A*), вторая колонка – значения второго операнда (*B*), а третья – значение *A*||*B*.

Таблица 3.1. Таблица истинности для оператора «или»

Значение A	Значение B	Значение A‖B
True	True	True
True	False	True
False	True	True
False	False	False

Хотите верьте, хотите нет, но в табл. 3.1 содержатся все возможные комбинации значений операндов. Вот примеры условий с оператором «или» и результаты вычислений для этих условий:

```
if (true || false)        //true
```

```
if ((2>3) || false)       //false
if ((3*5) < 90 || (3<5))  //true
```

Оператор «или» также является левоассоциативным. Использование нескольких операторов «или» в пределах одного выражения не всегда столь же тривиально, как в случае операторов «меньше» и «больше». Если истинен хотя бы один из операндов, все условие также является истинным.

Это означает, что компьютер при вычислении результата может «срезать углы». Как только становится понятно, что нет необходимости вычислять оставшуюся часть оператора, компьютер прерывает вычисление. Например, если встречается условие с оператором «или» и первый операнд имеет значение `true`, компьютеру необязательно вычислять значение второго операнда. Разумеется, он избегает этих вычислений, поскольку знает, что все условие истинно независимо от значения второго операнда.

Оператор «и»

Оператор «и» похож на оператор «или». Он также является бинарным и левоассоциативным. Чтобы оператор «и» вычислялся в значение `true`, оба операнда должны иметь значение `true`. Оператор «и» представлен парой амперсандов (`&&`). Ниже представлена таблица истинности для оператора «и».

Таблица 3.2. Таблица истинности для оператора «и»

Значение A	Значение B	Значение A‖B
True	True	True
True	False	False
False	True	False
False	False	False

Вот несколько примеров применения оператора «и»:

```
if (true && true) //true
if ((3<4) && (3<5)) //true
if ((99> -32) && (-32>0.11)) //false
```

Оператор отрицания

Оператор отрицания (`!`) изменяет логическое значение на противоположное: `true` на `false`, а `false` на `true`. Оператор является *унарным*, или одноместным, то есть ему требуется всего один операнд. Таблица истинности для оператора отрицания представлена в табл. 3.3.

Некоторые примеры применения оператора отрицания:

```
if ( !true) // false
```

```
if ( !(2>3)) //true
if ( !( (2>3) || (3>2) ) ) //false
```

Таблица 3.3. Таблица истинности для оператора отрицания

A	!A
True	False
False	True

Ветвление кода и операторы выбора

Люди часто принимают решения исходя из определенных условий. Человек может пойти к врачу, если чувствует себя плохо. Решение, идти ли к врачу, основано на определенном условии: плохом самочувствии. То же верно и для компьютерных программ. Программу можно спроектировать таким образом, чтобы выбор исполняемого кода происходил на основе определенных условий.

Как вы уже знаете, условия в C++ имеют два возможных значения: истина (`true`) или ложь (`false`). Условие, для которого может быть вычислено значение, состоит из операторов и операндов.

Существует два вида операторов выбора: операторы `if else` и операторы `switch`.

Проверка условий с помощью оператора if

Наиболее важным из операторов выбора является, вероятно, оператор `if`. Принцип работы оператора таков: особый фрагмент кода, заключенный в оператор (он носит название *подчиненного*), выполняется только в случае, когда определенное условие (*управляющее условие*, или просто *условие*) является истинным. Общий синтаксис оператора `if`:

```
if (условие)
    подчиненныеОператоры
```

Выполнение подчиненных операторов (которых может быть произвольное количество, даже нулевое) происходит только в том случае, если *условие* вычисляется в значение `true`. Обратите внимание на отсутствие точки с запятой после условия. Обе строки обобщенного синтаксиса составляют единый оператор, поэтому точка с запятой не требуется. Кроме того, заметим, что *условие* отделяется от прочих частей оператора скобками. Эти скобки являются обязательным элементом. Если подчиненный оператор только один, синтаксис выглядит следующим образом:

```
if (условие)
    Оператор;
```

За оператором следует точка с запятой, отмечая для компилятора его окончание. Если подчиненных операторов несколько, синтаксис выглядит так:

```
if (условие)
{
     несколькоОператоров
}
```

Каждый из подчиненных операторов может быть произвольным оператором C++ и должен заканчиваться точкой с запятой. Отметим, что точка с запятой не требуется после закрывающей фигурной скобки (}). Вероятно, из главы 2 вы помните, что блок кода – это любой его фрагмент, заключенный в фигурные скобки. Блок кода может использоваться везде, где допустим единственный оператор. Оператор `if` не исключение. В этой главе вы увидите еще несколько примеров блоков кода, обращайте на них внимание.

Оба варианта синтаксиса допускают нулевое число подчиненных операторов (обратите внимание на точку с запятой в первом варианте и ее отсутствие во втором):

```
if (условие);
```

или:

```
if (условие) {}
```

Пример:

```
int swords = 9;
if(swords < 8) //условие
{
     cout<<"Число мечей меньше, чем 8";
}
```

Оператор `cout` никогда не будет выполнен, поскольку (управляющее) условие `swords < 8` (является ли значение целочисленной переменной меньшим 8?) ложно (`false`). Девять не меньше восьми. Поскольку мы хотим выполнить лишь одну строчку кода, можно переписать этот пример в альтернативном варианте, без скобок, как показано ниже:

```
int swords = 9;
if(swords < 8) //условие
     cout<<"Число мечей меньше, чем 8";
```

В данном примере, поскольку фигурные скобки не используются, оператор `if` заканчивается точкой с запятой (включительно). Операторы `if` могут существовать и без подчиненных операторов, как показано здесь:

```
if(swords < 8);
if(swords < 8){}
```

Однако причин использовать такие конструкции нет, поскольку они абсолютно бесполезны.

Как мы уже говорили, `if` является наиболее распространенным оператором управления. В сочетании с операторами `else if` и `else`, о которых речь пойдет далее, оператор `if` может стать одним из самых мощных инструментов.

В игру вступают операторы else if и else

Иногда мы выбираем альтернативный вариант действий, если определенное условие не выполняется. Например, если вы чувствуете себя плохо, то можете пойти к врачу; в противном случае – останетесь дома. Оператор, следующий за словами «в противном случае», и является альтернативным вариантом действий. Если условие, плохое самочувствие, не выполняется, вы предпримете альтернативные действия, останетесь дома.

В языке C++ слова «в противном случае» представлены оператором `else`. Оператор `else` должен входить в состав оператора `if`, он не может существовать самостоятельно. Вот его синтаксис:

```
else подчиненныеОператоры
```

Подчиненных операторов может быть произвольное число. Если этих операторов больше одного, следует использовать блок из фигурных скобок:

```
else
{
     несколькоОператоров
}
```

В этой конструкции может присутствовать любое число подчиненных операторов (даже нулевое), каждый из которых завершается точкой с запятой. Если подчиненный оператор один, блок не нужен:

```
else Оператор;
```

Объединение операторов `if` и `else` производится их последовательной записью:

```
if (условие)
     подчиненныеОператоры1
else подчиненныеОператоры2
```

Ключевое слово `else` должно следовать непосредственно за оператором `if`, в противном случае возникнет ошибка компиляции.

Пример:

```
int swords = 9;
if (swords <8)
     cout << "Мечей меньше, чем 8";
```

```
else
     cout << "Мечей не меньше, чем 8.";
```

В данном случае на экран выводится сообщение `Мечей не меньше, чем 8`**, поскольку условие оператора ложно; оператор** `else` **выполняется автоматически. Будь целочисленная переменная** `swords` **инициализирована значением 6, выполнение оператора привело бы к выводу текста** `Мечей меньше, чем 8`**, а ветвь оператора** `else` **была бы пропущена.**

Оператор, находящийся в подчинении оператора `else`**, сам может являться оператором** `if`**. Именно так возникает оператор** `else if`**, пользующийся дурной репутацией. Конструкция выглядит следующим образом:**

```
if (условие1)
     подчиненныеОператоры1
else if (условие2)
     подчиненныеОператоры2
```

Условия операторов (*условие1* **и** *условие2***) являются различными, как и наборы подчиненных операторов (***подчиненныеОператоры1* **и** *подчиненныеОператоры2***). Предшествующий блок кода выполняется по нижеприведенным правилам:**

- **Если истинно условие** *условие1***, выполняется блок операторов** *подчиненныеОператоры1***, а оставшуюся часть конструкции** `if – else if` **компьютер игнорирует.**
- **Если условие** *условие1* **ложно, а условие** `условие2` **истинно, выполняется блок операторов** *подчиненныеОператоры2***, а оставшаяся часть конструкции** `if – else if` **игнорируется.**
- **Если оба условия,** *условие1* **и** *условие2***, ложны, не выполняется ни один из блоков подчиненных операторов.**

Вложенными операторами `if` **называется конструкция, в которой один из операторов** `if` **существует внутри другого. На рис. 3.1 приведена схема выполнения для структур** `if`**,** `else if` **и** `else`**.**

```
if (условиеA)
{

подчиненныеОператорыA
}
else if (условиеB)
{

подчиненныеОператорыB
}
else
{

подчиненныеОператорыC
}
```

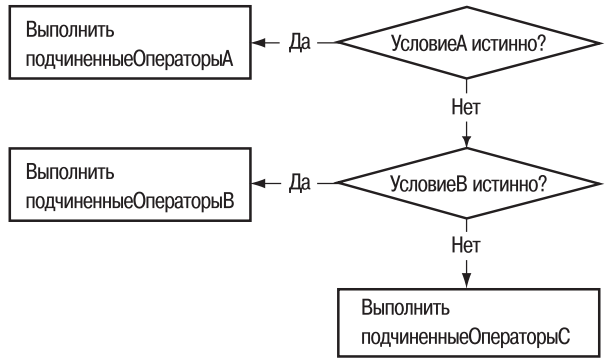


Рис. 3.1. Так работают структуры if, else if, else

Игра «Три испытания чести»

О храбрый рыцарь, перед тобой лабиринт из темных, ужасных комнат. Ты должен пройти ряд испытаний, которые приготовил для тебя злой маг, чтобы освободить девицу, которую он похитил. Если ты выдержишь все испытания чести, ты и попавшая в беду девица будете жить долго и счастливо. Первая комната полна золота. Взяв хотя бы один слиток, ты провалишь испытание, но приобретешь золото. Вторая комната полна бриллиантов. Взяв хоть один, ты покажешь свою жадность и не сможешь спасти девицу. В последней комнате необходимо защитить крестьянина от дракона. Если ты пройдешь все три испытания, злой маг отпустит девушку.

Вот так выглядит пример с применением операторов `if` и `else if`:

```
//3.1 - "Три испытания чести" - Марк Ли и Дирк Хенкеманс
//Premier Press
#include <iostream>
using namespace std;
int main(void)
{
cout << "Добро пожаловать. Три испытания чести."
     "\n3Злой маг похитил девицу, и "
     "ее судьба в твоих руках."
     "\n0Он предлагает тебе пройти три испытания "
     "чести в его Роковом Лабиринте.";
bool goldTaken, diamondsTaken, killedByDragon;
cout << "\n\nТы входишь в первую комнату. "
     "\n3Здесь столько золота, что ты едва веришь своим глазам."
     " \nВозьмешь ли ты золото (1 - да, 0 - нет)? ";
cin >> goldTaken;
if(goldTaken)
     cout << "\n3олото остается тебе, но "
          "ты провалил первое испытание. "
          "\nИгра окончена.\n\n";
else
{
     cout << "Поздравляем, "
          "ты прошел первое испытание чести!"
          "\n\nТы переходишь во вторую комнату."
          "Она полна сверкающих бриллиантов"
          "\nВозьмешь ли ты бриллианты"
          " (1 - да, 0 - нет)? ";
     cin >> diamondsTaken;
     if(diamondsTaken)
          cout << "Бриллианты достаются тебе, "
               "но ты провалил второе испытание."
               "\nТвоя жадность очевидна. "
               "\nИгра окончена.\n\n";
     else
     {
```

```
            cout << "Поздравляем, ты "
                 "прошел первые два испытания чести."
                 "\n\nТы входишь в третью комнату. "
                 "\nНа несчастного крестьянина "
                 "напал дракон!"
                 "\nДвигаться дальше, не обращая на них "
                 "внимания (1 - да, 0 - нет)?";
            cin >> killedByDragon;
            if (killedByDragon)
                 cout << "\nТы пытаешься проскользнуть мимо, "
                      "и дракон замечает твое присутствие."
                      "\nОдним огненным дуновением "
                      "он превращает тебя в пепел."
                      " Ты мертв."
                      "\nИгра окончена.\n\n";

            else
                 cout << "Поздравляем, ты "
                      "прошел все три испытания!\n\n"
                      "У выхода из лабиринта тебя "
                      "ждет злой маг.\n\n"
                      "Он пытается превратить тебя в жабу,"
                      " но тебе удается избежать этого.\n"
                      "Одним взмахом своего меча "
                      "ты прекращаешь конфликт.\n\n"
                      "Вы с девушкой живете "
                      "долго и счастливо.\n\n"
                      "Конец.\n\n";
       }
  }
  return 0;
  }
```

В этом примере основная часть программы начинается оператором `if`, который проверяет любовь игрока к золоту. Если игрок выбирает золото (и вводит цифру `1`), условие в скобках вычисляется в значение `true` (число `1` эквивалентно истине) и игрок может оставить золото себе. Но, сделав это, он не может перейти к следующему испытанию (и, как следствие, к третьему), то есть в комнату с бриллиантами. Другими словами, если первое условие истинно, оставшаяся часть структуры `if-else` игнорируется.

С другой стороны, если игрок не притрагивается к золоту, то успешно проходит первое испытание и переходит в следующую комнату. Затем, если игрок берет бриллианты, то может оставить их себе, но в этом случае не сможет спасти крестьянина и, как следствие, девушку. Если одно из условий истинно, компьютер выполняет подчиненные операторы, связанные с этим условием. Так что в нашем примере, если игрок выбирает бриллианты, компьютер выполняет код, который позволяет сохранить бриллианты, а затем выполняется переход в конец структуры оператора.

Третий оператор `if` схож с двумя предыдущими. Игрок не может спасти крестьянина, если взял золото или бриллианты. Строка `if (killedByDragons)` является примером одиночного подчиненного оператора в структуре `if`. Обратите внимание, одиночный оператор не обязательно заключать в фигурные скобки. Однако если заключить оператор `cout` в скобки, поведение программы не изменится.

Структуры завершаются операторами `else`. Операторы `else` определяют *поведение по умолчанию*: если условие соответствующего оператора `if` ложно, выполняется код блока `else`. Так что если игрок не взял золото и бриллианты, а затем победил дракона, то храбрый рыцарь может жениться на девушке и жить долго и счастливо.

Истории из жизни

Не следует недооценивать оператор `if`. Это один из наиболее важных операторов языка C++. Он реализует основную структуру, которая позволяет компьютеру выбирать вариант действий. Операторы `if` делают код программы нелинейным. Компьютер получает возможность реагировать на обстоятельства, изменяя порядок развития событий.

Вы обнаружите многочисленные применения для оператора `if` в своих программах. Например, для создания систем *ИИ* (искусственного интеллекта) или обработки ввода пользователя в компьютерных играх. Если пользователь нажимает стрелку вправо, персонаж игры должен двигаться вправо.

Условный оператор

Некоторые условия являются тривиальными, но для них приходится создавать полноценные операторы `if`. Рассмотрим такой код:

```
if (x>y)
     z=y;
else
     z=x;
```

Этот фрагмент присваивает переменной `z` значение меньшего из чисел `x` и `y`. Создание полноценной конструкции `if-else` для таких случаев может быть утомительным. К счастью, в C++ существует альтернатива – *условный оператор*. Предыдущий пример становится гораздо короче, если использовать условный оператор:

```
z = (x>y) ? y : x;
```

Общий синтаксис условного оператора:

```
условие ? выражение1 : выражение2
```

где условие и выражения являются допустимыми по правилам языка. Если условие истинно, условный оператор вычисляется в значение *выражение1*; в противном случае в значение *выражение2*. Условный оператор является выражением и может применяться везде, где допустимы выражения. Можно провести аналогии с применением оператора `if`. Однако следует помнить, что выражения не являются операторами. К примеру, невозможно выполнить такой код:

```
(x>y)? cout << "x больше, чем y." : cout << "x не больше, чем y."; //ошибка
```

Следующий пример иллюстрирует корректное использование условного оператора:

```
int a = 5;
int b = 10;
int max = (b>a) ? b : a;
int min = (b<a) ? b : a;
cout << "Максимальное значение: " << max << endl;
cout << "Минимальное значение: " << min << endl;
```

Вывод:

```
Максимальное значение: 10
Минимальное значение: 5
```

Оператор switch

Второй оператор выбора – оператор `switch`. Многие программисты избегают его использования, но в умелых руках это весьма мощный инструмент.

Представьте, что необходимо написать игру, в меню которой шесть пунктов. Можно реализовать обработку выбора с помощью шести идущих подряд операторов `if`. А еще лучше воспользоваться конструкцией `if-else if-else`. Но самое простое решение – оператор `switch`. Он позволяет сравнить значение переменной с целым рядом значений и на каждое соответствие отреагировать отдельным блоком кода. Вот общий синтаксис оператора `switch`:

```
switch(выражение)
{
case выражение1:
     подчиненныеОператоры1
     break;          (необязательно)
case выражение2:
     подчиненныеОператоры2
     break;          (необязательно)
case выражение3:
     подчиненныеОператоры3
     break;          (необязательно)
}
```

Здесь *выражение*, *выражение1*, *выражение2* и *выражение3* — это переменные или выражения, значения которых сравниваются. Строки `break` мы вскоре обсудим. Такой оператор `switch` эквивалентен следующей конструкции `if-else if`:

```
if (выражение == выражение1)
     подчиненныеОператоры1
else if (выражение == выражение2)
     подчиненныеОператоры2
else if (выражение == выражение3)
     подчиненныеОператоры3
...
```

Как видите, конструкция `if-else if` может быть гораздо менее удобной. Выполняя оператор `switch`, компьютер сравнивает *выражение* и *выражениеN* (где `N` — произвольное число) для всех операторов `case`. Если выражения равны, выполняется код оператора `case`, пока не будет встречено ключевое слово `break`. Если ни одно сравнение не вычисляется в значение `true`, оператор `switch` ничего не делает.

С ключевым словом `break` читатели еще не знакомы. Оно *завершает работу* (предписывает компьютеру перейти в конец) ближайшего охватывающего оператора `switch` или оператора цикла. Если это ключевое слово используется в любом другом контексте, при компиляции возникает синтаксическая ошибка. Строка `break` — это общий синтаксис, который показывает, что ключевое слово `break` является необязательным в данном случае. Вскоре вы узнаете, к каким последствиям приводит отсутствие этого слова, а пока будем использовать его на постоянной основе.

Итак, мы разобрались с непонятными определениями, настало время примеров:

```
int menuChoice = 3;
switch(menuChoice)
{
     case 1:
          cout << "Выбран пункт 1!" << endl;
          break;
     case 2:
          cout << "Выбран пункт 2!" << endl;
          break;
     case 3:
          cout << "Выбран пункт 3!" << endl;
          break;
     case 4:
          cout << "Выбран пункт 4!" << endl;
          break;
}
cout << "Мы находимся за пределами оператора switch!" << endl;
```

Вывод:

```
Выбран пункт 3!
Мы находимся за пределами оператора switch!
```

Как можно понять по выводу, компьютер выбрал подходящий оператор `case` и выполнил его. Все остальные операторы `case` были пропущены.

Заметим, что наличие оператора `break` в каждом операторе `case` не является обязательным (а временами из его отсутствия даже можно извлекать пользу). Рассмотрим такой пример:

```
int choice;
cout << "Пожалуйста, введите число: ";
cin >> choice;
switch (choice)
{
     case 1:
     case 2:
     case 3:
          cout << "Вы набрали 1, 2, либо 3." << endl;
          break;
     case 4:
          cout << "Вы набрали 4!" << endl;
     case 5:
          cout << "Вы набрали 4 или 5." << endl;
          break;
}
```

Вывод (пользователь ввел 1):

```
Пожалуйста, введите число: 1
Вы набрали 1, 2, либо 3.
```

Вывод (пользователь ввел 4):

```
Пожалуйста, введите число: 4
Вы набрали 4!
Вы набрали 4 или 5.
```

Вариант по умолчанию

Не всегда существует возможность предусмотреть действия для всех возможных значений в операторе `switch`, а иногда бывает желательно выполнять одно и то же действие для многих вариантов. В C++ существует оператор «иначе», который называется `default` и входит в состав оператора `switch`. Оператор варианта по умолчанию, `default`, идентичен оператору `case` за тем исключением, что не имеет связанного значения. Оператор варианта по умолчанию имеет следующий синтаксис:

```
default:
     подчиненныеОператоры
     break;
```

Когда встречается оператор варианта по умолчанию, автоматически выполняются подчиненные операторы. Для оператора `default`, в отли-

чие от case, не производится сравнение значений. Его следует располагать в конце перечня вариантов case. Пример:

```
int choice;
cout << "Пожалуйста, введите число: ";
cin >> choice;
switch (choice)
{
     default:
          cout << "\nВы набрали " << choice << endl;
}
```

Вывод:

```
Пожалуйста, введите число: 5
Вы набрали 5
```

Соблюдаем порядок действий

Если в одно выражение входит несколько операторов, компьютер должен определить порядок, в котором они выполняются. По счастью, операторы выполняются в стандартизированном порядке, который носит название *порядка действий*.

Очевидно, читатели уже знакомы с этим термином, ведь порядок действий изучается в курсе математики начальной школы:

Скобки

Возведение в степень

Деление/умножение

Сложение/вычитание

Этот перечень называется *списком приоритетов*. Наивысший приоритет имеют операции, расположенные ближе к началу списка, а наименьший – расположенные ближе к концу. Если два оператора расположены на одном уровне, они имеют *равный* приоритет. Например, в математическом выражении

```
(3+7*3)/6-2*2
```

можно воспользоваться этим списком, чтобы определить, в каком порядке вычислять выражение. Прежде всего, вычисляются скобки, а именно их содержимое. Поэтому в первую очередь будет вычислено выражение 3+7*3. В этом выражении два оператора: умножения и сложения. Поскольку умножение расположено выше по списку, эта операция выполняется первой, за ней – сложение, что в результате дает 3+21, или 24.

Полученное выражение выглядит так: 24/6-2*2. На этом этапе, учитывая равный приоритет операций умножения и деления, они выполняются слева направо. Выражение приобретает вид 4-4, и его легко вычислить. Окончательный ответ: 0.

В C++ существует аналогичный порядок действий (табл. 3.4).

Таблица 3.4. Приоритеты операторов C++

Оператор	Название
`::global`	Оператор разрешения контекста
`lvalue++, lvalue--`	Постфиксный инкремент, постфиксный декремент
`sizeof(`*`expr`*`), ++`*`lvalue`*`, --`*`lvalue`*	Оператор `sizeof`, префиксный инкремент, префиксный декремент
`*, /, %`	Умножение, деление, вычисление остатка
`+, -`	Сложение, вычитание
`<, <=, >, >=`	Меньше, меньше или равно, больше, больше или равно
`==, !=`	Равенство, неравенство
`&&`	Логическое «и»
`||`	Логическое «или»
`expr ? expr : expr`	Условный оператор

Здесь `expr` – любое допустимое выражение C++, *`lvalue`* – любой идентификатор, значение которого может изменяться, а `global` – имя любой глобальной переменной. Очевидно, эти правила более сложны, чем простой порядок действий в математике, но через некоторое время этот список станет для вас привычным. Научитесь ценить этот список, но избегайте создания кода, который бы зависел от него.

Перечисленные операции могут использоваться в любых сочетаниях в условном операторе. Условный оператор даже может выходить за пределы одной строки. Впрочем, это не очень хорошая идея, поскольку этот оператор в таком случае становится совершенно нечитаемым. Если необходимо создать многострочный условный оператор, лучше разбить его на несколько операторов. К примеру, следующий алгоритм определяет, попадает ли целое число `k` в заданный интервал. Первая граница интервала является суммой чисел от `1` до `n` (`1 + 2 + 3 + . . . + n`); формула для этой суммы – `n * (n + 1) / 2`. Вторая граница – та же сумма, но для чисел от `1` до `n + 1` (`1 + 2 + 3 + . . . + (n + 1)`); формула – `(n + 2) / 2`. Чтобы получить эту формулу, необходимо просто подставить `n + 1` вместо `n` в обычную формулу: `(n + 1) * (n + 2) / 2`.

```
if(n * (n + 1) / 2 >= k && ((n + 1) * (n + 2) / 2 ) <=  k)
//определяет, попадает ли число k в интервал
//сумм  n и n + 1
{
     cout << "k принадлежит интервалу сумм n и n+1";
}
```

И хотя этот пример не является длинным, он очень насыщен математически, а значит, его лучше разбить на несколько строк. Обратившись к списку приоритетов, мы увидим, что в этом условии последним вычисляется оператор `&&`. Значит, условие можно упростить до вида `A&&B`, где `A = (n * (n + 1) /2) >= k`, а `B = ((n + 1)*(n + 2) /2) <= k`. Это очень эффективный способ упрощать выражения. А вот результат для нашего фрагмента кода:

```
int a = (n * (n + 1) /2) >= k;
int b = ((n + 1) * (n + 2) /2) <= k;
if (a && b)
{
     cout << "k принадлежит интервалу сумм n и n+1";
}
```

Переходим к операторам циклов

Оператор цикла – это управляющая структура, выполняющая фрагмент кода до тех пор, пока не перестанет действовать определенное условие. Концепцию можно сделать более понятной на примере легенды о Геракле. Чтобы продолжать спокойно жить, Гераклу предстояло совершить двенадцать подвигов (то есть оператор цикла должен был бы выполниться несколько раз). Совершив подвиги, он снова получил возможность жить так, как ему хочется. В программировании эти подвиги могут быть совершены не один за другим, а в цикле; как только цикл закончит с двенадцатым подвигом, компьютер продолжит выполнение оставшегося кода программы.

Цикл while

Пример с Гераклом можно реализовать посредством цикла `while`. Цикл `while` выполняет подчиненные операторы по кругу, пока определенное условие не становится ложным. Вот синтаксис цикла `while`:

```
while(условие)
     подчиненныеОператоры
```

Условие проверяется при каждом выполнении цикла. Как только оно перестает быть истинным (`true`), цикл завершается. Заметим, что если условие ложно с самого начала, подчиненные операторы не будут выполнены ни разу. Также следует помнить, что циклы должны рано или поздно завершаться. Избегайте *бесконечных циклов* (циклов, предусловия которых никогда не становятся ложными).

Мы готовы совершить двенадцать подвигов за Геракла (речь о которых шла в предыдущем разделе). В следующем примере мы создадим программу с циклом `while`. Необходимо ограничить цикл `while` определенным условием: число подвигов должно быть не больше двенадцати.

```
//3.2 - Двенадцать подвигов Геракла - Дирк Хенкеманс и Марк Ли
//Premier Press
#include <iostream>
using namespace std;      //introduces namespace std

int main( void )
{
//эта переменная хранит число совершенных подвигов.
int taskNumber = 0;

     while(taskNumber < 12) // пока taskNumber не станет>= 12
     {
          taskNumber++;
          cout<<"Геракл совершил "
               << taskNumber << " подвиг." << endl;
     }
     return 0;
}
```

Вывод:

```
Геракл совершил 1 подвиг.
Геракл совершил 2 подвиг.
Геракл совершил 3 подвиг.
Геракл совершил 4 подвиг.
Геракл совершил 5 подвиг.
Геракл совершил 6 подвиг.
Геракл совершил 7 подвиг.
Геракл совершил 8 подвиг.
Геракл совершил 9 подвиг.
Геракл совершил 10 подвиг.
Геракл совершил 11 подвиг.
Геракл совершил 12 подвиг.
```

Обратите внимание, что окончательное значение переменной `taskNumber` – 12, а не 13. Уже при этом условие цикла становится ложным. Кроме того, заметим, что для `taskNumber` выполняется приращение на каждом проходе цикла. Это позволяет не только отслеживать число совершенных подвигов, но и гарантировать завершение цикла в определенный момент. Схема работы оператора `while` приведена на рис. 3.2.

```
while (условие)
{
   подчиненныеОператоры
}
```

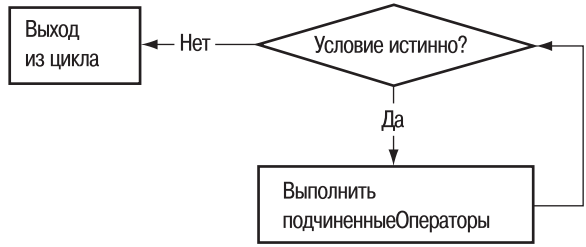


Рис. 3.2. Так работает оператор while

И снова об операторах инкремента и декремента

Предыдущий пример можно сделать более кратким и выразительным, переместив оператор ++ в условие цикла. Как мы уже говорили, условие может содержать произвольное число операторов. Это верно и для условия в «Двенадцати подвигах Геракла». Но прежде следует узнать о постфиксных и префиксных операторах. Для инкремента и декремента символ оператора может предшествовать операнду (префиксная запись) или следовать за ним (постфиксная запись):

```
операнд$$ //постфиксный
$$операнд //префиксный
```

Здесь символы $$ представляют оператор. Эти варианты не только записываются различным образом, они имеют различный смысл. В постфиксной записи операция выполняется для операнда после извлечения его значения, в префиксной – до извлечения значения. Пример:

```
int x = 5;
int y = 6;
int z = y++; // z = 6, y = 7
z = ++x; // z = 6, x = 6
```

А вот так мы можем воспользоваться приращением в примере для Геракла:

```
int taskNumber = 0;
while(taskNumber++ < 12)
{
     cout<<"Геракл совершил "
          <<taskNumber<<" подвиг."<<endl;
}
```

Вывод:

```
Геракл совершил 1 подвиг.
Геракл совершил 2 подвиг.
...
Геракл совершил 11 подвиг.
Геракл совершил 12 подвиг.
```

Вы можете спросить, в какой момент происходит приращение `taskNumber`: до или после сравнения переменной с числом `9`? Ответ таков: если ++ следует за переменной, приращение происходит после вычисления всего выражения. Если ++ предшествует переменной, приращение выполняется до всех остальных вычислений.

Вот еще один пример использования префиксных и постфиксных операторов:

```
//3.3 - ++, пример программы - Дирк Хенкеманс и Марк Ли
//Premier Press
#include <iostream>
```

```
using namespace std;

int main( void )
{
     int var1 = 0, var2 = 0;

     cout << var1++ << endl;
     cout << ++var2;

     return 0;
}
```

Вывод:

```
0
1
```

Хотя на момент завершения программы операторы `var1` и `var2` имеют значение 1, можно видеть, что выполнение оператора `var1++` происходит после вывода значения переменной `var1`, а оператор `++var2` выполняет приращение `var2` до вывода значения.

Оператор `--` работает аналогичным образом. Предварите переменную этим оператором, чтобы выполнить его ранее других, либо поместите оператор после переменной, чтобы выполнить его после вычисления всего выражения.

Цикл do while

Цикл `do while` сходен с циклом `while` с той разницей, что проверка условия происходит после завершения итерации цикла. Таким образом, цикл всегда выполняется по меньшей мере один раз.

Синтаксис цикла `do while`:

```
do
{
     подчиненныеОператоры
}
while(условие);
```

Не забывайте (хотя это очень легко сделать) завершать условие точкой с запятой. Цикл `do while` выполняется, пока не станет ложным условие, точно так же, как и цикл `while`. Однако цикл `do while` в первый раз выполняется еще до проверки условия, поэтому при любых обстоятельствах будет выполнен по меньшей мере один раз.

Хорошим примером применения цикла `do while` является реализация меню для игры, как мы покажем далее. Необходимо отображать меню в цикле еще до того, как пользователь сможет выбрать один из пунктов. Затем, пока пользователь не введет корректную информацию, меню будет отображаться снова и снова. К примеру, если пункты меню пронумерованы цифрами от 1 до 4, цикл должен повторяться, пока

пользователь не введет число между 0 и 5 (допустимый номер пункта меню). После получения допустимого номера цикл завершает работу.

```
//3.4 - Простое меню - Дирк Хенкеманс - Premier Press #include<iostream>
using namespace std;
int main(void)
{
     cout<<"Ваш отряд двигался по холмам "
                    << "в предместье Кье'лла \n"
          <<"и попал в засаду "
                    << "злых негодяев!!! \n \n";

     int response = 0;
     do
     {

          cout << "Какие действия вы предпримете? \n"
               << " 1) В атаку на злых негодяев!!! \n"
               << " 2) Сбежать \n"
               << " 3) Попытаться поговорить с негодяями \n";
          cin>>response;
     }
     while (response < 1 || response > 3);
     if (response == 1)
     {
          cout << "Сражение затянулось до утра, "
               << "и к восходу солнца уже трудно было \n"
               << "отличить живых от мертвых!\n";
     }
     else if (response == 2)
     {
          cout << "Вы сбежали от негодяев "
               << "под защиту деревьев, и никогда больше \n"
               << "с ними не встретитесь.\n";
     }
     else
     {
          cout<<"Вы пытаетесь поговорить с ними, "
               << "но они, похоже, не очень хотят слушать. \n"
               <<"Они забирают все деньги "
              << "и удаляются довольные, оставив вас у разбитого корыта.\n ";
     }
     return 0;
}
```

Программа отображает краткое введение и затем предоставляет пользователю выбор из трех пунктов меню. Если пользователь выбрал несуществующий пункт, меню отображается заново, и так до тех пор, пока не будет введен правильный номер. Каждый из пунктов меню связан с независимой концовкой игры.

Меню являются основой для большинства текстовых программ, поскольку являются удобным средством организации взаимодействия с пользователем. В главе 4 «Пишем функции» вы научитесь создавать каркас меню для текстовых приключений, который может использоваться повторно.

Оператор for

Среди операторов, с которыми читателям предстоит познакомиться, `for` входит в число самых гибких. Оператор `for` позволяет организовать цикл с удобным управлением.

Ниже приведен синтаксис оператора `for`:

```
for (инициализация; условие; выражение)
     подчиненныеОператоры
```

Здесь *инициализация* – любой допустимый оператор инициализации переменной или выражение, *условие* – любое допустимое условие, а *выражение* – любое допустимое выражение.

Такое определение может оказаться чрезмерно сложным для новичка. Но в действительности оператор `for` устроен очень просто. Взгляните на этот пример:

```
for(int count = 0; count < 10; count++)
{
     cout << count << " ";
}
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9
```

На экране отображаются числа от `0` до `9`. И хотя этот код на первый взгляд кажется сложным, на самом деле он таковым не является. Ниже перечислены шаги, выполняемые для оператора `for`:

1. Выполнить оператор инициализации.
2. Проверить условие (если условие ложно (`false`), работа оператора прекращается).
3. Выполнить *подчиненныеОператоры*.
4. Выполнить *выражение*.
5. Проверить *условие*. Если условие истинно (`true`), перейти к шагу 3.

Конец оператора `for`.

Первая часть `for` – оператор *инициализации*. В отличие от оставшейся части цикла, он выполняется только один раз в начале оператора `for`. Инициализация может быть представлена любой формой инициализации переменной, которых существует три. Первую форму мы наблюдали в предыдущем примере (`int count = 0`). Она включает объявление

и инициализацию переменной. Вторая форма устанавливает значение ранее объявленной переменной (к примеру, `count = 0`). Третья форма – пустой раздел инициализации. Проиллюстрируем все три формы таким примером:

```
for (int count = 0; count < 10; count++) //первая форма
     cout << count << " ";
cout << endl;

int count;
for (count = 0; count < 10; count++) //вторая форма
      cout << count << " ";
cout << endl;

int index = 0;
for (; index < 10; index++) //третья форма
     cout << index << " ";
cout << endl;
```

Вывод:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Этот пример иллюстрирует и другой момент. Обратите внимание, переменная `count` объявлена дважды. Расположение объявлений делает такое действие допустимым. Второе объявление принадлежит локальной области видимости, а значит, переменная существует до конца функции или блока кода (в данном случае – до конца фрагмента кода). Но первое объявление не является локальным. Переменные, объявленные в разделе инициализации, имеют ограниченную область видимости. Область видимости таких переменных простирается до конца оператора `for`, следовательно, как только оператор заканчивается, переменная, объявленная в разделе инициализации, прекращает свое существование. Таким образом, второе объявление не создает конфликта, поскольку первая переменная `count` к этому моменту уже не существует.

Второй важной частью оператора `for` является *условие*. Оно имеет тот же смысл, что условия во всех прочих управляющих операторах. Оно проверяется сразу после выполнения инициализации, а затем каждый раз перед началом цикла. Как только условие становится ложным, оператор `for` завершается.

Условие можно опускать, как показано в следующем примере:

```
for (int index = 0; ; i++)
     cout << index;
```

Данный фрагмент кода реализует бесконечный цикл. Отсутствие условия равноценно наличию условия, которое всегда вычисляется в значение `true`.

Третья часть оператора – *выражение*. Выражение вычисляется сразу после выполнения подчиненных операторов. Таким образом, если подчиненные операторы не выполняются (условие ложно изначально), выражение также не вычисляется.

Выражение тоже можно опускать, как показано в следующем примере:

```
for (int count = 0; count < 10; )
{
     cout << count;
     count++;
}
```

Обратите внимание, что присутствие выражения (`count++`) в конце списка подчиненных операторов эквивалентно его указанию в операторе `for`.

Наиболее распространенное применение оператора `for` – многократное выполнение определенной задачи. Прежде всего, следует объявить переменную и присвоить ей начальное значение в операторе инициализации. Выражение содержит оператор инкремента или декремента для переменной, объявленной в разделе инициализации; это гарантирует, что правильно поставленное условие станет ложным после определенного числа повторений цикла. Избегайте изменения значения переменной-«счетчика» в теле цикла (в подчиненных операторах). Чаще всего в качестве имени такой переменной используются идентификаторы `count` и `index` (для краткости – `c` или `i`).

Вложенная структура

По мере роста сложности программ появится необходимость во вложенных управляющих конструкциях. *Вложенные* конструкции программирования содержатся внутри других конструкций. К примеру, может появиться необходимость использовать оператор `if` внутри оператора `for`. Любой управляющий оператор может на законных основаниях помещаться внутри другого управляющего оператора. Вот пример цикла `do while`, вложенного в оператор `if`:

```
if (choice == displayMenu)
{
     do
     {
     //displayMenu
     }
     while(!valid Input);
}
```

Создание таблицы умножения требует двух вложенных циклов. В следующем примере вложенный цикл `for` полностью выполняется на каждой итерации внешнего цикла `for`. Этот пример иллюстрирует один из способов создания таблицы умножения:

```
//3.5 - Таблица умножения - Дирк Хенкеманс
//Premier Press
#include<iostream>
using namespace std;

int main(void)
{
     cout << "Таблица умножения:" << endl
          << "  1\t2\t3\t4\t5\t6\t7\t8\t9" << endl
          << "  ------------------"
          << "---------------------"
          << "------------" << endl;
     for(int c = 1; c < 10; c++)
     {
          cout << c << "| ";
          for(int i = 1; i < 10; i++)
          {
          cout << i * c << '\t';
          }
          cout << endl;
     }
     return 0;
}
```

Вывод:

```
Таблица умножения:
     1     2     3     4     5     6     7     8     9
  ------------------------------------------------------------
1|   1     2     3     4     5     6     7     8     9
2|   2     4     6     8    10    12    14    16    18
3|   3     6     9    12    15    18    21    24    27
4|   4     8    12    16    20    24    28    32    36
5|   5    10    15    20    25    30    35    40    45
6|   6    12    18    24    30    36    42    48    54
7|   7    14    21    28    35    42    49    56    63
8|   8    16    24    32    40    48    56    64    72
9|   9    18    27    36    45    54    63    72    81
```

Очевидно, что вложенные управляющие операторы являются очень полезными для решения разнообразных задач.

Прыгаем по коду: операторы ветвления

Операторы ветвления позволяют перемещаться по коду, изменяя естественный порядок выполнения программы. Четыре оператора ветвления описаны в табл. 3.5.

Мы уже определили назначение оператора `break` (раздел «Оператор switch» выше по тексту главы). Для операторов цикла `break` является альтернативным способом завершения.

Таблица 3.5. Операторы ветвления	
Ключевое слово	**Назначение**
`break`	Завершает работу оператора `switch` или оператора цикла, в который непосредственно заключен
`continue`	Принудительно начинает следующую итерацию оператора цикла, в который непосредственно заключен
`goto`	Осуществляет переход в определенную точку кода
`return`	Завершает работу функции и возвращает значение (более подробно в главе 4)

Про оператор `return` мы расскажем в главе 4, которой он больше соответствует по теме.

Оператор continue

Оператор `continue` предписывает продолжить выполнение кода с начала следующей итерации оператора цикла. Этот оператор делает программирование на C++ более комфортным. Иногда лишь с его помощью можно заставить фрагмент кода работать так, как нужно. В отсутствие `continue` пришлось бы полностью переписывать крупные фрагменты кода. Пример:

```
int input;
for ( ; ; ) //'беспрестанно' - бесконечный цикл
{
     cout << "\nПожалуйста, введите число от 1 до 10: ";
     cin >> input;
     if (input < 1 || input >9) continue;
     break;
}
```

Вывод:

```
Пожалуйста, введите число от 1 до 10: 10
Пожалуйста, введите число от 1 до 10: -50
Пожалуйста, введите число от 1 до 10: 5
```

В этом примере выполнение оператора `for` продолжается, пока пользователь не введет допустимое число (из интервала от 1 до 10). Оператор `continue` гарантирует, что оператор `break` не будет выполнен, пока условие не станет ложным.

Оператор goto

Многие программисты отрицательно относятся к оператору `goto`. Они считают, что он приводит к созданию сложного для восприятия кода и является решением только для ленивых. Очень часто так оно и есть, но оператору `goto` тоже можно найти применение.

goto осуществляет переход в указанную точку кода, которая отмечена специальной меткой. *Метка* – это идентификатор, за которым следует двоеточие. Она может размещаться в начале любого оператора (даже пустого). Синтаксис для метки:

```
метка: оператор
```

Оператор должен заканчиваться точкой с запятой. Синтаксис оператора goto:

```
goto метка;
```

Компьютер перейдет к строке кода, в начале которой расположена метка. Пример:

```
int i;
int j;
int input;
cout << "Пожалуйста, введите число: ";
cin >> input;
for (j = 0; j < 10; j++)
     for (i = 0; i < 10; i++)
          if (input == i*j) goto FoundIt;
FoundIt: cout << "\nВы набрали число " << i*j;
```

Вывод:

```
Пожалуйста, введите число: 10
Вы набрали число 10
```

В этом примере происходит выход из вложенных циклов, как только произведение i*j станет равным введенному числу. И хотя эта программа имеет изъян – если ввести простое число или число, большее 81, программа сообщит, что набрано число 100 – она вполне служит нашим целям.

Одним из практичных применений оператора goto является принудительный выход за пределы вложенных конструкций, поскольку оператор break позволяет завершить выполнение лишь одной управляющей конструкции, а именно той, в которую непосредственно заключен.

Создаем случайные числа

Случайные числа несовместимы с компьютерами, однако можно создавать псевдослучайные числа при помощи сложных математических формул. Здесь мы не будем вдаваться в подробности этих формул, потому что они слишком сложны для новичка, а применять их можно и без такого специального знания.

Случайные числа создаются на основе значения инициализации (random seed). (*Значение инициализации* – это число, которое использует-

ся в работе генератора случайных чисел.) Случайный инициализатор обычно выбирается на основе положения секундной стрелки системных часов компьютера.[1]

Такой вариант является идеальным для случайного инициализатора, поскольку значение невозможно предсказать. А вероятность того, что программа будет дважды запущена за одну секунду, настолько мала, что случайные числа каждый раз будут другими.

Процесс генерации случайных чисел состоит из двух этапов: инициализации и собственно генерации чисел. Чтобы получить доступ к секундной стрелке компьютерных часов, необходимо включить часть стандартной библиотеки C++: `<ctime>` или `<time.h>`:

```
#include <ctime>
```

Генератор случайных чисел также входит в стандартную библиотеку, поэтому следует включить также `<cstdlib>` или `<stdlib.h>`:

```
#include <cstdlib>
```

Для инициализации генератора случайных чисел применяется функция `srand()` (сокращение от seed random; функции описаны в главе 4). В скобках передается число, с помощью которого мы хотим инициализировать генератор. Число может быть любым, но в данном случае мы используем функцию `time()`. Функция `time()` позволяет получить число секунд, прошедших с 1 января 1970 года; это может показаться странным, на таков стандартный способ обращения к секундной стрелке часов компьютера. Инициализация генератора происходит таким образом:

```
srand(time(0));
```

Значение `(0)` выходит за пределы темы этой главы, поэтому мы не будем о нем говорить. Просто убедитесь, что эта строка выполняется лишь один раз. Не следует повторно инициализировать генератор, поскольку это может привести к потере случайных качеств (выполнение цикла часто занимает гораздо меньше одной секунды).

После инициализации генератор готов к использованию. Чтобы сгенерировать случайное число, воспользуйтесь функцией `rand()`. Эта функция возвращает большое случайное число, например 2453. Вот так выглядит вызов функции `rand()` в коде:

```
cout << rand();
```

---

[1] На самом деле для инициализации обычно используется не «положение секундной стрелки» (число от 0 до 59), а текущее системное время в секундах (очень большое число, например 1 022 153 141). Об этом авторы еще скажут ниже – простим им эту маленькую неточность. – *Примеч. ред.*

Чтобы сгенерировать случайное число из определенного интервала (скажем, от 1 до 10), можно применить такую формулу:

```
rand() % (max - min + 1) + min;
```

Здесь *max* – максимальное значение, а *min* – минимальное. К примеру, чтобы сгенерировать число из интервала от 10 до 20, воспользуйтесь следующим кодом:

```
int num = rand() % 11 + 10; // (20 - 10 + 1) = 11
```

Формула работает, поскольку `rand % 11` принимает значения от 0 до 10. Добавление числа 10 дает число из интервала от 10 до 20.

Функция `rand()` весьма полезна, она является составной частью нашей следующей игры.

Игра «Угадай число»

До сих пор вы создавали игры, которые не меняли поведения, реагируя на ввод пользователя. Настало время применить случайные числа, операторы выбора и цикла для создания динамических, отзывчивых программ.

В оставшейся части книги вы будете создавать все более сложные игры, реагирующие на ввод пользователя. Первой из них станет игра «Угадай число». Программа выбирает случайное число от 1 до 100. Пользователь должен отгадать число, а программа сообщает, является ли введенное число слишком большим, слишком маленьким либо исходным. Как только пользователь отгадает число, программа завершает работу, отображая при этом общее число попыток.

```
//3.6 - Игра "Угадай число"- Дирк Хенкеманс
//Premier Press
#include<iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

int main(void)
{
     cout<<"Добро пожаловать в игру \"Угадай число\"!!! \n";
     cout<<"Я загадал число от 1 до 100. \n \n";

srand(time(0));
     //хранит случайное число
     int numPicked = rand() % 100 + 1;
     int guess = 0; //хранит число, введенное пользователем
     int guessNum;  //хранит число попыток

     for(guessNum = 0; guess != numPicked; guessNum++)
     {
          cout<<"Итак, какое это число? \n";
```

```
            cin>>guess;

            if(guess < numPicked)
                cout<<"\n3агаданное число больше!!! \n \n";
            else if(guess > numPicked)
                cout<<"\n3агаданное число меньше!!! \n \n";
        }
        cout<<"\nТочно!!! \n"
            <<"Ты угадал за "<<guessNum<<" попытки.";

        return 0;
    }
```

Вывод:

```
Добро пожаловать в игру "Угадай число"!!!
Я загадал число от 1 до 100.

Итак, какое это число?
50
Загаданное число больше!!!

Итак, какое это число?
75
Загаданное число меньше!!!

Итак, какое это число?
63
Точно!!!
Ты угадал за 3 попытки.
```

Пишем игру «Римский полководец»

Замечтавшись в облаках программирования, вы повстречали императора Рима. Он отчаянно нуждается в полководце, который возглавил бы наступление на германские войска. Справитесь ли вы с этой задачей?

Разумеется, вы соглашаетесь, поскольку это великолепная возможность применить на практике новые знания, связанные со случайными числами, циклами `while` и `do while`, операторами `if` и `switch`, а также условным оператором.

Наберите следующий код в редакторе исходных текстов своего компилятора и выполните программу. Изучите каждый управляющий оператор в коде. Они для вас столь же понятны, как текст на русском языке? Если нет, то очень скоро станут.

```
//3.7 - Игра "Римский полководец" - Марк Ли и Дирк Хенкеманс
//Premier Press
#include <iostream>
#include <string>
#include <ctime>
#include <cstdlib>
```

```
using namespace std;      //используем пространство имен std
int main( void )
{
     srand(time(0)); //инициализация генератора случайных чисел

     string name; //хранит имя игрока
     bool end = false;  //используется для проверки того,
                        // выбрал ли пользователь выход из игры
     bool lost; //используется для проверки того, проиграл ли пользователь
игру

     int menu_choice; //хранит пункт меню, выбранный пользователем

     //начальные силы игрока
     int archers = 50;
     int catapults = 25;
     int swordsmen = 100;

     //начальные силы германских войск (случайные числа)
     // случайное число из интервала от  70 до 20
     int g_archers = rand() % (51) + 20;
     int g_catapults = rand() % (41) + 10; //из интервала от 50 до 10
     //из интервала от 150 до 50
     int g_swordsmen = rand() % (101) + 50;

     //эти переменные хранят номера, соответствующие пунктам меню
     int archers_menu, catapults_menu, swordsmen_menu;
     int fight_menu;

     cout << "Добро пожаловать, искатель приключений, как твое имя?\n";
     cin >> name;
     cout << "Что ж, " << name
          << ", добро пожаловать в игру \"Римский полководец\".\n"
          << "\nТебе предстоит возглавить войска Рима в битве "
          << " против Германии.";
     while (!end) //главный цикл игры
     {
          //переменные, хранящие число боевых единиц, отправленных в атаку
          int archers_sent=0, catapults_sent=0;
          int swordsmen_sent=0;
          cout << "\nВ твоем распоряжении " << archers
               << " лучников, " << catapults
               << " катапульт, а также "
               << swordsmen << " легионеров.\n"
               << "\nГермания выставила " << g_archers
               << " лучников, "
               << g_catapults << " катапульт, а также "
               << g_swordsmen
               << " легионеров.\n";
          do //цикл подготовки к сражению
          {
               //отслеживаем, какие пункты меню были
               //использованы
```

```
int i = 1;
if (archers > 0 &&
     ((archers - archers_sent) != 0))
{
     archers_menu = i;
     cout << "[" << i << "] Выдвинуть лучников\n";
     i++;
}
else archers_menu = 0;
if (catapults > 0 &&
     ((catapults - catapults_sent) != 0))
{
     catapults_menu = i;
     cout << "[" << i << "] Выдвинуть катапульты\n";
     i++;
}
else catapults_menu = 0;
if (swordsmen > 0 &&
     ((swordsmen - swordsmen_sent) != 0))
{
     swordsmen_menu = i;
     cout << "[" << i << "] Выдвинуть легионеров\n";
     i++;
}
else swordsmen_menu = 0;
fight_menu = i;
cout <<"["<< i <<"] В атаку!\n";

cin >> menu_choice;
if (menu_choice == archers_menu)
{
     do {
          cout << "Сколько лучников"
               " отправить в наступление?\n";
          cin >> archers_sent;
     }while (!(archers_sent > -1
          && archers_sent <= archers));
}
else if  (menu_choice == catapults_menu)
{
     do {
          cout << "Сколько катапульт"
               " отправить в наступление?\n";
          cin >> catapults_sent;
     }while (!(catapults_sent > -1 &&
          catapults_sent <= catapults));
}
else if (menu_choice == swordsmen_menu)
{
     do {
          cout << "Сколько легионеров"
```

```
                    " отправить в наступление?\n";
               cin >> swordsmen_sent;
          }
          while (!(swordsmen_sent > -1 &&
               swordsmen_sent <= swordsmen));
     }
}
//конец цикла подготовки к сражению
while (menu_choice != fight_menu);

cout << "\nСражение началось...\n";

int archers_dead, catapults_dead, swordsmen_dead;
int g_archers_dead, g_catapults_dead;
int g_swordsmen_dead;

//каждая катапульта убивает 2 лучников
archers_dead = 2 * g_catapults;
//каждый легионер разрушает  1 катапульту
catapults_dead = g_swordsmen;
//каждый лучник убивает 3 легионеров
swordsmen_dead = 3 * g_archers;

g_archers_dead = 2 * catapults_sent;
g_catapults_dead = swordsmen_sent;
g_swordsmen_dead = 3 * archers_sent;

//число боевых единиц
//не должно стать меньше 0.
archers = (archers_dead < archers) ?
     archers - archers_dead : 0;
catapults = (catapults_dead < catapults) ?
     catapults - catapults_dead : 0;
swordsmen = (swordsmen_dead < swordsmen) ?
     swordsmen - swordsmen_dead : 0;

g_archers = (g_archers_dead < g_archers) ?
     g_archers - g_archers_dead : 0;
g_catapults = (g_catapults_dead < g_catapults) ?
     g_catapults - g_catapults_dead : 0;
g_swordsmen = (g_swordsmen_dead < g_swordsmen) ?
     g_swordsmen - g_swordsmen_dead : 0;

cout << "Битва была долгой. "
      << archers_dead << "лучников погибло.\n"
      << catapults_dead << " катапульт разрушено.\n"
      << swordsmen_dead << " легионеров мертвы.\n";
//если армия игрока разбита, он проиграл
if ((archers + catapults + swordsmen) == 0)
     end = lost = true;
//если армия германцев разбита, игрок победил
else if ((g_archers + g_catapults
     + g_swordsmen) == 0)
{
```

```
                end = true;
                lost = false;
            }
        } //конец главного цикла игры

        //вывести соответствующий результатам текст
        if (lost)
        {
            cout << "\nТы потерпел поражение. Попробуй еще раз.\n";
            return 0;
        }
        cout << "\nПоздравляем с победой!\n";
        return 0;
}
```

Вывод:

```
Добро пожаловать, искатель приключений, как твое имя?
Маркус
Что ж, Маркус, добро пожаловать в игру "Римский полководец".

Тебе предстоит возглавить войска Рима в битве против Германии.
В твоем распоряжении 50 лучников, 25 катапульт, а также 100 легионеров.

Германия выставила 61 лучников, 50 катапульт, а также 52 легионеров.
[1] Выдвинуть лучников
[2] Выдвинуть катапульты
[3] Выдвинуть легионеров
[4] В атаку!
1
Сколько лучников отправить в наступление?
50
[1] Выдвинуть катапульты
[2] Выдвинуть легионеров
[3] В атаку!
1
Сколько катапульт отправить в наступление?
25
[1] Выдвинуть легионеров
[2] В атаку!
1
Сколько легионеров отправить в наступление?
100
[1] В атаку!
1

Сражение началось...
Битва была долгой. 100 лучников погибло.
52 катапульт разрушено.
183 легионеров мертвы.

Ты потерпел поражение. Попробуй еще раз.
```

Резюме

В этой главе вы узнали, как создавать программы с ветвлениями, реагирующие на ввод пользователя. Путешествие в неизведанные земли продолжилось логическими операторами, операторами выбора и цикла. Эти уроки рассказывают о важной части программирования и позволят значительно повысить функциональность ваших программ.

Задания

1. Создайте условный оператор (`if`), который присваивал бы `x` значение `x/y`, если `y` не равно `0`.
2. Создайте цикл `while`, вычисляющий сумму положительных целых чисел от 1 до некоторого числа *n* (проверьте результат по формуле `n*(n+1)/2`).
3. Создайте условный оператор, который присваивал бы `x*y` для четного `x`, в противном случае для нечетного `x` и `y`, не равного `0`, присваивал бы `x/y`; наконец, если ни одно из предыдущих условий не вычисляется в `true`, выводил бы на экран сообщение, что значение `y` равно `0`.

4

Пишем функции

Если вы читали главы в порядке следования, то продвинулись уже довольно далеко в своих изысканиях. Сделайте небольшой перерыв, чтобы информация улеглась в голове, приготовьте себе чашку черного кофе, устройтесь в любимом кресле и соберитесь с духом. Впереди вас ждет замечательный путь.

В этой главе вы изучите:

- Создание функций
- Ключевое слово `void`
- Значения аргументов по умолчанию
- Области видимости переменных
- Функцию `main`
- Разработку макроопределений

Разделяй и властвуй

Вообразите себя полководцем Римской империи, возглавляющим наступление на полчища варваров Германии. Любой из ваших гениальных стратегов подтвердит, что атаковать всю армию противника сразу – идея стратегически не слишком мудрая. Гораздо более умно будет рассечь армию противника на несколько частей и атаковать каждую часть по отдельности. Таким образом ваша армия получает значительный перевес, а Римская империя может спокойно расти дальше.

То же относится и к программированию на C++. Если необходимо создать крупное приложение в определенный срок, то простейший способ справиться с заданием – разделить его на несколько более мелких задач и решать каждую в отдельности. Любой математик скажет – сведите проблему к сочетанию уже решенных, и от задачи ничего не останется. Такова идея применения функций.

Функции позволяют программисту разделять код на многочисленные фрагменты, с которыми удобно работать. Кроме того, использование функций позволяет не повторять фрагменты кода раз за разом, а создавать их лишь единожды, а затем просто использовать. Функции избавляют от необходимости искать решение каждый раз, когда встретится похожая задача, они просто выполняют свою работу, решая эти задачи за вас. Такие решения могут использоваться по мере необходимости.

Можно, например, написать функцию `getWindowSize`, вычисляющую размер определенного окна. Как только понадобится выяснить размер любого другого окна, эту задачу вы поручаете уже существующей функции. Помните, что функция и код, поручающий ей выполнение задачи, должны говорить на общем языке. В нашем примере функция `getWindowSize` должна знать, о каком из окон идет речь, и должна вернуть вызывающему коду размер этого окна.

Прежде чем двинуться дальше, рассмотрим определения терминов этой главы (некоторые из них на данном этапе могут быть не очень понятными, но к концу главы все прояснится):

- **Аргументы.** Фрагменты информации (данных), которые *вызывающая процедура* передает функциям (определения вызывающей процедуры и функции даны ниже). Аргумент также может называться *параметром*.
- **Список аргументов.** Список аргументов, которые должны быть переданы, чтобы состоялся вызов определенной функции.
- **Блок программы (или блок кода).** Любой код, заключенный в фигурные скобки.
- **Вызов функции.** Передача функции определенного поручения.
- **Вызывающая процедура.** Любой фрагмент кода, в котором происходит вызов функции.
- **Функция.** Сегмент кода, выполняющий определенную задачу.
- **Возвращаемое значение.** Любое значение, возвращаемое функцией вызывающей процедуре. Заметим, что каждая функция может иметь только одно возвращаемое значение (рис. 4.1).

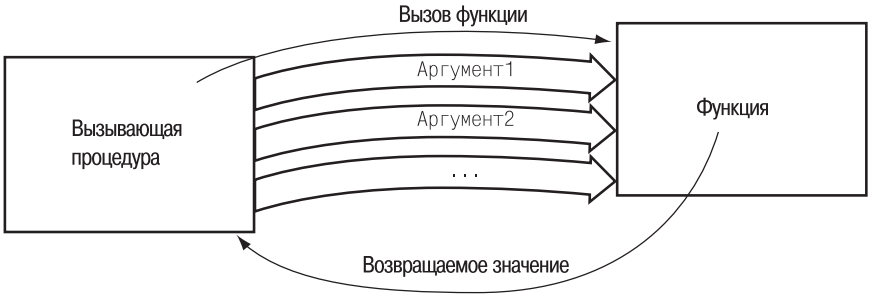


***Рис. 4.1.** Концептуальная схема процесса вызова функции*

Изучаем синтаксис функций

В этом разделе мы перейдем от теории к практике: вы научитесь создавать код для работы с функциями. Прежде всего, следует овладеть искусством применения синтаксиса функций. Пока что следует знать лишь о трех элементах работы с функциями. Эти три элемента станут основанием для взаимодействия программ с функциями и создания объявлений функций:

- Объявление функции
- Определение функции
- Вызов функции

В следующих разделах мы изучим каждый из этих элементов.

Объявление функции

Объявление функции сообщает компилятору, что функция существует. Вы предоставляете компилятору имя функции, ее аргументы, а также тип возвращаемого значения. Объявление функции также называют *прототипом*, поскольку оно является моделью функции. Общий синтаксис объявления функции:

```
тип_возвращаемого_значения имя_функции(список_аргументов);
```

Здесь *тип_возвращаемого_значения* – это тип данных, которые возвращает функция, *имя_функции* – соответственно имя функции, а *список_аргументов* – список аргументов, которые необходимо указать при вызове функции. Не забывайте о точке с запятой в конце объявления.

список_аргументов может быть пустым, если аргументы не нужны:

```
тип_возвращаемого_значения имя_функции();
```

В противном случае список аргументов представляет собой список объявлений переменных, разделенных запятыми:

```
тип_возвращаемого_значения имя_функции(тип_арг1 имя_арг1, тип_арг2 имя_арг2,
...);
```

Обратите внимание – каждое объявление аргумента является просто объявлением переменной (синтаксис должен быть вам знаком, он описан в главе 2 «Продолжаем погружение: переменные»).

Пример – функция сложения

Идея функции сложения относительно проста – взять два числа, сложить их, вернуть результат. И хотя в данном примере мы будем работать с целыми числами, функцию можно реализовать для любого численного типа данных.

Чтобы создать объявление функции, необходимо наличие следующей информации:

- Имя функции
- Тип возвращаемого значения для функции
- Список аргументов функции

В качестве имени функции может выступать любой допустимый идентификатор (идентификаторы описаны в главе 2). Однако более всего подходит точное имя, описывающее назначение функции. В нашем случае никаких проблем с выбором имени не будет: функция называется `add`.

Возвращаемое значение должно иметь целочисленный тип. Воспользуемся типом `int`, поскольку функция возвращает число.

Последнее, что нам нужно, – список аргументов функции. Поскольку функция принимает два аргумента, необходимо создать два объявления переменных, которые мы произвольно назовем `a` и `b`. Они будут иметь целочисленный тип, поскольку представляют собой два числа, которые необходимо сложить. Так выглядит полученный список аргументов:

```
(int a, int b)
```

Воспользуемся общим синтаксисом и сложим всю информацию с целью создания объявления функции `add()`:

```
int add(int a, int b);
```

Прежде чем мы продолжим создание функции, нам придется изучить процесс определения функций.

Определение функции

Информация из этого раздела относится к глубинной сущности функций. В этом разделе мы займемся написанием кода, заключенного в функцию. Определение функции схоже с объявлением – добавляются две фигурные скобки (`{` и `}`), в которые и заключается код. Общий синтаксис:

```
тип_возвращаемого_значения имя_функции(список_аргументов)
{
    код;
}
```

Здесь *код* – это один или несколько операторов, каждый из которых заканчивается точкой с запятой. Обратите внимание, что в конце определения функции точка с запятой не требуется.

Одним из существенных различий языков C и C++ является возможность объявлять переменные по мере необходимости в языке C++. Без-

де, где в C++ может присутствовать оператор, может существовать и объявление переменной.

Несмотря на возможность объявлять переменные в большинстве мест программы, мы советуем проявлять постоянство в этом вопросе. Мы привыкли объявлять переменные в начале блоков программы. В результате мы всегда с легкостью находим, где была объявлена та или иная переменная.

Если функция возвращает значение, в ней должен присутствовать по меньшей мере один оператор `return`. *Оператор возврата* состоит из ключевого слова `return`, за которым следует выражение, типом данных соответствующее возвращаемому типу данных функции, плюс точка с запятой. Например, для функции, возвращающей значение типа `float`, допустимым будет такой оператор:

```
return 3.141592;
```

По достижении оператора возврата работа функции прекращается. Операторов возврата в функции может быть несколько. Часто применяются такие конструкции:

```
if (условие)
     return выражение1;
return выражение2;
```

Этот код возвращает только *выражение1*, если условие вычисляется в значение `true`. В противном случае возвращается значение *выражение2*.

Переменные, созданные объявлениями аргументов, можно использовать в теле функции. Они заполняются данными, которые передаются при вызове функции.

Пример – функция сложения (продолжение)

Теперь мы готовы определить функцию сложения. Из описания функции известно, что необходимо вернуть сумму двух аргументов. Определение никаких сложностей не вызывает:

```
int add(int a, int b)
{
     return a + b;
}
```

Как видите, функция относительно короткая и простая. В реальных приложениях функции могут становиться весьма объемными, хотя правила хорошего тона предписывают сокращать их размеры до минимума.

Вызов функции

Чтобы воспользоваться функцией в определенном фрагменте кода, следует вызвать ее из этого фрагмента. Вызов функции является пред-

писанием компьютеру начать выполнение другого фрагмента кода (того, который содержится в теле функции) и по завершении продолжить работу в основном фрагменте с того же места.

Вызов функции состоит из ее имени и передаваемых аргументов. Общий синтаксис:

```
имя_функции(значение1, значение2, ...);
```

Типы данных значений должны соответствовать типам данных аргументов в определении функции. К примеру, если первый аргумент в определении функции является целочисленным, *значение1* также должно быть целым числом. Исключением являются случаи, когда передаваемое значение может быть с легкостью приведено к нужному типу. В последнем примере мы могли бы передать значение типа `float` или `double` (скажем, `3.56`), и тогда произошло бы усечение до целого числа (в нашем случае до значения `3`).

Если список аргументов в объявлении и определении функции пуст, можно воспользоваться пустыми скобками (хотя, как мы расскажем далее, предпочтительно использовать ключевое слово `void`):

```
имя_функции();
```

Значения, передаваемые функции, присваиваются переменным списка аргументов по версии объявления функции. К примеру, если объявление функции выглядит так:

```
int my_Function(int first, int second)
{
     return first;
}
```

а вызов функции

```
my_Function(5,3)
```

семантика этого вызова равноценна семантике следующих строк инициализации:

```
int first = 5;
int second = 3;
```

Сначала компилятор сравнивает тип передаваемых данных с требуемым типом. В нашем примере C++ проверяет, что функции `my_Function` передается пара целых чисел. Если требуется преобразование (приведение), скажем, типа `float` в тип `int`, выполняется преобразование. Если типы данных являются несовместимыми, возникает ошибка компиляции. Таким образом язык гарантирует, что функции смогут работать с полученными данными.

С возвращаемыми значениями дело обстоит не так просто. Когда функция вызывается в коде, этот вызов заменяется возвращаемым значени-

ем. Поскольку my_Function возвращает целое число, эту функцию можно вызывать в любой точке кода, где допустимо целое число. Пример:

```
my_Int = 5 + my_Function(5,3);
```

После завершения работы функции my_Function вызов заменяется значением, которое она возвращает (эта подстановка неочевидна, так что имейте ее в виду, создавая программы):

```
my_Int = 5 + 5;
```

Пример – функция сложения (продолжение)

Теперь у вас достаточно информации, чтобы создать вызов функции сложения. Следующая программа запрашивает у пользователя два числа и с помощью функции add выводит их сумму:

```
#include <iostream>
using namespace std;
int main (void)
{
     int number1, number2;
     cout << "Введите первое слагаемое: ";
     cin >> number1;
     cout << "\nТеперь второе: ";
     cin >> number2;
     cout << "\nСумма: " << add(number1, number2) << endl;
}
```

А теперь все вместе

Полная программа с функцией имеет три основных составляющих:

- Необязательное объявление функции
- Функция main
- Определение функции

До вызова функцию необходимо объявить либо определить. Если определение функции расположено в начале программы, объявление становится необязательным. Однако если определение функции расположено в конце программы, а вызов функции расположен выше по тексту программы, необходимо сообщить компьютеру, что функция существует. Для этого достаточно поместить объявление функции в начало программы.

Ниже приведен полный код «Программы сложения» с функцией add. В этой программе объявление функции необходимо, поскольку ее определение расположено в конце программы:

```
//4.1 - Программа сложения - Марк Ли - Premier Press
#include <iostream>
```

```
using namespace std;
//объявление add
int add(int a, int b);
//отражает типичное применение функции
int main (void)
{
     int number1, number2;
     cout << "Введите первое слагаемое: ";
     cin >> number1;
     cout << "\nТеперь второе: ";
     cin >> number2;
     cout << "\nСумма: " << add(number1, number2)
          << endl;
}

// Суммирует два числа и возвращает результат
int add(int a, int b)
{
     return a + b;
}
```

До сих пор мы размещали весь код в разделе `int main (void)`, внутри фигурных скобок. (Этот раздел кода называется функцией `main`.) Объявления и определения функций должны находиться вне этого раздела. (Не волнуйтесь, если что-то непонятно, со временем все прояснится.)

Второй способ написать ту же программу – поместить определение функции в начале кода, а не в конце. В этом случае объявление не требуется, поскольку его роль берет на себя определение:

```
#include <iostream>
using namespace std;

int add(int a, int b)
{
     return a + b;
}

int main (void)
{
     int number1, number2;
     cout << "Введите первое слагаемое: ";
     cin >> number1;
     cout << "\nТеперь второе: ";
     cin >> number2;
     cout << "\nСумма: " << add(number1, number2) << endl;
}
```

Теперь определение функции в начале программы работает так же хорошо, как объявление в сочетании с определением в конце. Эти варианты равноценны, использование того или иного является вопросом личных предпочтений.

Ключевое слово void

Не все функции возвращают значения. Единственной целью создания функции может быть вывод текста. И в таком случае нет необходимости возвращать значение.

В отличие от случая аргументов, если возвращаемое значение отсутствует, нельзя просто оставить пустое место перед именем функции в объявлении или определении.

Здесь в игру вступает ключевое слово `void`. Слово `void` схоже с нулем в том плане, что не представляет значения. Для функции, не возвращающей ничего, используйте слово `void` вместо типа возвращаемого значения:

```
void имя_функции(список_аргументов)
```

При этом, однако, следует убедиться, что функция не попытается вернуть значение. Можно использовать пустой оператор возврата:

```
return;
```

Этот оператор завершает работу функции. Пустой оператор возврата в `void`-функции является необязательным. Если отсутствует оператор `return`, функция завершает работу, когда кончается ее код.

Если у функции нет аргументов (вне зависимости от наличия или отсутствия возвращаемого значения), можно воспользоваться ключевым словом `void` вместо пустого списка аргументов. Оба варианта равноценны в плане эффективности.

Перегрузка функций

Допустим, необходимо создать функции сложения для нескольких типов данных. Как подойти к вопросу? Один из вариантов – создать отдельную функцию для каждого типа данных, функцию с уникальным именем. Функция сложения двух целых чисел будет называться `int_add()`, а функция сложения двух действительных – `float_add()`. Этот процесс может быть довольно занудным, а когда понадобится вызвать функцию, придется еще искать верное имя для конкретного типа данных.

К счастью, в C++ существует альтернативный подход. Все функции могут иметь не уникальные имена, а одно и то же имя. Это и есть *перегрузка* функций. К примеру, все варианты функции сложения называются `add`. Но как их различает компьютер? Когда мы вызываем функцию `add`, как он определяет, который из вариантов следует использовать?

Ответ очень прост. Компьютер сравнивает типы и число аргументов, пытаясь подобрать соответствие вызова и одной из существующих функций. Пример:

```
int add (int a, int b)
{
     return a + b;
}
float add (float a, float b)
{
     return a + b;
}
int main(void)
{
     cout << add(5,3);
     cout << add(5.5, 4.7);
     return 0;
}
```

Первый вызов функции `add` приводит к выполнению целочисленной версии функции (то есть первой), поскольку числа `5` и `3` оба являются целыми; аргументы соответствуют прототипу целочисленной версии. Второй вызов `add` приводит к выполнению версии функции для чисел с плавающей точкой.

Значения аргументов по умолчанию

В C++ можно указать значение по умолчанию для некоторых параметров. Если вызывающая процедура передает неполный список аргументов, используются стандартные значения.

Общий синтаксис для реализации такого подхода:

тип_возвращаемого_значения имя_функции(*тип_арг имя_арг = значение_по_умолчанию*)

Здесь *значение_по_умолчанию* – и есть значение, присваиваемое аргументу, если он опущен при вызове. Разумеется, аргументов по-прежнему может быть несколько:

тип_возвращаемого_значения имя_функции(*арг1 = значение1, арг2 = значение2*)

арг1 и *арг2* представляют типы и имена аргументов.

Возможно присвоить значения по умолчанию лишь некоторым из аргументов. В этом случае аргументы, не имеющие таких значений, должны идти первыми в списке аргументов. Иначе говоря, если определенный аргумент имеет значение по умолчанию, все последующие аргументы в списке также должны иметь значения по умолчанию.

При вызове таких функций некоторые аргументы можно считать несуществующими. Если функция объявлена так:

```
int my_Function(int a, int b, int c = 0);
```

ее можно вызвать двумя способами:

```
cout << my_Function(5, 3, 6);
cout << my_Function(5, 3);
```

В первом случае значением `c` будет `6`, во втором случае значением `c` будет `0`.

Такие конструкции полезны, когда присутствует редко востребованный аргумент, который все-таки должен существовать для сохранения функциональности. Сохранение функциональности и удобства применения – важный шаг на пути к отличному коду.

Область видимости переменных – смотрите дальше

Весь код программы принадлежит четырем категориям, для каждой из которых существуют собственные правила доступа:

- Код внутри функции `main`
- Код другой функции
- Код внутри блока программы, как в случае оператора `if` пространства имен
- Код вне функций

Каждая из категорий подчиняется правилам относительно того, сколько живет переменная (*время жизни переменной*) и откуда она доступна (*область видимости переменной*).

Общие правила для времени жизни и области видимости переменных:

- Если переменная объявлена вне блока программы, она называется *глобальной*, ее область действия и время жизни простираются от точки объявления до конца исходного файла. (Как мы упоминали выше по тексту главы, блок программы – это произвольный код, заключенный в фигурные скобки.)
- Если переменная объявлена внутри блока программы, она называется *локальной*, ее область видимости и время жизни простираются до конца блока.
- Если переменная объявлена в списке аргументов функции, она называется *параметром*, область видимости и время жизни простираются до конца функции.

Это означает, что функции не могут получать доступ к переменным, объявленным внутри функции `main`, если только эти переменные не передаются функциям в качестве аргументов.

Если переменная объявлена внутри блока, заключенного в другой блок, ее область видимости ограничена внутренним блоком.

Поясним приведенные правила на примере:

```
//4.2 - Пример, область видимости переменных - Марк Ли - Premier Press
#include <iostream>
using namespace std;

int subtract (int a, int b);

int global = 5;

int main(void)
{
     int a, b;
     a = 5;
     b = 3;
     cout << "Значение переменной a функции main: " << a << endl
          << "Значение переменной b функции main: " << b << endl
          << "Значение переменной global: " << global << endl;
     global = 2 + subtract(a,b);
     cout << "Новое значение переменной a функции main: " << a << endl
          << "Новое значение переменной global: " << global << endl;
     return 0;
}

int subtract(int a, int b)
{
     cout << "Значение a функции subtract: " << a << endl
          << "Значение b функции subtract: " << b << endl;
     a = a - b + global;
     return a;
}
```

Вывод:

```
Значение переменной a функции main: 5
Значение переменной b функции main: 3
Значение переменной global: 5
Значение переменной a функции subtract: 5
Новое значение переменной a функции main: 5
Новое значение переменной global: 9
```

В строке 5 объявлена глобальная переменная `global`. В строке 11 объявлены локальные переменные функции `main`, `a` и `b`. В строке 17 значения переменных `main`, `a` и `b` копируются в параметры `a` и `b` функции `subtract`. Функция `subtract` вычисляет выражение и присваивает результат переменной `a` в строке 27. Это значение возвращается функцией `subtract` и подставляется в выражение `global = 2 + subtract(a,b)`. Окончательно:

```
global = 2 + 7;
```

Другими словами, раз `subtract` возвращает значение собственной переменной `a`, в строке 17 к этому значению (в данном случае оно равно `7`) прибавляется число `2`.

Обратите внимание, что в этом коде существуют две не связанных, совершенно самостоятельных версии переменных a и b: переменная a функции main не имеет ничего общего с переменной a функции subtract.

Оператор разрешения контекста

Время от времени возникают ситуации, когда совпадают имена локальной и глобальной переменных. Это допустимо в C++, но может приводить к путанице. Пример:

```
int intVar;
int main(void)
{
     int intVar;     // не идентична глобальной переменной intVar
     intVar = 5;
     return 0;
}
```

На первый взгляд сложно определить, какой из переменных intVar присваивается значение 5, но существует специальное правило. Если локальная переменная имеет то же имя, что и глобальная, говорят, что локальная переменная *затеняет* глобальную. Все операции для этого имени будут выполняться над локальной переменной. Но как в таком случае получить доступ к глобальной переменной?

Ответ – с помощью *оператора разрешения контекста*. Чтобы работать с глобальной версией переменной, просто предварите ее парой двоеточий (::). Предшествующий фрагмент кода можно изменить следующим образом для работы с глобальной переменной:

```
int intVar;
int main(void)
{
     int intVar;
     ::intVar = 5;
     return 0;
}
```

Эффект затенения действует также и для вложенных блоков. Скажем, в случае оператора if в функции main переменная, объявленная в блоке if, может затенять глобальную переменную или локальную переменную функции main. Однако оператор разрешения контекста может использоваться только для доступа к глобальной переменной. Если переменная оператора if затеняет переменную main, можно получить доступ к затененной переменной. Пример:

```
#include <iostream>
#include <string>
using namespace std;
```

```
string str = "Люди и эльфы могут сосуществовать.";

//пример работы оператора разрешения контекста (::)
int main(void)
{
     string str = "Эльфы часто селятся в лесах.";
     cout << str.c_str() << "---" << ::str.c_str() << endl;
}
```

Вывод:

```
Эльфы часто селятся в лесах.---Люди и эльфы могут сосуществовать.
```

Статические переменные

Иногда бывает удобно использовать одну и ту же переменную в функции при каждом ее вызове. Локальные переменные не подходят, поскольку они создаются заново при каждом вызове функции. Разумеется, можно использовать глобальные переменные, но это не всегда удобно. Существует еще одно решение – *статические переменные*.

Область видимости статических переменных – такая же, как у обычных переменных того же уровня, но время жизни простирается до конца программы. Статические переменные позволяют создавать функции, у которых есть «память».

Статические переменные объявляются при помощи ключевого слова `static`:

```
static тип_переменной имя_переменной;
```

Создание статической переменной происходит лишь один раз за все время выполнения программы. Это означает, что инициализация статической переменной в строке объявления

```
static тип_переменной имя_переменной = значение;
```

также будет выполнена лишь один раз.

Взгляните на следующий пример:

```
//4.3 - Пример, статические переменные - Дирк Хенкеманс
//и Марк Ли - Premier Press
#include <iostream>
using namespace std;

int incrementFunction1(void);
int incrementFunction2(void);

//вызывает функции приращения
int main(void)
{
     for(int c = 0 ; c < 4 ; c++)
     {
          cout<<"Увеличиваю обе переменные" <<endl;
```

```
            cout<< "Значение function1: У
                 << incrementFunction1() <<endl;
            cout<< "Значение function2: У <<
                 incrementFunction2() <<endl;

      }
      return 0;
}

//функция приращения со статической переменной
int incrementFunction1(void)
{
      static int x = 0;
      x++;
      return x;
}

//функция приращения с обычной переменной
int incrementFunction2(void)
{
      int y = 0;
      y++;
      return y;
}
```

Вывод:

```
Увеличиваю обе переменные
Значение function1: 1
Значение function2: 1
Увеличиваю обе переменные
Значение function1: 2
Значение function2: 1
Увеличиваю обе переменные
Значение function1: 3
Значение function2: 1
Увеличиваю обе переменные
Значение function1: 4
Значение function2: 1
```

В этом примере переменные `x` и `y` являются локальными. `x` – статическая переменная, инициализируется лишь единожды, в то время как `y` получает нулевое значение при каждом вызове функции `incrementFunction2`. По этой причине `incrementFunction2` всегда возвращает `1`, тогда как `incrementFunction1` продолжает увеличивать `x`.

Добро пожаловать на гонки улиток

Настало время испытать приобретенные в этой главе знания. В игре «Гонки улиток» обращайте внимание на места, связанные со следующими моментами:

- Существует лишь одна глобальная переменная (`money`). Она является глобальной потому, что объявлена вне функций.
- `money` используется в качестве локальной переменной функции `race()` и в качестве глобальной переменной; по умолчанию происходит обращение к локальной переменной. При обращении к глобальной переменной используется оператор разрешения контекста (`::`).
- В этом примере две версии функции `race()`. Функции имеют различные аргументы и являются примером перегрузки функций.

А теперь – что касается общего тона. Вам никогда не хотелось, чтобы время хотя бы чуть-чуть замедлило ход? Отправляйтесь на гонки улиток, и это желание сбудется. (Не беспокойтесь, в конце концов, одно из этих очаровательных созданий придет к финишу, пусть и придется провести некоторое время в ожидании.)

```
//4.4 - Игра "Гонки улиток" - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
#include <ctime>
using namespace std;

//объявления функций
int main(void);
int race(int, int);
void race(void);
int menu(void);
int placeBet(int);
void ini(void);

//переменные
int money = 200;

//Функция main
int main(void)
{
     ini();

     int userResponse;
     cout<< "Добро пожаловать на гонки улиток!!!" <<endl;
     while(userResponse = menu())
     {
          switch(userResponse)
          {
          case 1:
          case 2:
          case 3:
               ::money +=
               race(placeBet(userResponse), userResponse);
               break;
          case 4: //пользователь не сделал ставку
               race();
               break;
          }

     }
```

```
     return 0;
}

//отображает главное меню
//возвращает выбор пользователя
int menu(void)
{
     int userResponse;
     cout << "У вас " << money << " долларов."<< endl;
     do
     {
          cout<< "Меню гонок" <<endl
               << "1) Поставить на улитку 1" << endl
               << "2) Поставить на улитку 2" << endl
               << "3) Поставить на улитку 3" << endl
               << "4) Просто наблюдать " << endl
               << "0) Уйти домой" << endl;
          cin>> userResponse;
     }
     while(userResponse < 0 && userResponse > 4);
     return userResponse;
}

//выбор ставки
int placeBet(int userResponse)
{
     int betAmount;
     cout<< "Улитка " << userResponse << " - отличный выбор!"
          << endl;
     cout<< "Сколько вы желаете поставить на улитку "
          << userResponse <<"?";
     cin >> betAmount;
     return betAmount;
}

//просто смотрит гонки
void race (void)
{
     race(0, 0);
}

//сделана ставка
int race (int money, int userResponse)
{
     //хранит случайное число
     int winner = rand() % 3 + 1;
     cout<< "Улитки рванулись вперед" << endl
          << "Посмотрите, какая СКОРОСТЬ!!!" << endl
          << "Победила улитка " << winner;
     if(winner == userResponse)
     {
          cout<< "Вы выиграли!" <<endl;
          return 2 * money;
     }
```

```
        cout<<"Вы проиграли " << money << " долларов." <<endl;
        return -1 * money;
}

//инициализирует программу
void ini(void)
{
        srand(time(0));
}
```

Программа гонок довольно проста – вы можете развивать и совершенствовать ее, делая более интересной.

Что скрывает функция main

Вы, скорее всего, обратили внимание, что мы упорно называем `main` функцией. Тому есть причина, `main` – действительно функция. «Не можете быть! – скажете вы. – Функции мы изучили только что, а `main` используем с самого начала книги!» Не поддавайся страху, юный искатель приключений. Все гораздо проще, чем можно подумать.

Можно в общих чертах определить *функцию* как средство разделения кода программы на мелкие сегменты, более удобные в обращении.

Подумайте об операционной системе своего компьютера. Если она многозадачная, то выполняет сразу много программ. В каждой из этих программ есть функция `main` (или ее аналог).

В общем случае каждая программа начинается с началом функции `main` и заканчивается с ее окончанием.

Теперь подумайте об операционной системе как о программе. Из нее вызываются все остальные функции `main` (рис. 4.2).

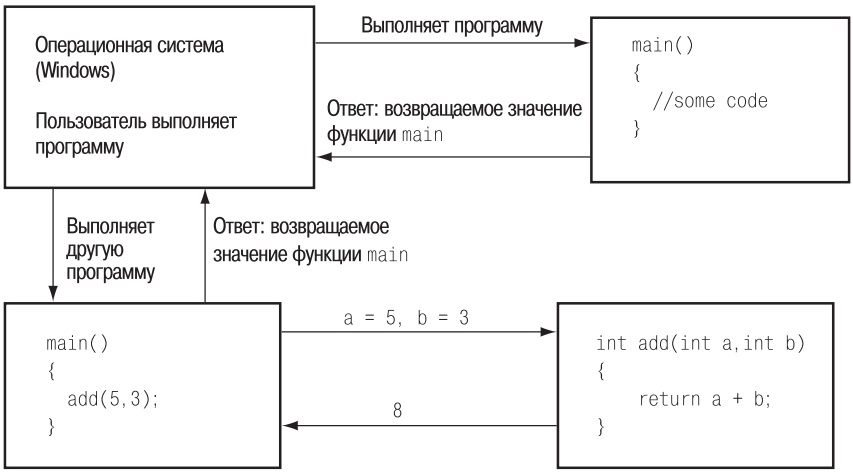


***Рис. 4.2.** Так операционная система взаимодействует с функцией main*

Вы уже заметили параллель между операционной системой и создаваемыми программами? Есть и другая параллель – между функций `main` и функциями, которые создаете вы. Поэтому `main` является функцией.

По нашему опыту большинство программистов именно в этот момент перестают понимать, как именно работает программа. Как компьютер решает, откуда начать? Мы предвидели некоторые затруднения, а потому проведем читателей через этот процесс, шаг за шагом.

Порядок выполнения

Ниже приведен порядок выполнения задач в любой программе:

1. Создаются все глобальные переменные. (Глобальные переменные инициализируются и сохраняются в коде готовой программы при компиляции.)
2. Начинает работу функция `main`.
3. Строки кода функции `main` выполняются последовательно.
4. Функция `main` завершает работу.
5. Уничтожаются все локальные переменные.
6. Когда программа выгружается из памяти, уничтожаются все глобальные переменные.

Глобальные переменные инициализируются до вызова функции `main`, поскольку находятся за ее пределами. Их область видимости больше, чем область видимости функции `main`.

Из функции `main` вызываются все остальные функции; она является вместилищем всей программы (не считая глобальных переменных). Так что, несмотря на независимость остальных функций от `main`, они до некоторой степени подчинены ей.

Передача аргументов main

Когда операционная система запускает программу, она иногда передает этой программе информацию. Тот тип программ, созданием которого мы сейчас занимаемся, может получать только один вид информации – аргументы командной строки. Более совершенные программы, в частности разработанные для Windows, могут получать различные виды информации.

В системах DOS и UNIX, чтобы выполнить программу, следует набрать ее имя в командной строке. Для некоторых программ дополнительная информация может указываться непосредственно после имени. Например, в случае броузера можно набрать веб-адрес после имени программы, и сразу после запуска броузер откроет страницу по указанному адресу.

Если вы используете операционную систему Windows, то, вероятно, не знакомы с набором команд в командной строке (в отличие от тех, кто имел дело с DOS или UNIX). Чтобы получить доступ к приглашению MS-DOS, нажмите кнопку Пуск на панели задач Windows и из открывшегося меню выберите Программы → Сеанс MS-DOS, либо Пуск → Программы → Стандартные → Командная строка. В поисках дополнительной информации обращайтесь к руководствам по работе с командной строкой, доступным в Интернете.

В главе 6 «Сложные типы данных» вы узнаете, как использовать полученные аргументы командной строки в своих программах. Пока что достаточно знать, что это в принципе возможно.

Возвращаемые значения main

До сих пор во всех наших программах функция `main` возвращала значение `0`. Практически во всех программах используется это значение, но возможно присутствие и других, имеющих различный смысл.

Вызовы программ операционной системой часто имеют следующую структуру:

```
if (вызов_программы)
    сообщение_об_ошибке;
```

Здесь *сообщение_об_ошибке* – это фрагмент кода, отображающий некоторое сообщение об ошибке.

Поскольку до сих пор все программы возвращали значение `0`, код сообщения об ошибке не выполнялся. Возврат значения `0` означает, что программа отработала без ошибок. Если вернуть в систему ненулевое значение, выполнится строка *сообщение_об_ошибке*, что будет означать возникновение ошибки в программе.

Возвращаемое значение функции `main` сообщает компилятору о результатах выполнения программы. Эта информация нам еще пригодится в последующих главах, так что постарайтесь ее запомнить.

Макроопределения: константы на стероидах

Возможно, читателей это шокирует, но мы уже познакомили их с одним видом *макроопределений*: с константами, объявляемыми посредством ключевого слова `define`. Все макроопределения создаются с применением этого ключевого слова. Макроопределения – это способ замены краткого фрагмента кода чем-то более подходящим. Единственное назначение макроопределений – облегчать жизнь. Они никак не влияют на эффективность кода. Более того, исполняемый код не меняется вообще. Пример:

```
#define PI 3.141592
```

Этот код предписывают компилятору заменить слово `PI` в коде числом `3.141592`. В главе 2 мы называли этот оператор определением констан-

ты. В качестве более общего названия подойдет макроопределение. Константа, определенная ключевым словом `define`, является разновидностью макроопределения.

Однако применение макроопределений не ограничивается константами. Они могут использоваться для замены распространенных фрагментов кода еще более короткими вариантами. Макроопределение может выполнять подстановку для любого оператора, как показано ниже:

```
#define PRNT cout <<
...
PRNT("Здравствуй, мир");
```

В данном примере каждое вхождение строки `PRNT` в тексте программы будет заменяться кодом `cout <<`, что, по существу, сделает текст программы более читаемым. Обратите внимание на скобки, в которые заключен аргумент `"Здравствуй, мир"` – как при вызове функции. Без макроопределения приведенный код выглядел бы так:

```
cout << "Здравствуй, мир";
```

Вспомним, что фрагмент информации, передаваемый функции, называется *аргументом*. Макроопределения в этом отношении схожи с функциями. В приведенном примере строка `"Здравствуй, мир"` может считаться аргументом, передаваемым макроопределению. В действительности это не совсем верно, поскольку компилятор просто производит подстановку кода, строка никуда не посылается. Макроопределение не возвращает значения, но такая возможность существует.

Распространенный пример макроопределения, возвращающего значение, – `MAX`. Оно возвращает большее из двух чисел. Если числа равны, возвращается одно из них. А вот и `MAX` во всей красе:

```
#define MAX(a,b) (a>b) ? a : b
```

Макроопределение имеет два аргумента, `a` и `b`. Раздел кода

```
MAX(a,b)
```

аналогичен определению функции с той разницей, что не требуется объявление типа. Остаток строк

```
(a>b) ? a : b
```

можно считать кодом функции.

Вот пример использования этого макроопределения в программе:

```
int x = 5;
float y = 5.5;
cout << MAX(x, y);
```

Вывод:

```
5.5
```

А так код выглядит без макроопределения:

```
int x = 5;
float y = 5.5;
cout << (x > y) ? x : y;
```

Обратите внимание, что `a` и `b` в макроопределении заменяются переменными `x` и `y` при подстановке. `a` и `b` используются для представления аргументов, переданных макроопределению.

Как все это связано с функциями? Грубо говоря, макроопределения являются упрощенными вариантами функций. Понятия схожи. Макроопределения могут принимать аргументы и возвращать значения, как функции. Чтобы произвести подстановку, как и при вызове функции, необходимо указать имя макроопределения и его аргументы.

Игра «Приключение в пещере»

Выбравшись из Зловещего макро-леса, ты видишь в отдалении дым. Подобравшись ближе, ты понимаешь, что горит деревня. Деревню карликов атаковали гоблины. Благородный искатель приключений, лежит ли твой путь через пещеру гоблинов, избавишь ли ты этот мир от порочных злодеев? Карликам нужна твоя помощь, а тебе понадобятся все знания о функциях, чтобы помочь им!

```
//4.5 - Игра "Приключение в пещере" - Дирк Хенкеманс - Premier Press
#include <iostream>
using namespace std;

bool intro(void);
void room(bool enemy, bool treasure, string description);
//информация об игроке
string name = "";
//информация о противнике
string enemyName = "";
//информация о сокровище
string treasureName = "";
//описания комнат;
const string room1 = "Ты проник через вход в пещеры.";
const string room2 = "Ты продвигаешься дальше, вглубь пещер.";
const string room3 = "Ты достиг предельных глубин пещер.";

int main( void )
{
     if (intro())
          return 0; //игрок отказался идти в пещеру, завершаем программу
     treasureName = "золотой меч";
     enemyName = "гоблина";
     room(true, true, room1);
     enemyName = "вомбата";
     room(true, false, room2);
```

```
     enemyName = "лорда хобгоблинов.";
     treasureName = "груду драгоценностей.";
     room(true, true, room3);
     return 0;
}

bool intro(void)
{
     cout<<"О храбрый рыцарь!!! Как твое имя? \n";
     cin>>name;
     cout<<"Нам нужна твоя помощь, "<< name
          <<", нашу деревню захватили  \n "
          <<"гоблины из северных пещер.  Примешь "
          << " ли ты вызов? \n \n";
     cout<<"1) да \n"
          <<"2) нет \n \n";
               int response;
     cin>>response;
               return !(response == 1);
}

//Отображает описание комнаты и варианты действий
void room(bool enemy, bool treasure, string description)
{
     while(true)
     {
          cout<<description.c_str()<<endl<<endl;
          int response = 0;
          do
          {
              cout<<"Что ты желаешь сделать? \n"<<endl;
if(enemy)
                  cout<<" 1)Напасть на подлого "<<
                         enemyName.c_str()<<endl;
              else if(!enemy)
                  cout<<" 1)Перейти в следующую комнату.";
              if(treasure)
                 cout<<" 2)Взять <<treasureName.c_str()
                         <<endl;
              cin>>response;
           }
         while(response < 1 || response > 2);
                  switch(response)
         {
             case 1:
                 if(enemy)
                 {
                     enemy = !enemy;
                     cout<<"Ты убил смертоносного "
                         <<enemyName.c_str()<<endl;
                 }
                 else if(!enemy)
```

```
                    return;
                break;
            case 2:
                treasure = !treasure;
                cout<<"Ты взял "
                        <<treasureName.c_str()<<endl;
                break;
        }
    }
}
```

Не торопитесь, изучите эту программу. Разберитесь в принципах ее работы. Попробуйте скомпилировать ее. Затем внесите изменения. Сделайте все по-своему. Соотнесите код программы с концепциями, изложенными в этой главе.

Резюме

В этой главе мы изложили очень важные положения, и с каждым словом вы становитесь все ближе к статусу профессионального программиста. Вы уже научились создавать и использовать функции, знаете, что такое область видимости переменной, а также почему `main` является функцией.

На данном этапе у вас есть базовый набор инструментов, необходимых для создания собственных программ. Попытайтесь писать программы самостоятельно и выясните, на что вы способны. Пользуйтесь книгой в качестве справочника, но старайтесь как можно больше делать по памяти.

Если вы запутались, просто перечитайте главу. Прежде чем двигаться дальше, следует разобраться с уже изученным материалом.

Задания

1. Создайте функцию `multiply`, которая перемножает два числа и возвращает результат.
2. Измените функцию `multiply`, чтобы она запоминала, сколько раз ее вызывали.
3. В чем разница между глобальной переменной и статической переменной? В каких ситуациях предпочтительно использовать тот или иной вариант и почему?
4. Попробуйте переписать игру «Приключение в пещере», не используя функции. Упражнение покажет, насколько они полезны.
5. Если вы честно выполнили все задания, купите себе прохладительный напиток.

5

Боевые качества ООП

C++ позволяет повторно использовать код, избавляя нас от необходимости снова и снова писать одни и те же фрагменты. И хотя таким свойством обладают многие языки программирования, повторно используемый код обычно приходится модифицировать, чтобы он работал. Поддержка объектно-ориентированного программирования (ООП) в языке C++ сводит повторное использование объекта, такого как герой игры, к простому переносу в следующую программу. Можно даже клонировать его и создать могущественную армию супергероев, и при этом практически не придется писать дополнительный код.

В этой главе вы научитесь:

- Объявлять классы
- Создавать объекты
- Создавать модули тестирования объектов
- Создавать публичные и скрытые методы и поля классов
- Применять фундаментальные принципы ООП
- Создавать многопользовательские стратегические игры

Введение в объектно-ориентированное программирование

Объект программирования можно считать обычным объектом реального мира. Каждый объект реального мира имеет определенные свойства и подвержен внешним воздействиям. Скажем, у лука есть конкретные свойства: цвет, число стрел, вес, а также конкретные возможности, например, способность стрелять. Заказав в арсенале лук, мы не можем знать заранее, какими свойствами он обладает. Но когда мы его увидим, сможем определить цвет, вес и размер колчана. Как вы узнаете в следующем разделе, похожим образом дело обстоит и с объектами мира программирования.

Прежде чем продолжить изучение ООП, необходимо рассмотреть ключевые термины.

Класс схож с общей моделью, на основе которой создаются объекты. Например, класс `dog` будет обладать характеристиками, присущими общему представлению о собаке.

Создание класса – это создание нового типа данных. В процессе создания класса мы рассказываем компьютеру о роде и объеме данных, которые может хранить новый тип, а также какие действия могут выполняться для нового типа. Затем можно использовать новый класс для создания переменных нового типа (эти переменные называются *объектами*). Несмотря на кажущуюся сложность, создание собственных классов – дело очень простое, что мы и докажем в последующих разделах.

Все поля данных и методы объявляются внутри класса. На рис. 5.1 приведен пример того, как может выглядеть класс `Bow`. Класс `Bow` описывает базовые свойства и возможности лука. Класс `Bow` объясняет компьютеру, что такое лук, в терминах программирования, так что в дальнейшем лук уже не будет вызывать у компьютера удивления.

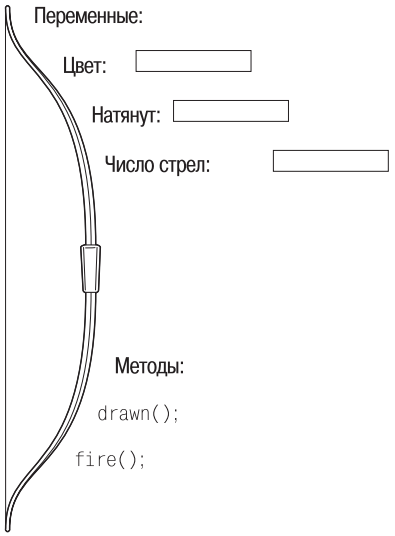


Рис. 5.1. *Класс Bow, его объектные методы и атрибуты*

Объект – это конкретный экземпляр класса. Иначе говоря, класс *объявляет* свойства, которыми обладает объект, а объект хранит конкретные значения этих свойств. Например, класс `Bow` описывает такое свойство лука, как цвет, но не его значение. Таким образом, каждый объект `Bow` имеет собственное значение цвета. На рис. 5.2 показаны два объекта `Bow`. Каждый объявлен с помощью класса `Bow`, но, если класс только описывает свойства лука, то оба объекта уже имеют конкретные значения этих свойств.

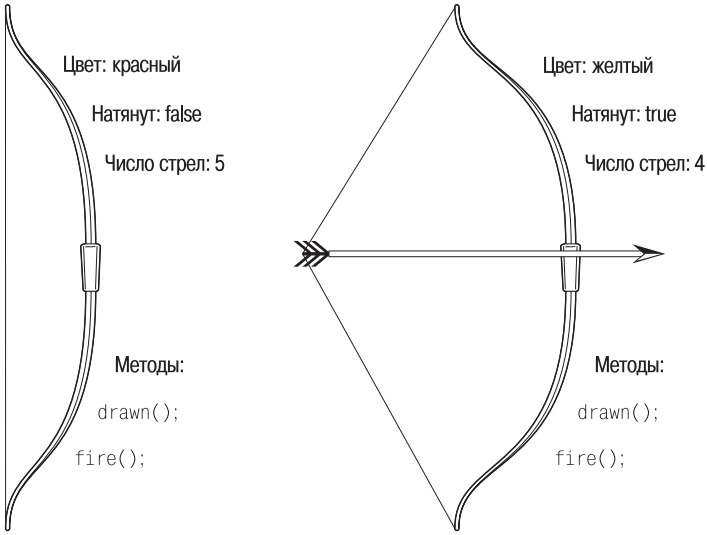


Рис. 5.2. *Два экземпляра объекта класса Bow*

В качестве другого примера возьмем класс `Knight`. Всем известно, что рыцари обаятельны, спасают принцесс, убивают драконов, а также находят магические предметы. Такая манера описания атрибутов похожа на процесс создания класса `Knight`. Создав конкретного рыцаря, скажем, сэра Ланселота, мы создадим конкретный объект. Разумеется, сэр Ланселот может спасать принцесс, убивать драконов и вместе с королем Артуром отправляться на поиски магических предметов, как и указано классом. Однако его действительные *атрибуты* – имя меча, число спасенных принцесс, а также степень обаяния – хранятся в объекте `SirLancelot`.

Методы (известные еще как *компонентные функции* и *объектные методы*) – это функции, которые принадлежат классу и определяют действия, которые можно выполнять для объекта класса. Для класса `Bow` могут понадобиться методы вроде `draw()` (натянуть тетиву) и `fire()` (выстрелить). Методы – те же функции, с которыми мы уже работали, с той разницей, что они являются частью объекта. Методы позволяют определить возможности объекта, а их основное назначение – работа с данными объекта (известными также как компоненты данных и *поля данных*).

Поля данных для класса – это переменные, которые определяют, какого рода данные может хранить объект класса. Например, объект класса `Bow` может хранить цвет, вес и размер колчана.

Компонентами класса являются все методы и поля данных, принадлежащие к этому классу. Например, компонентами объекта `Bow` будут поля данных цвета, веса и размера колчана, а также методы `fire()` и `draw()`.

Чтобы запросить выполнение определенных действий объектом, ему следует послать *сообщение*. Именно так происходит выполнение методов объекта. Например, чтобы выстрелить из лука `bow1`, следует послать сообщение `fire`, которое приведет к выполнению метода `fire()`. Это возможно потому, что в классе `Bow` существует метод `fire()`. Другими словами, `bow1` знает, как стрелять, потому что все объекты класса `Bow` знают, как это делать. Задача программиста – послать объекту `bow1` сообщение, предписывающее выстрелить, в момент, когда это необходимо. Отправкой сообщений мы займемся в разделе «Работа с объектами» далее по тексту главы.

Знакомимся с классами

Классы могут принимать самые разнообразные формы, так что настало время нырнуть в глубины ООП. Поначалу может быть страшновато, но подумайте о том, как много вы уже узнали. Едва начав постигать принципы, вы тут же поймете, что ООП гораздо проще, чем кажется. На деле – существенно проще, чем многие люди привыкли считать. В этом разделе мы облегчим изучение классов до предела и научим читателей объявлять их, создавать методы, разграничивать доступ к объектам и организовывать хранение классов в файлах.

Объявление класса

Следующий фрагмент кода представляет собой общий синтаксис объявления класса. Сейчас мы подробно рассмотрим эту конструкцию, но внимательно изучите ее, прежде чем двигаться дальше:

```
class ИмяКласса
{
    списокКомпонентов
};
```

Здесь *списокКомпонентов* – это перечень компонентов класса, *ИмяКласса*, соответственно, его имя. Обратите внимание, что *ИмяКласса* начинается с прописной буквы. Это общепринятое правило, и мы рекомендуем следовать ему, поскольку оно делает код существенно более понятным. Кроме того, следует отметить, что объявление заканчивается точкой с запятой. Отсутствие точки с запятой в данном случае может приводить к странным ошибкам, так что не забывайте о ней.

списокКомпонентов состоит из объявлений компонентов класса. Здесь могут присутствовать объявления полей данных или методов. Объявления полей данных – это уже знакомые нам объявления переменных. К примеру, расположенное в пределах объявления класса `int x;` – объявление поля данных. Однако поля данных не могут инициализироваться в точке объявления. Их следует инициализировать в одном из методов либо извне класса. Например, объявление поля данных `int x = 5;`

приведет к возникновению ошибки. Область видимости для всех полей данных совпадает с областью видимости объекта класса. При этом поля данных не всегда доступны извне класса. В классе `Bow` поля данных могут хранить информацию о том, сколько стрел осталось, какого цвета лук и натянута ли тетива. Компоненты данных класса определяют и описывают свойства класса (называемые также *атрибутами*). Таким образом, для каждого объекта лука существует свойство цвета.

Объявления методов столь же легко осмыслить, как объявления полей данных. *Объявления методов* – это объявления функций, размещенные в пределах объявления класса (вспомните, что объявления функций могут включать или не включать их реализации). Доступ ко всем методам класса можно получить только через объект класса (за исключением статических методов, речь о которых пойдет в разделе «Статические компоненты»).

В классе могут присутствовать специальные методы: конструктор и деструктор. Оба специальных метода являются необязательными, но предоставляют особую функциональность, которая не может быть реализована другим способом.

Конструктор выполняется каждый раз при создании экземпляра класса – то есть при каждом объявлении нового объекта. Конструктор обычно используется для установки начальных значений полей данных. Например, конструктор класса `Bow` может устанавливать логическую переменную `drawn` в значение `false`, а число стрел в значение `20`. Имя конструктора всегда совпадает с именем класса. Конструктор не имеет возвращаемого значения (даже `void`).

Классы и конструкторы легче понять, если сравнивать объявляемые классы со встроенными типами данных, которые C++ понимает изначально, такими как `float` *и* `int`. *Все типы данных имеют подобие конструкторов, даже встроенные типы C++. Например, для определения целочисленной переменной используется ключевое слово* `int`. *C++ известны свойства целого числа. Как следствие, C++ подготавливает к работе переменную, которая может принимать только целочисленные значения. Однако C++ не знает, как должен выглядеть объект* `Bow`, *когда он впервые определен. Поэтому класс* `Bow` *объясняет C++, какими свойствами будет обладать объект* `Bow`, *а конструктор создает объект* `Bow` *с некоторыми начальными значениями.*

Противоположностью конструктора является *деструктор*, который выполняется при уничтожении объекта. Имя деструктора всегда совпадает с именем класса, но предваряется знаком «тильда» (`~ИмяКласса()`). Деструктор не может принимать аргументы или возвращать значение. Деструктор класса `Bow` будет иметь имя `~Bow()`. Деструктор часто используется для выполнения служебных задач очистки.

Вот так к этому моменту выглядит класс `Bow`:

```
class Bow
{
```

```
    //объявления полей данных
    string color;
    bool drawn;
    int numOfArrows;

    Bow(string aColor);   //конструктор
    ~Bow();                    //деструктор

    //методы
    void draw();
    int fire();
};
```

В классе `Bow` три поля данных, связанных с атрибутами, которые будут доступны для каждого объекта `Bow`. Кроме того, класс `Bow` содержит четыре метода: `Bow()`, `~Bow()`, `draw()` и `fire()`, код для которых еще не написан. Приведенный фрагмент кода можно скомпилировать (поскольку компилятор предполагает, что реализация методов находится в другом месте), но если попытаться его использовать, ничего хорошего не выйдет – ведь объявленные методы не реализованы.

Создание методов

Мы уже рассмотрели объявления классов в общих чертах, и теперь пришло время более внимательно изучить методы. В этом разделе содержатся специальные правила и указания по синтаксису, а также рекомендации, чего следует опасаться при проектировании методов классов.

Прежде всего, существует два способа объявлять методы. Чаще всего метод объявляется в объявлении класса, а реализуется вне объявления класса. Второй вариант – метод объявляется и реализуется немедленно внутри объявления класса.

В основном рекомендуется использовать первый способ. При этом для реализации метода применяется особый синтаксис. Синтаксис для реализации метода вне класса:

```
тип_возвращаемого_значения ИмяКласса::имяМетода(списокАргументов)
{
    реализацияМетода
}
```

Здесь *ИмяКласса* – имя класса, а *реализацияМетода* – код тела метода. Как можно видеть, синтаксис весьма похож на синтаксис определения функции. Обратите внимание на еще одно применение оператора разрешения контекста (`::`). Мы сообщаем компьютеру, что следует воспользоваться областью видимости определенного класса.

И вот мы уже готовы создать реализации методов `draw()` и `fire()` класса `Bow`. Вот они:

```
//натягивает тетиву
void Bow::draw()
```

```
{
     drawn = true;
     cout<<color<< " лук: тетива натянута." <<endl;
}
//стреляет, если тетива натянута
int Bow::fire()
{
     if(!drawn)
     {
          cout<< color << лук: тетива не натянута, "
               << "невозможно произвести выстрел." << endl;
          return 0;
     }
     int score;
     score = rand() % (10 - 0 + 1) + 0;
     if(score == 0)
          cout<<color<< "лук: промах!!!" <<endl;
     else
          cout<< color << лук: выбил " << score
               << " очков!!!" <<endl;
     return score;
}
```

Помните, эти реализации должны находиться вне объявления класса. Кроме того, они должны следовать после объявления класса.

Второй вариант – одновременно объявить и реализовать метод. Синтаксис вам уже знаком. Пример:

```
class Hello
{
     void Display() { cout << "Здравствуй, мир.\n"; }
};
```

Такой вариант объявления метода очень прост. Загвоздка состоит в том, что в этом случае метод будет *подставляемым* (*inline*). А поскольку отсутствует ключевое слово, которое напоминало бы об этом, следует проявлять осторожность в отношении методов, объявленных таким образом.

Если метод реализован вне класса, его по-прежнему можно сделать подставляемым, предварив реализацию ключевым словом `inline`. Если метод (или функция) является подставляемым, при компиляции все вызовы этого метода заменяются кодом метода (функции). Подставляемые функции похожи на макроопределения. Они позволяют выиграть в скорости, но увеличивают размер файлов программы.

Проектируем конструкторы и деструкторы

Для новичка конструкторы и деструкторы часто оказываются достаточно сложной темой, но в этом разделе мы поможем читателям разобраться со сложностями.

Конструктор вызывается автоматически при создании объекта, деструктор вызывается автоматически при его уничтожении. Конструктор инициализирует компоненты данных и выполняет прочие задачи, связанные с созданием объекта. Деструктор выполняет задачи, связанные с окончательной «чисткой».

Конструкторы и деструкторы похожи на методы: их объявления могут соседствовать с реализациями либо быть разнесены в коде. Ниже приведен синтаксис для реализации в момент объявления:

```
class ИмяКласса
{
     //конструктор
     ИмяКласса(списокАргументов)
     {
          реализация
     }

     //деструктор
     ~ИмяКласса()
     {
          реализация
     }

     //прочие компоненты
};
```

А вот синтаксис для раздельных объявления и реализации:

```
class ИмяКласса
{
     ИмяКласса(списокАргументов);
     ~ИмяКласса();
     //прочие компоненты
};

ИмяКласса::ИмяКласса(списокАргументов)
{
     реализация
}

ИмяКласса::~ИмяКласса(списокАргументов)
{
     реализация
}
```

Обратите внимание, что конструктор может иметь аргументы. Если конструктор имеет аргументы, пользователь класса должен передать значения этих аргументов при создании объекта. Аргументы конструктора – вещь необходимая для многих объектов. Например, в случае класса `Date` аргументы могут определять дату, которая будет сохранена объектом `Date`.

Деструктор, напротив, не может иметь аргументов. Он вызывается автоматически, и пользователь может не иметь возможности эти аргументы предоставить.

Поскольку аргументы допустимы для конструктора, может возникнуть необходимость в его перегрузке. Это допустимо в C++ и довольно часто встречается в крупных классах. Например, можно предоставить пользователям класса `Date` возможность инициализации объекта строковым представлением даты или целочисленным. Такая перегрузка конструктора делает класс более гибким и расширяет функциональность, к которой имеют доступ пользователи.

Деструктор не может быть перегружен, поскольку не имеет ни возвращаемого значения, ни аргументов.

Удобный и быстрый способ инициализации полей данных в конструкторе связан с применением списка инициализации. *Список инициализации* – это список полей данных, которые требуется инициализировать; для каждого поля приводится значение, заключенное в скобки. Общий синтаксис:

```
ИмяКласса(списокАргументов) :
     поле1(значение1), поле2(значение2)
{
     реализация
}
```

В списке инициализации может быть произвольное число записей, но с ростом их количества затрудняется его восприятие. Синтаксис *поле1(значение1)* эквивалентен *поле1 = значение1* (то есть имеет ту же функциональность), но действует только в списках инициализации.

Методы конструктора и деструктора для класса `Bow`:

```
Bow::Bow(string aColor)
{
     numOfArrows = 10;
     drawn = false;
     color = aColor;
      //инициализация по времени
     //(функция rand() понадобится в методе fire())
     srand(time(0));
}

Bow::~Bow()
{
}
```

Константные методы – никакого риска

Вы будете часто создавать методы, которые не изменяют значения полей данных. Эти методы можно делать константными. Константный

метод не может (в принципе) изменять значения полей данных. Попытка выполнить подобное действие приведет к ошибке синтаксиса.

Чтобы объявить константный метод, следует поместить ключевое слово `const` за списком аргументов. Синтаксис выглядит следующим образом:

```
тип_возвращаемого_значения имяМетода(списокАргументов) const;
```

Если *константный метод* реализуется за пределами класса, ключевое слово `const` должно присутствовать и в объявлении, и в реализации.

Константные методы имеют еще одно назначение: они (и только они) доступны при создании *неизменяемого объекта* (с помощью ключевого слова `const`). Такое ограничение доступа гарантирует, что неизменяемый объект действительно не изменится.

Модификаторы доступа

C++ позволяет разграничивать доступ к компонентам класса. Это очень мощная возможность, поскольку позволяет защищать поля данных от случайного изменения (о защите данных вы узнаете больше в разделе «Изучаем принципы ООП»). *Модификатор доступа* – это ключевое слово, которое указывает, где доступны компоненты класса. Модификаторы доступа имеют следующий синтаксис:

```
class ИмяКласса
{
     компонентыКласса
     модификаторДоступа:
     компонентыКласса
};
```

Модификатор доступа влияет на все компоненты класса (в том числе и методы), которые следуют за ним, – вплоть до следующего модификатора доступа или до конца объявления класса.

Модификаторов доступа для классов существует два: `public` и `private` (на самом деле их три, о модификаторе `protected` вы узнаете в главе 8 «Наследование»). Опишем действие этих модификаторов:

- общие (открытые) компоненты (`public`). Доступны везде, откуда доступен объект класса, и из самого класса (вернее, из его методов);
- частные (закрытые) компоненты (`private`). Доступны только из самого класса. Объект класса не способен обращаться к частным компонентам кроме как через общие методы. По умолчанию все компоненты класса считаются частными.

Проиллюстрируем использование модификаторов примером:

```
class ClassName
{
     int x;
```

```
public:
     int y;
     int z;
private:
     int a;
};
```

В этом примере `x` и `a` являются частными компонентами, а `y` и `z` – общими. Модификаторов доступа в классе может быть произвольное количество.

Вот так выглядит класс `Bow` с модификаторами доступа:

```
class Bow
{
     //объявления компонентов данных
     string color;
     bool drawn;
     int numOfArrows;
public:
     Bow(string aColor);   //конструктор
     ~Bow();                    //деструктор

     //методы
     void draw();
     int fire();
};
```

Распределение классов по файлам

Классы имеют свойство разрастаться со временем, и хранение всего кода в одном файле становится весьма неудобным. Кроме того, для повторного использования классов в других программах приходится просто копировать их код. Этот процесс может становиться довольно занудным.

К счастью, существуют определенные правила для размещения классов в файлах. Обычно объявление хранится в одном файле, а реализация всех методов – в другом. Файл объявления класса обычно получает имя *ClassName*`.h`, где *ClassName* – название класса. Вы уже знакомы с файлами .cpp, содержащими код C++, так вот файлы с расширением .h также являются допустимыми в C++ (но их применение связано с хранением объявлений классов, а не программ). Файл реализации обычно называется *ClassName*`.cpp`.

Если вы следуете этим правилам, C++ дает возможность делать замечательные вещи. Поскольку класс хранится отдельно от использующей его программы, его необходимо включить в программу при помощи директивы `#include`. Синтаксис этой директивы следующий:

```
#include "имя_файла"
```

Вместо того чтобы включать оба файла (*ClassName*.h и *ClassName*.cpp), следует включить лишь один, *ClassName*.h. Компилятор включит файл .cpp автоматически (разумеется, если оба файла находятся в одном каталоге). Удобно, правда?

Вот так выглядит класс Bow, сохраненный в двух файлах:

```
//Bow.h
class Bow
{
     //объявления компонентов данных
     string color;
     bool drawn;
     int numOfArrows;
public:
     Bow(string aColor);   //конструктор
     ~Bow();                    //деструктор

     //методы
     void draw();
     int fire();
};

//Bow.cpp
Bow::Bow(string aColor)
{
     numOfArrows = 10;
     drawn = false;
     color = aColor;
     //инициализация по времени
     //(функция rand() понадобится в методе fire())
     srand((unsigned)time(0));
}

Bow::~Bow()
{
}

//натягивает тетиву
void Bow::draw()
{
     drawn = true;
     cout<< <<color<< " лук: тетива натянута." <<endl;
}
//стреляет, если тетива натянута
int Bow::fire()
{
     if(!drawn)
     {
          cout<< color << лук: тетива не натянута, "
               << "невозможно произвести выстрел." << endl;
          return 0;
     }
```

```
        int score;
        score = rand() % (10 - 0 + 1) + 0;
        if(score == 0)
             cout<<color<< " лук: промах!!!" <<endl;
        else
             cout<< color << " лук: выбил " << score
                  << " очков!!!" <<endl;
        return score;
    }
```

Классовый этикет

В этом разделе представлены советы, призванные облегчить программирование с применением ООП. Воспользуйтесь ими, и код станет легче для понимания и отладки.

- Начинайте имена классов с прописной буквы. Это правило действует не только в C++, но и во множестве других объектно-ориентированных языков программирования.
- Добавляйте в начало каждого метода и класса комментарий, поясняющий назначение кода.
- Ограничивайте функциональность каждого метода одним действием. Общее правило: если метод длиннее 20 строк, лимит функциональности уже превышен.
- Каждый класс должен моделировать лишь одну сущность. `Weapon` (оружие) и `Soldier` (солдат) – два разных класса.
- Прежде чем добавить класс к проекту, тщательно тестируйте его. Когда известно, что классы проекта работают корректно, отладка сводится к проверке правильности взаимодействия классов.

Следуйте этим правилам, и ваша производительность возрастет, а кроме того, будет сэкономлено время на отладке.

Работа с объектами

Прочитав изложение, посвященное созданию классов, вы наверняка горите желанием узнать о них побольше. Скоро вы займетесь созданием собственных классов и объектов, а пока что мы расскажем о создании объектов. Этот раздел посвящен объектным переменным, конструкторам по умолчанию, доступу к компонентам класса, а также созданию модулей тестирования для классов и статических компонентов.

Объектные переменные

Мы уже рассказывали, что создание класса во многом похоже на создание типа данных. Это отношение становится очевидным, как только начинается работа с объектами. Именем класса можно пользоваться

точно так же, как именами примитивных типов данных (которые встроены в C++). *Объектная переменная* – это переменная, которая хранит объект определенного класса.

Для создания объекта выполните следующие шаги:

1. Создайте класс, который станет шаблоном для объекта. Этот шаг мы завершили созданием класса `Bow`.
2. Создайте идентификатор объектной переменной и определите, какой объект она будет хранить (объект какого класса). Аналогичным образом объявляются и переменные.
3. По необходимости добавьте аргументы конструктора.

Для создания объектов существует два варианта синтаксиса:

Первый способ:

```
имяКласса идентификаторОбъекта(аргументы);
```

Второй способ:

```
имяКласса идентификаторОбъекта = имяКласса(аргументы);
```

Что ж, ожидание закончилось, пришло время создавать объекты. Для примера создадим пару различных объектов `Bow`:

```
Bow blue("Синий");
Bow red = Bow("Красный");
```

Мы только что создали два лука, с именами `red` и `blue`. Каждая из этих объектных переменных хранит объект, обладающий рядом свойств (таких как цвет и количество стрел), и каждая умеет выполнять два действия – натягивать тетиву и стрелять (с помощью методов `draw()` и `fire()`). Объект, хранимый в объектной переменной, можно изменить посредством оператора присваивания так же, как для обычных переменных. Пример:

```
Bow b1("синий");
Bow b2("красный");
b1 = b2;
```

Переменные `b1` и `b2` в итоге хранят различные копии объекта красного лука.

Легкий выход – конструкторы по умолчанию

При создании объектной переменной без аргументов компьютер выполняет конструктор по умолчанию. *Конструктор по умолчанию* бывает двух видов. Если класс не имеет конструкторов вообще, конструктором по умолчанию является пустой конструктор без аргументов; если автор класса реализовал конструктор без аргументов, этот конструктор и будет конструктором по умолчанию.

Пустой конструктор по умолчанию существует в каждом классе, пока не будет создан иной конструктор. Если все конструкторы класса требуют наличия аргументов, в классе нет конструктора по умолчанию. Если автор класса не создал конструктор по умолчанию, стандартный не делает ровным счетом ничего – только создает новую объектную переменную, которая соответствует объявлению класса объекта. Вот пример конструктора по умолчанию:

```
Cpoint3d::Cpoint3d()
{
}
```

Доступ к компонентам

Общие компоненты доступны везде, где доступен сам объект. Получить доступ можно с помощью оператора доступа к компонентам (.), который разделяет идентификатор объекта и идентификатор компонента. Чтобы присвоить значение общему полю данных, воспользуйтесь следующим синтаксисом:

```
идентификаторОбъекта.имяКомпонента = значение;
```

Чтобы получить значение общего поля данных объекта, следует поменять операнды местами:

```
переменная = идентификаторОбъекта.имяПеременной;
```

Как видите, объекты во многом похожи на обычные переменные. Еще один пример применения описанного синтаксиса:

```
class S
{
public:
     int x;
};

int main(void)
{
     S s;
     s.x = 99;
}
```

Поскольку поле данных `x` объекта `s` является общим, доступ к полю и его изменение возможны в любой точке программы (если доступен объект `s`). Пока существует объект `s`, весь код программы, имеющий доступ к `s`, может напрямую изменять поле данных `x` этого объекта.

Доступ к общим методам организован так же. Синтаксис вызова общего метода:

```
объект.метод(аргументы)
```

Пример вызова общего метода:

```
class S
{
     int x;
public:
     int getValue() { return x; }
     void setValue (int temp) { x = temp; }
     S(int temp) : x(temp) {}
     ~S() {}
};

int main (void)
{
     S s(5);
     cout << s.getValue();
}
```

Вывод:

```
5
```

Частные компоненты недоступны извне класса. Как правило, доступ к частным компонентам организуется через общие методы класса. Подобное разграничение доступа позволяет автору класса позаботиться о том, чтобы поля данных использовались корректно.

Модули тестирования

После создания класса его следует протестировать. Одним из преимуществ ООП является простота тестирования и отладки классов. Идея заключается в том, чтобы тестировать каждый класс в изоляции от других классов. Если класс работает так, как задумано, то не должен служить источником ошибок (теоретически ☺) при совместном использовании с остальными, не менее надежными частями программы. Для тестирования классов создаются модули тестирования.

Модуль тестирования – это программа, которая проверяет все способности класса. Тестированию подвергаются все методы, конструкторы и деструкторы.

Пример модуля тестирования для класса `Bow` приведен ниже (полная программа хранится на компакт-диске книги). Файл называется bowTest.cpp.

```
//5.1 - Модуль тестирования класса Bow - Дирк Хенкеманс
//Premier Press
#include <iostream>
using namespace std;

int main( void );
void bowTest(void);

//класс Bow здесь...
```

```
//главная функция
int main( void )
{
     bowTest();
     return 0;
}

//тестирование класса лука
void bowTest(void)
{
     cout<<"создан желтый лук"<<endl;
     Bow yellow("желтый");
     cout<<"пробуем выстрелить из желтого лука"<<endl;
     yellow.fire();
     cout<<"натягиваем тетиву лука"<<endl;
     yellow.draw();
     cout<<"пробуем выстрелить из желтого лука"<<endl;
     yellow.fire();
}
```

Обратите внимание на многочисленные операторы `cout` – они позволяют убедиться, что класс работает так, как задумано. Если происходит сбой в программе по какой-либо причине, можно точно определить, в какой строке произошел этот сбой – ориентиром послужит последнее сообщение `cout`. Следуйте методике создания модулей тестирования для всех своих классов. Это позволит избежать многих затруднений.

Состязание лучников

Его величественное высочество Король Нолан Бард приказал устроить состязание лучников в честь шестнадцатого дня рождения своей дочери Анастасии. Победитель состязания получит руку и сердце принцессы.

Основой игры «Состязание лучников» послужит класс соревнования. В каждом соревновании будет три участника и произвольное число туров. После проведения туров объявляется победитель. Изучите код и убедитесь, что поняли его до конца.

```
//5.2 - Состязание лучников - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <cstring>
#include "bow.h"
using namespace std;

class ArcheryCompetition
{
//компонентные переменные
private:
     //переменные
     int rounds;
     float redScore;
```

```
     Bow red;
     float blueScore;
     Bow blue;

public:
     //конструктор
     ArcheryCompetition(int lrounds);
     //деструктор
     ~ArcheryCompetition();

     //методы
     int compete(void);
};

//создает объект ArcheryCompetition
ArcheryCompetition::ArcheryCompetition(int lrounds) :
     rounds(lrounds), red(Bow("красный")),
     blue(Bow("синий")), redScore(0), blueScore(0)
{
}

//деструктор
ArcheryCompetition::~ArcheryCompetition()
{
}

//основа игры.
//Проводит состязание и определяет, кто победил
int ArcheryCompetition::compete()
{
     //провести все туры, отслеживая результаты
     for(int i = 0; i < rounds; i++)
     {
          cout<<"Тур номер "<<i+1<<"."<<endl;
          red.draw();
          blue.draw();

          redScore = (red.fire() + redScore * i)/(i+1) ;
          blueScore = (blue.fire() + redScore * i)/(i+1);

     }

     //определяем, кто победил
     if(redScore == blueScore)
          cout<<"Ничья!!! \n";
     else if(redScore < blueScore)
          cout<<"Синий лук получает руку принцессы!! \n";
     else
          cout<<"Красный лук получает руку принцессы!! \n";
     return 1;
}

void main(void)
{
     //управляющая функция
```

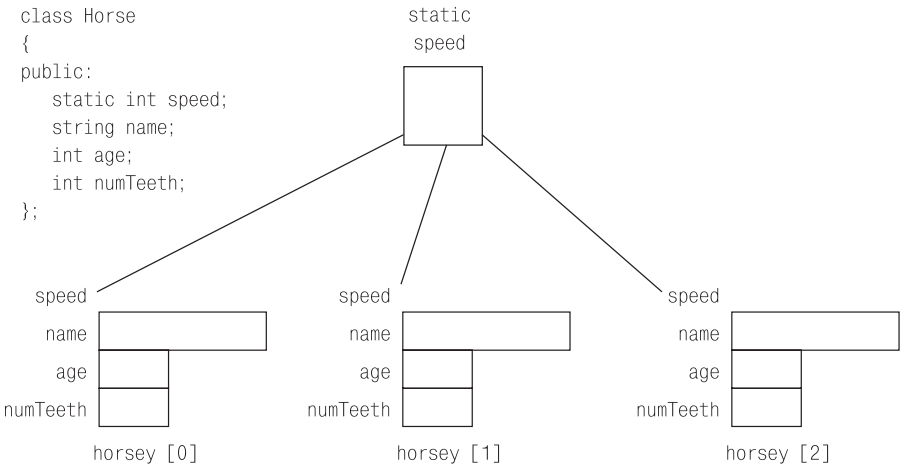
```
        //создает объект и вызывает
        //соответствующие методы
        ArcheryCompetition plymouthSquare(2);
        plymouthSquare.compete();
        int get = 0;
        cin>>get;
    }
```

Статические компоненты

До сих пор переменные каждого объекта были уникальными. Но что если необходимо создать переменную, значение которой будет общим для всех объектов конкретного класса? Можно воспользоваться глобальной переменной, но это нарушит принцип абстракции данных. Чтобы объявить переменную, существующую для всех объектов класса и допускающую стандартное для ООП разграничение доступа, воспользуйтесь статическими компонентами. *Статический компонент* – это переменная, которая является общей для всех экземпляров класса. Синтаксис объявления статического компонента:

```
class ИмяКласса
{
static типПеременной идентификаторПеременной;  //объявление
     //остальной код...
};
//глобальная инициализация
типПеременной ИмяКласса::идентификаторПеременной = x;
```

Концептуальная схема работы статических переменных приведена на рис. 5.3.



Три объекта, `horsey[0]`, `horsey[1]` и `horsey[2]`, используют общую статическую переменную `speed`.

Рис. 5.3. Статические переменные одинаковы во всех объектах класса

Обратите внимание, чтобы воспользоваться статическим компонентом, необходимо объявить переменную в классе, а затем инициализировать ее вне класса. Переменная *должна* быть инициализирована вне класса.

Предположим, у нас есть класс `Horse`, и все лошадки впряжены в одну повозку. Они должны двигаться с одинаковой скоростью, но скорость может снижаться или увеличиваться в зависимости от уклона пути. Код класса `Horse` может выглядеть так:

```
//5.3 - Программа демонстрации применения статических переменных
// Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
using namespace std;
//
class Horse
{
public:
     static int speed;
     //прочие компоненты класса horse
};

int Horse::speed = 3;//не забывайте - глобальная инициализация

int main(void)
{
     Horse horse1;
     Horse horse2;
     Horse::speed = 5;
     cout<<horse2.speed<<endl<<horse1.speed<<endl;
     Horse::speed = 6;
     cout<<horse2.speed<<endl<<horse1.speed<<endl;
     return 0;
}
```

Вывод:

```
5
5
6
6
```

Заметим, что скорости лошадей `horse1` и `horse2` одинаковы независимо от того, к какому из объектов мы обращаемся. Это происходит потому, что существует лишь один экземпляр компонента `speed` на все лошадиные объекты, впряженные в повозку.

Изучаем принципы ООП

Три следующих аспекта ООП делают программы легкими в сопровождении и позволяют повторно использовать объекты:

- Абстракция данных

- Инкапсуляция
- Полиморфизм

Первые два термина вы уже встречали, поскольку абстракция данных и инкапсуляция связаны с созданием классов с частными компонентами данных. В следующих разделах мы расскажем, что скрывается за этими простыми понятиями со звучными именами.

Абстракция данных

Купив печенье «Subway», вы можете знать, что это печенье из белого шоколада с австралийским орехом, но, скорее всего, не сможете назвать сорт муки или сколько было израсходовано соды для приготовления. И, скорее всего, эта информация вас не интересует. Печенье вкусное, то есть полностью выполняет свое назначение.

Точно так же не всегда важно знать, как работает класс, если он выполняет свои задачи. Это принцип *абстракции* (или *сокрытия*) *данных*. *Абстракция данных* – это процесс сокрытия компонентов данных и кода, реализующего функциональность, за интерфейсом, не позволяющим пользователю искажать данные. Смысл в том, что данные скрываются в реализации класса. Доступ к компонентам данных можно получить только через компоненты методов. Таким образом, работа с данными происходит через общие (public) функции, а не напрямую.

Основное правило – используйте минимально допустимую область видимости для всех переменных и компонентов. Кроме того, следует минимизировать использование глобальных переменных. В этом случае данные будут изменяться только теми фрагментами кода, которым это разрешено. Если необходимо модифицировать скрытые данные объекта, это можно сделать только с помощью методов объекта, предназначенных для этих целей. Разработчик класса определяет, какого уровня доступ к данным должен получить пользователь готового класса.

Инкапсуляция

Вспомним, как выглядело производство в девятнадцатом веке. Все действия по созданию конечного продукта выполнялись одним человеком. Разделения труда не существовало. Производитель свитеров делал все, начиная с создания шерсти (вообще говоря, эта часть работы выполнялась овцой, но смысл понятен) и заканчивая продвижением готового продукта на рынке; в результате, понятное дело, цикл разработки получался довольно длительным. С распространением специализации люди все больше становились экспертами в узких вопросах, предоставляя решение прочих задач другим людям.

Аналогично построена *инкапсуляция*. Каждый созданный класс должен выполнять один вид работ или представлять одну сущность. Затем классы используются совместно для представления более слож-

ных сущностей и понятий. Вспомните состязание лучников. В программе было несколько луков, созданных из класса `Bow`, и отдельный класс, который представлял собственно состязание.

Приведем еще один пример инкапсуляции. Предположим, мы хотим создать автомобиль. Чтобы получить готовый продукт, необходимо начать с создания более мелких его частей. Каждая из частей имеет определенное назначение и функциональность, а вместе они составляют автомобиль. Различные части автомобиля (рис. 5.4) имеют различное назначение.

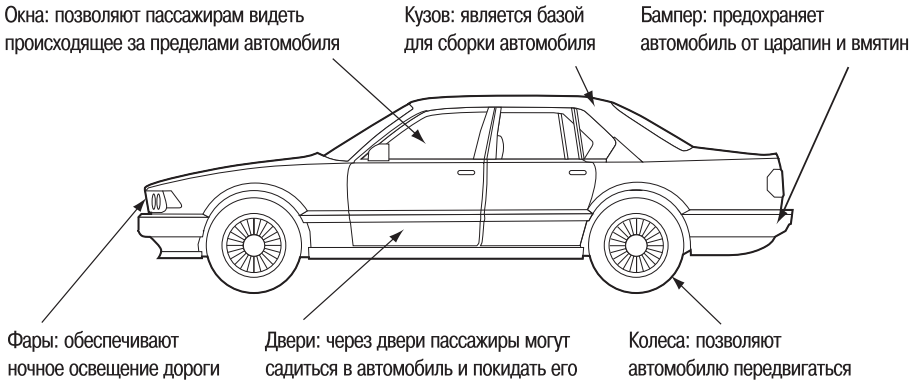


Рис. 5.4. *Можно разделить объект на множество мелких объектов, каждый из которых служит уникальной цели*

Полиморфизм

Слово *полиморфизм* звучит зловеще, а означает просто «много форм». Полиморфизм – это принцип ООП, согласно которому каждый объект может использоваться более чем в одной программе. Основное положение полиморфизма – код анализирует условия, в которых используется, и приспосабливается к этим условиям. Класс `Bow` всегда будет делать то, что ожидается, даже если его использовать на 64-битном компьютере с процессором Alpha или суперкомпьютере Cray. В главе 8 мы очень подробно изучим полиморфизм. А пока что продолжим изучать основы C++.

Использование трех описанных концепций ООП (абстракции данных, инкапсуляции и полиморфизма) позволит значительно повысить уровень вашей жизни как программиста, а также уровень общественной жизни, поскольку вы сэкономите уйму времени на отладке.

Отладка

Ваш код не всегда будет идеальным. Время от времени что-то идет не так, и бывает сложно понять, в чем проблема. Иногда причину можно

обнаружить на месте. В этом разделе мы обсудим виды ошибок в программировании и способы их избежать.

Существует четыре вида ошибок, для каждого из которых существует конкретная причина и конкретный способ разрешения.

- **Незначительные ошибки.** Ошибки, которые не влияют на работу программы. Эти ошибки, как правило, не вызывают проблем. Незначительные ошибки могут возникать, если вы не следуете соглашениям или забываете комментировать разделы кода. Эти ошибки часто остаются незамеченными, пока не становятся причиной явных проблем.
- **Ошибки времени компиляции.** Ошибки в синтаксисе языка программирования. Скажем, вы забыли букву `t` в ключевом слове `int`, пытаясь объявить целочисленную переменную; это является нарушением синтаксиса языка программирования и приведет к получению соответствующего сообщения при попытке компиляции.
- **Ошибки времени выполнения.** Ошибки, которые проявляются только при запуске программы и служат причиной ее сбоев. Эти ошибки трудно отлаживать; примером может послужить объявленный, но не реализованный конструктор. Подобные ошибки сложно диагностировать, но они приводят к аварийным отказам программ (и появлению неприятных маленьких сообщений от системы Windows). Компилятор не способен обнаружить эти ошибки; однако он может отобразить *предупреждение* (которое не прерывает процесс компиляции). Советуем прислушиваться к предупреждениям компилятора и стараться заранее исключать ошибки из кода.
- **Семантические ошибки.** Ошибки, связанные с тем, что программа не ведет себя так, как задумано. Компилятор не способен обнаружить эти ошибки. Программа безотказно работает, но делает совсем не то, что нужно. Представьте, что орки запрограммированы на то, чтобы атаковать самих себя. Как правило, эти ошибки легко исправить, но не всегда.

Отладка, несомненно, является наиболее занудной составляющей программирования и обычно занимает очень много времени.

Черный ящик

Черный ящик – это метод ООП, основанный на трех принципах ООП, описанных выше: абстракции данных, инкапсуляции и полиморфизме. Идея метода заключается в том, что объекту посылается сообщение, а объект выполняет корректные действия, даже если вы не знаете, каким образом. Это происходит потому, что объект контролирует собственные данные, что позволяет ему защитить себя от недопустимых параметров и значений. Вам не нужно знать (и, возможно, никогда не понадобится знать), что внутри класса, но к моменту завершения

Истории из жизни

Семантические ошибки могут быть причиной очень сильной головной боли. Иногда кажется, что их вообще невозможно исправить. Давным-давно я (Дирк Хенкеманс) создал свою первую игру с использованием DirectX (DirectX – это библиотека, разрабатываемая Microsoft, о ней мы расскажем в главах 13 «DirectX» и 14 «Создаем пиратское приключение»). Я написал пять страниц кода, но приложение упорно отображало только пустой экран. Я потратил неделю на отладку. Перечитал все свои конспекты по DirectX: синтаксис верный, со связыванием все в порядке. Наконец, через неделю я понял, в чем дело. Я забыл – внимание! – двойные кавычки, когда вводил имя файла со спрайтами в IDE. Создать эту проблему было невероятно легко, а обнаружить – невероятно трудно.

тестирования черного ящика объект оказывается качественно протестированным и работает так, как ожидается.

Тестирование черного ящика значительно облегчает отладку. Если отлаживать небольшие фрагменты кода и знать наверняка, что они функциональны в любой ситуации, выжить смогут только ошибки, возникающие при взаимодействии уже отлаженных фрагментов кода между собой.

Ошибки связывания

Связывание происходит в тот момент, когда компьютер берет все файлы программы, включая и стандартные библиотеки, и компилирует их в исполняемый файл (.exe). Как правило, ошибки связывания очень трудно диагностировать, поскольку не существует конкретных правил отладки. Но, пусть это и звучит зловеще, лишь две вещи могут служить причиной ошибок связывания:

- Автор программы не включил или некорректно включил библиотечные файлы (файлы стандартных библиотек и всех прочих, которыми пользуется программа).
- Автор попытался объявить команду, которая не существует, а компилятор этого не заметил. Например, если объявить конструктор для класса, но не реализовать его, при попытке скомпилировать программу возникнет ошибка связывания.

Если вы не можете понять, как справиться с определенной ошибкой, попробуйте исправить сначала другие ошибки. Иногда ошибка, вызывающая затруднение, после этого просто «исчезает».

Игра «Завоевание»

Сейчас вы используете приобретенный в ООП опыт для создания текстовой приключенческой стратегической игры «Завоевание», в которой игрок является королем и повелевает армиями в целях покорения врагов. Для каждого игрока будет создан класс `Nation`, и каждому игроку предстоит на своем ходу заниматься созданием государства и борьбой с соседями.

```
//5.4 - Завоевание - Дирк Хенкеманс - Premier Press

#include <iostream>
#include <string>

using namespace std;

//нации игроков
class Nation
{
public:
     int land;
     int troops;
private:
     string name;

     int food;
     int gold;
     int people;
     int farmers;
     int merchants;
     int blacksmiths;

public:
     Nation(string lName);
     Nation();

     bool takeTurn(void);

private:
     void menu(void);
};

Nation nation1;
Nation nation2;

//начальные значения для нации
Nation::Nation(string lName) :
     name(lName), land(20), food(50), troops(15),
     gold(100), people(100), farmers(0),
     merchants(0), blacksmiths(0)
{

}

//конструктор по умолчанию
```

```
Nation::Nation()
{

}

//выполнить ход игрока
bool Nation::takeTurn()
{
     cout << "Ход << name << ".\n";
     people += land * 0.2;
     food += farmers - people * 0.25;
     gold += merchants * 20;
     troops += blacksmiths;

     menu();

     if (nation1.land <= 0 || nation2.land <= 0) return false;
     return true;
}

//отображение и обработка меню
void Nation::menu()
{
while (true)
{
     int input = 0;
     cout << "пища " << food << endl
          << "золото " << gold << endl
          << "территория " << land << endl
          << "купцы " << merchants << endl
          << "войсковые соединения " << troops << endl
          << "безработные " << people << endl;

     cout << "1) купить земли \n"
          << "2) нанять фермеров \n"
          << У3) нанять купцов \n"
          << У4) нанять оружейников \n"
          << "5) в атаку! \n"
          << "6) сделать ход \n";
     cin >> input;

     switch (input)
     {
     case 1: //покупка земли
          cout << "Вы купили " << gold/20
               << " участков земли. \n";
          land += gold/20;
          gold %= 20;
          cout << "Теперь у вас " << gold << " золота. \n";
          break;
     case 2: //найм фермеров
          farmers += people;
          cout << "Вы наняли " << people << " фермеров. \n";
          people = 0;
          break;
```

```
        case 3: //найм купцов
             merchants += people;
             cout<< "Вы наняли " << people << " купцов. \n";
             people = 0;
             break;
        case 4: //найм кузнецов
             blacksmiths += people;
             cout << "Вы наняли " << people << " кузнецов. \n";
             people = 0;
             break;
        case 5: //сражение
             cout << "Сражение затянулось до поздней"
                  " ночи, и все погибли! \n";
             if (nation1.troops < nation2.troops)
             {
                  nation2.land += 10;
                  nation1.land -= 10;
             }
             else if (nation1.troops > nation2.troops)
             {
                  nation2.land -= 10;
                  nation1.land += 10;
             }

             nation1.troops = 0; //война - кровавое занятие!!!
             nation2.troops = 0;

             break;
        case 6:return; //конец хода
        }
   }
   }

   //главная функция игры
   int main(void)
   {
        string tempString;

        cout << "Добро пожаловать в игру Завоевание! \n";
        cout << "Игрок 1, введите свое имя: \n";
        cin >> tempString;
        nation1 = Nation(tempString);

        cout << "Игрок 2, введите свое имя: \n";
        cin >> tempString;
        nation2 = Nation(tempString);

        while(nation1.takeTurn() && nation2.takeTurn())
        {

        }

        return 0;
   }
```

Резюме

В этой главе вы много узнали об объектах. Понимание ООП требует особого подхода к программированию. Сокрытие данных в классах и защита этих данных требует предварительного планирования, но позже окупается управляемостью объекта и повторным использованием кода. Проверьте свое понимание концепций ООП, выполнив следующие задания, и при необходимости перечитайте разделы этой главы, прежде чем переходить к следующей.

Задания

1. Создайте класс для представления персонажа ролевой игры. Класс должен хранить имя персонажа, его классовую принадлежность и расу.
2. Изложите три базовых принципа ООП.
3. В чем различие между классом и объектом?
4. Имея возможность выбрать между общим (public), частным (private), глобальным и локальным объявлением без потери функциональности, что вы выберете?
5. Какие атрибуты конструкторов и деструкторов отсутствуют у других функций?

6

Сложные типы данных

А теперь, любезные читатели, закончив изучение основ языка C++, мы вступаем на земли, которые не нанесены на карты. Вы уже готовы постичь некоторые более сложные аспекты языка C++. Но не стоит бояться – пусть это и потребует отдачи сил, но через некоторое время станет столь же просто, как и изучение главы 1 «Путешествие начинается».

В этой главе вы изучите:

- Массивы
- Указатели
- Строки в стиле C
- Ссылки
- Основы работы с динамической памятью

Работа с массивами

Каждая из переменных, рассмотренных в предшествующих главах, способна хранить лишь один элемент информации. Чтобы сохранить второй, необходимо создать еще одну переменную. Переменные такого рода называются *скалярными* переменными. Но что делать, если необходимо хранить множество элементов однородных данных? Будет весьма неудобно создавать для каждого элемента переменную. А что если требуется работать со многими тысячами записей сотрудников? Задача очень быстро становится невыполнимой.

По счастью, у большинства проблем есть решения. В нашем случае таким решением являются массивы. *Массив* – это специальная группа переменных, которая позволяет хранить много однотипных значений.

Отдельная переменная в массиве называется *элементом*. Каждый элемент связан с определенным индексом. *Индекс* – это число, которое указывает, к какому из элементов массива обращается пользователь.

Новичков обычно смущает одно странное на первый взгляд обстоятельство, связанное с массивами: нумерация элементов начинается с нуля. Поэтому для массива из пяти элементов будут допустимы индексы 0, 1, 2, 3 и 4. Приобретя навыки работы с массивами, вы привыкнете к такой странной нумерации, и она начнет казаться естественной. На рис. 6.1 массив изображен в виде ряда коробок; имя массива – `charArray[]`, коробки содержат отдельные элементы массива (такие как `char_array[0]`).

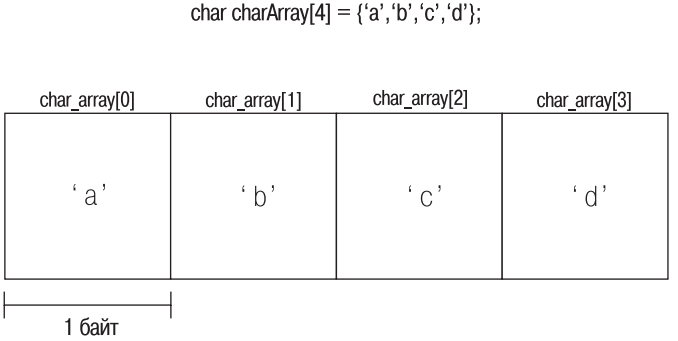


***Рис. 6.1.** Массив можно представить в виде набора ящиков, выстроенных в ряд*

Создание массивов

Создание массивов – очень простой процесс, особенно для тех, кто уже изучил переменные. Чтобы создать массив, наберите имя типа данных, а затем имя переменной с оператором индекса. Вот так может выглядеть конечный код:

```
тип_данных имя_массива[число_элементов];
```

В результате создается массив по имени *имя_массива*, элементы имеют тип *тип_данных*, а их число определяется параметром *число_элементов*. Например, чтобы создать массив из десяти целых чисел с именем `int_array`, воспользуйтесь конструкцией

```
int Int_array[10];
```

Помните, что нумерация начинается с нуля, поэтому элементы массива имеют индексы от 0 до 9. Элемент 10 на деле не существует в массиве, хотя такое впечатление создается, если взглянуть на объявление.

Элементы массива могут иметь любой тип, включая и определенный пользователем. Массив из десяти объектов класса `my_class` можно создать следующим образом:

```
my_class my_class_array[10];
```

Число в квадратных скобках должно быть *константным выражением*, то есть для определения числа элементов массива нельзя использо-

вать переменную. Как вы узнаете в последующих главах, такая неспособность к изменению размера является одним из крупных недостатков массивов, в сравнении с другими сложными типами данных.

Несколько примеров объявлений массивов:

```
int int_array[10];
float float_array[60];
char char_array[6];
```

Инициализация массивов

Заполнить массив данными можно разными способами. Первый способ: массив может быть инициализирован в момент объявления. Этой цели служит список инициализации, то есть список значений, используемых для заполнения массива. Используйте следующий синтаксис:

тип_данных имя_массива[*число_элементов*] = {*значение1*, *значение2*, *значение3*, ...};

Все значения списка инициализации (*value1*, *value2* и т. д.) имеют тип *тип_данных*. Обратите внимание, что при таком способе инициализации *число_элементов* можно опускать. Компьютер самостоятельно определит размер массива исходя из числа элементов в списке инициализации. Некоторые примеры:

```
float float_array[3] = {0.25, .876, 3.0};
char char_array[6] = {'П','р','и','в','е','т'};
```

Если не указано число элементов, но присутствует список инициализации, по умолчанию массив будет иметь размер, равный числу элементов в списке. В следующей строке кода `char_array2` автоматически получит размер `6`:

```
char char_array2[] = {'д','р','а','к','о','н'};
```

Если число элементов в списке инициализации меньше числа элементов массива, оставшиеся значения заполняются числом `0`. Например, конструкция

```
int my_array[5] = {1,2,3};
```

эквивалентна

```
int my_array[5] = {1,2,3,0,0};
```

Еще один способ инициализации – при помощи цикла `for`. Необходимо объявить массив стандартным образом, а затем заполнить его значениями в цикле. Пример:

```
int int_array[10];
for (int i = 0; i < sizeof(int_array)/sizeof(int); i++)
{
```

```
        int_array[i] = i;
}
```

Обратите внимание, что цикл повторяется, пока `i` не станет меньше значения `sizeof(int_array)/sizeof(int)`. Это просто иной способ перебора всех элементов массива. `sizeof(int_array)` возвращает число байтов, занимаемых массивом, а `sizeof(int)` – число байтов, занимаемых каждым элементом. В результате деления получается число элементов в массиве. В этом примере его размер уже известен, поэтому применение оператора `sizeof()` в некотором роде излишне; но в большинстве случаев невозможно предсказать, сколько элементов содержится в массиве.

Работа с массивами

Доступ к элементам массива осуществляется так же просто, как и его создание. Каждый элемент массива можно считать самостоятельной скалярной переменной. Чтобы получить доступ к отдельному элементу, следует воспользоваться именем массива и оператором индекса, указав номер элемента:

```
имя_массива[номер_индекса];
```

Такая конструкция эквивалента обычному обращению к скалярной переменной. К примеру, если объявить массив

```
char char_array[10];
```

для отображения значения четвертого элемента (индекс с номером `3`), следует использовать такой код:

```
cout << char_array[3];
```

Кроме того, доступ к значениям из массива можно осуществлять по указателю, о чем мы расскажем позже в этой главе в разделе «Связь массивов и указателей».

Многомерные массивы

Поначалу многомерные массивы могут быть сложными для восприятия, но можно облегчить себе жизнь, если считать их *массивами массивов*. На рис. 6.2 представлено пять первичных элементов (крупные коробки) массива, каждый из которых содержит массив из четырех элементов. В результате получается многомерный массив размером 5×4.

Для создания двумерного массива достаточно добавить еще один оператор индекса в конец объявления:

```
тип_данных имя_массива[число_элементов][число_элементов2];
```

Этот код создает массив из *число_элементов* массивов, каждый из которых состоит из *число_элементов2* элементов типа *тип_данных*. Например

```
int my_Array[5][4];
```

создает массив из пяти элементов, каждый из которых является массивом из четырех целых чисел. Представляйте элементы многомерных массивов коробками памяти, каждая из которых содержит массив; повторимся, такое представление проиллюстрировано на рис. 6.2.

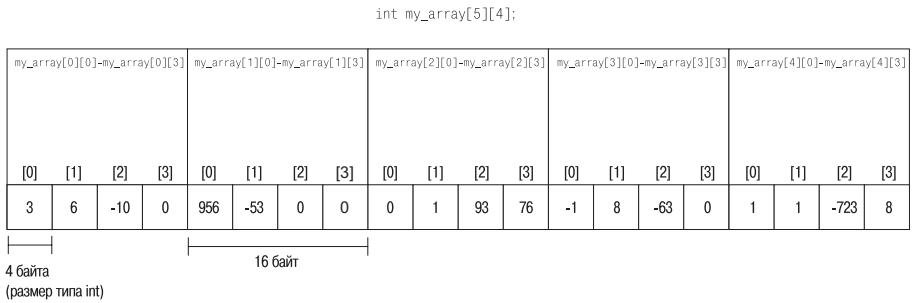


Рис. 6.2. *Двумерный массив – коробки в коробках. Внутри каждой из больших коробок хранится массив коробок поменьше. Массивы внутри массивов – это основание многомерности*

Массив может иметь произвольное число измерений. Однако слишком большое число измерений может быстро привести к переполнению памяти компьютера.

Любой многомерный массив может быть представлен в виде одномерного массива – перемножением всех его размеров. Для двумерного массива вроде этого:

```
char my_Array[10][10];
```

можно создать эквивалентный одномерный массив, умножив 10 на 10. Таким образом, массив

```
char my_Array[100];
```

имеет точно такое же общее число целочисленных элементов, как и предыдущий двумерный массив.

Распространенный метод перебора элементов многомерных массивов – вложенные циклы `for`. В следующем примере происходит инициализация всех элементов многомерного массива:

```
//6.1 - Многомерные массивы - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;
int main(void)
{
int numbers[10][10];
     for (int i = 0; i < sizeof(numbers)/(sizeof(int)*10); i++)
          for (int c = 0;c < sizeof(numbers)/(sizeof(int)*10); c++)
               cout << (numbers[i][c] = (i * 10) + c) << endl;
     return 0;
}
```

Вывод:

```
0
1
...
98
99
```

Работа с указателями

Указатели могут стать мощнейшим средством в умелых руках, но также являются одним из самых сложных в изучении инструментов C++. Однако мы обещаем, что не торопясь передадим читателям информацию по этой теме.

Простейшее определение указателя звучит так: *указатель* – это переменная, которая хранит адрес памяти. И хотя определение весьма простое, следствия из него – гораздо более сложные.

Адрес памяти, хранимый указателем, – это адрес другой переменной. Поэтому говорят, что указатель «указывает на переменную».

Синтаксис объявления указателей следующий:

```
тип* имя_указателя;
```

Здесь *тип* – это тип переменной, на которую происходит указание, *имя_указателя*, соответственно, – имя указателя. Но объявление – это только начало. Сложности заключаются в работе с указателями.

Во-первых, вам следует узнать о новом операторе, *операторе взятия адреса* (`&`). Этот оператор предшествует переменной и возвращает ее адрес в памяти. Этот оператор используется для присвоения значений указателям. Поясним на примере:

```
int my_int;
int* my_int_pointer = &my_int;
```

В результате выполнения приведенного кода `my_int_pointer` указывает на `my_int`. Вскоре вы сможете как-то использовать это обстоятельство, а пока просто изучите основы.

Обратите внимание, что если присвоить указателю значение `0`, он не указывает ни на какую переменную. Подобные указатели называются *нулевыми* или *неинициализированными*.

Для тех, кто испытывает сложности с этими понятиями, еще один пример:

```
//6.2 - Указатели - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;
int main(void)
{
```

```
    int an_int = 5;
    int* a_pointer = &an_int;
    cout << "Значение целочисленной переменной: " << an_int
         << "\nАдрес целочисленной переменной: " << &an_int
         << "\n3начение указателя: " << a_pointer
         << "\nАдрес указателя: " << &a_pointer;
}
```

Растем дальше: оператор разыменования

Полдела сделано. Теперь вы узнаете об операторе разыменования. *Оператор разыменования* (*) используется для обращения к объектам, на которые указывают указатели. Оператор разыменования предшествует имени указателя (либо выражению, которое вычисляется в указатель) и возвращает значение переменной, на которую указатель указывает. Этот оператор отличается от символа *, используемого для объявления указателей, и его семантику в каждом конкретном случае можно легко определить исходя из контекста. В объявлении указателя разыменование не происходит.

В предыдущем разделе встречался указатель a_pointer*, инициализированный при помощи символа* *, *но не следует путать оператор разыменования и объявление указателя. Несмотря на наличие символа* * *перед указателем в момент объявления, разыменование указателя на самом деле не происходит.*

Например, если объявить целочисленную переменную и указатель следующим образом:

```
int a = 78;
int* pb = &a;
```

оператором разыменования можно пользоваться для доступа к значению переменной a через указатель pb:

```
cout << "Значение a: " << *pb;
```

В результате выполнения этого кода выводится число 78. Оператор разыменования можно считать обратным оператору взятия адреса.

Проясним отношения операторов взятия адреса и разыменования при помощи такого примера:

```
int x;
int* px  = &x; // px указывает на x
*px = &x; // x содержит значение &x
```

Во второй строке px присваивается значение &x, а не *px, как в третьей. Обратите внимание, изменяя значение *px, мы изменяем значение x.

Можно также создавать указатели на указатели. Объявление для указателя на указатель на целое число выглядит так:

```
int** ppi;
```

Инициализация:

```
int * pi;
ppi = &pi;
```

Указатель на указатель хранит адрес указателя, который, в свою очередь, хранит адрес переменной. Таким образом, можно получить доступ к значению переменной через указатель на указатель. Достаточно дважды использовать оператор разыменования:

```
cout << **ppi;
```

В результате выполнения этого кода будет выведено значение, на которое указывает `pi`. Концепция указателя на указатель не всегда понятна сразу. Представьте себе дорожный знак, который указывает на другой дорожный знак, показывая вам его расположение.

`*ppi` – это значение указателя `pi`; второй оператор разыменования позволяет получить значение, на которое указывает `pi`. Можно развить эту идею и использовать многократные перенаправления и разыменования, но это будет не очень практично.

Указатели и объекты

Указатель может быть связан с объектом или структурой. Объявить указатель на строковой объект можно так:

```
string* ps;
```

Как видите, указатели этого вида не сильно отличаются от уже изученных – строками можно пользоваться так же, как и другими типами данных. Но следует отметить некоторые моменты. Во-первых, объект не создается, создается только указатель на объект. По этой причине конструктор `string` не вызывается (вызов конструктора в C++ происходит только при объявлении строки, поскольку она является классом, а не встроенным типом данных).

Во-вторых, если класс `Point` объявлен как:

```
class Point
{
public:
     int X,Y;
     Point (int lX, int lY) : X(lX), Y(lY){}
     void print();
};
void Point::print()
{
     cout << "Значение X: " << X
          << "\nЗначение Y: " << Y << endl;
}
```

и объект класса создан объявлением

```
Point p(5,3);
```

то указатель на этот объект можно создать так:

```
Point* pp = &p;
```

Доступ к компонентам этого объекта по указателю осуществляется следующим образом:

```
pp->X = 6;
pp->Y = 5;
```

В этом коде представлен новый оператор, *оператор выбора компонента* (`->`). Он ничем не отличается другого компонентного оператора (`.`), но предназначен для использования в указателях на объекты. Посредством этого оператора могут вызываться и компонентные функции:

```
pp->print();
```

Как можно догадаться, этот код приводит к выполнению метода `print`.

Указатель this

В каждом объекте класса существует неизменяемый указатель на этот объект, который называется указателем `this`. Указатель `this` позволяет обращаться к любым общим полям данных и функциям класса. Синтаксис использования этого указателя идентичен синтаксису для любого другого указателя на объект.

Например, метод `print()` класса `Point` (объявленного в предыдущем разделе) можно переписать с применением указателя `this` таким образом:

```
void Point::print()
{
     cout << "Значение X: " << this->X
          << "\nЗначение Y: " << this->Y << endl;
}
```

Это изменение не затрагивает семантику метода. `this` в данном случае – просто способ явным образом выразить, о каком экземпляре класса идет речь.

Интересно, что при помощи `this` можно получить значение объекта:

```
Point p = *this;
```

На данном этапе эта информация для читателей бесполезна, но может пригодиться в будущем.

Связь массивов и указателей

Массивы и указатели находятся в близких родственных отношениях. Если брать строгое определение, массивы являются указателями. Рассмотрим такой массив:

```
float f[10];
```

Имя массива, `f`, действует в качестве указателя на первый элемент. Следующий код отображает значение первого элемента:

```
cout << *f;
```

Если объявить указатель на тип элементов массива, можно присвоить ему адрес первого элемента массива следующим образом:

```
float* pf = f;
```

Этот фрагмент кода эквивалентен следующему:

```
float* pf = &f[0];
```

А теперь можно перебрать элементы массива, увеличивая указатель. Внимание! Приращение указателя выполняется ровно так же, как и для всякой другой переменной. Раз указатель хранит адрес памяти, приращение указателя изменяет адрес. Но значение адреса увеличивается не на единицу, оно увеличивается на размер типа указателя. Это позволяет переходить к следующему элементу массива, как в этом примере:

```
float f[] = {5.5, 0.5 , 6.7};
float* pf = f;
cout << *pf;
cout << " " << *(++pf);
cout << " " <<*(++pf);
```

Код выводит на экран `5.5 0.5 6.7`. Каждое приращение указателя переводит его на следующий элемент массива. Если выполняется оператор инкремента для указателя, C++ предполагает, что указатель указывает на массив, но не проверяет это обстоятельство, поэтому будьте внимательны и выполняйте приращения только для указателей на элементы массивов.

Точно так же действует оператор декремента (`--`), который мы изучили в главе 2 «Продолжаем погружение: переменные». Каждое уменьшение указателя перемещает его на предшествующий элемент массива. Старайтесь не выходить за пределы массива – подобные ошибки очень трудно обнаружить и устранить.

Кроме того, можно выполнять операции сложения и вычитания для указателей. Если `p` является указателем на массив, `p+`*n* выполняет переход на *n* элементов вперед, а `p-`*n* – на *n* элементов назад.

Вот пример перебора элементов массива посредством указателя:

```
int n[] = {0,1,2,3,4,5};
int* pn = n;
cout << *(pn+3) << endl;
cout << *++pn << endl;
*pn--;
cout <<*pn << endl << *(pn+4);
```

Вывод:

```
3
1
0
4
```

C++ не запрещает доступ к значениям, которые не принадлежат массиву. К примеру, если в предыдущем примере увеличить указатель на массив `n[]` *больше, чем на* `5`*, C++ с радостью вернет все данные, которые компьютер хранит по этому адресу. Более того, если эти данные нужны для работы Windows или другой программы, попытка поработать с ними может привести к сбою всей системы. Доступ к данным, не принадлежащим массиву, называется* ***выходом за пределы массива****.*

Неизменяемые указатели и указатели на константы

Значение адреса памяти, хранимое *неизменяемым указателем*, не может быть изменено. Однако может быть изменено значение переменной, на которую производится указание. Чтобы создать неизменяемый указатель, предварите его имя ключевым словом `const`, как показано ниже:

```
char* const p; // неизменяемый указатель на char
```

В результате выполнения кода создается неизменяемый указатель. Однако неизменяемый указатель является разновидностью константы, а потому должен быть инициализирован в момент объявления:

```
char p;
char* const pc = &p;
```

После этого адрес памяти, хранимый в `pc` (`&p`), уже не может быть изменен. Но может быть изменено значение `p` при помощи `pc`:

```
*pc = 'd'; //допустимо, изменяет p
pc = 0; //недопустимо
```

Кроме того, можно создавать *указатели на константы*. Указатель на константу может изменять хранимый адрес памяти, но не константу, на которую указывает. Чтобы объявить указатель на константу, следует предварить оператор `*` ключевым словом `const`, как показано ниже:

```
char const* pcc; // указатель на постоянное значение char
const int* pci; // указатель на постоянное значение int
```

В последнем фрагменте кода pcc может указывать на различные символы, но не может изменять их значения. Указатели на константы не обязательно инициализировать в момент объявления, поскольку указатель остается изменяемым, в отличие от переменной, на которую указывает:

```
char c;
pcc = &c;
*pcc = 'd'; // недопустимо
pcc = 0; // допустимо
```

Кроме того, можно создавать неизменяемые указатели на константы, которые не позволяют изменять адрес памяти и значение объекта указания. Чтобы создать подобный указатель, следует поместить ключевое слово const перед оператором * и после него:

```
int x;
const int* const cpci = x;
```

Единственное, что можно сделать при помощи cpci, – прочитать значение указателя и значение переменной, на которую он указывает (x).

Указатели и функции

Указатели незаменимы в качестве параметров функций и возвращаемых значений. Применение параметра в качестве аргумента функции может сделать программу гораздо более эффективной.

В обычной ситуации при передаче аргумента функции создается новая копия этого аргумента, значение которой и присваивается соответствующему параметру. Однако если параметром является указатель, необходимо скопировать только адрес памяти, что в случае объемных типов данных может сэкономить массу времени на копировании.

Другим преимуществом указателей-параметров функций является возможность изменять оригинал, а не полученную копию. Взгляните на пример того, что *не* следует делать:

```
#include <iostream>
using namespace std;
class Point
{
     public:
          int X = 0,Y=0;
     Point() : X(0), Y(0) {}
}
Point MoveUp(Point p)
{
     p.Y+=5;
     return point;
}
```

```
int main(void)
{
      Point point;
     point = MoveUp(point);
     cout << point.X << point.Y;
     return 0;
}
```

Вывод:

```
05
```

Код не очень хорош по двум причинам. Во-первых, для вызова функции `MoveUp()` полный объект точки должен быть скопирован дважды. Во-вторых, строка `point = MoveUp(point);` выглядит довольно неуклюже. С точки зрения проектирования, пользователю класса осталось слишком много работы.

Но если параметром `MoveUp()` сделать указатель, функцию можно значительно улучшить:

```
//6.3 - Передача указателя - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;
class Point
{
     public:
          int X,Y;
     Point() : X(0), Y(0) {}
};
void MoveUp(Point* p)
{
     p->Y+=5;
}
int main(void)
{
      Point point;
     MoveUp(&point);
     cout << point.X << point.Y;
     return 0;
}
```

Вывод:

```
05
```

Эффективность функции `MoveUp()` значительно выросла. Теперь вместо создания двух копий объекта `Point` при каждом вызове создается лишь одна копия адреса памяти. Кроме того, существенно упростился и вызов `MoveUp()`. Теперь нет неуклюжего оператора присваивания, мы просто вызываем метод. Что касается пользователей функции, такая конструкция гораздо проще для понимания.

И снова строки

Мы уже говорили о строках (в частности, в главе 1), но затронули лишь самые основные понятия, связанные с этим типом данных. В этом разделе мы возвращаемся к ним, чтобы более внимательно изучить работу с ними и их возможности.

Наступил момент, когда читатели готовы к изучению более сложных концепций, связанных со строками. В языке C строки представляются символьными массивами, а не объектами, как в C++. По этой причине в большинстве программ вы будете встречать именно символьные массивы. Соответственно мы рассмотрим строки в стиле C более подробно, чтобы вы могли читать любой код на C++.

Строковые литералы

Официальный тип строковых литералов, таких как `"Привет"`, — `const char []`. Строка `"Привет"` имеет тип `const char [7]`. Секундочку! Ведь в слове *Привет* шесть букв, а не семь! Не переживайте, ошибки здесь нет. Дополнительный элемент содержит завершающий пустой символ, `'\0'` (значение `0`), который сообщает компьютеру длину строки.

В конце каждого строкового литерала есть скрытый пустой символ (`'\0'`), который позволяет алгоритмам определять длину строки и факт достижения конца строки. Длина строки нужна не всем алгоритмам, но большинству из них.

Присвоить строковый литерал переменной типа `char*` можно следующим образом:

```
char* x = "Привет";
```

Однако со значением, на которое указывает `x`, нельзя работать. Если оно не будет постоянным, возникнет ошибка, как в следующем примере:

```
*x = 'T';
```

Этот код вызывает ошибку, потому что *нельзя* изменять значение константы. Чтобы получить строку, которую можно изменять, следует присвоить строковый литерал строковому объекту или символьному массиву. Пример объявления символьного массива, элементы которого могут изменяться:

```
char s[] = "Привет";
s[0] = 'T';
```

С этим кодом все в порядке, а новым значением строки будет `"Тривет"`.

Многие из стандартных библиотечных функций C принимают в качестве одного из аргументов значение типа `char*`. Однако если строка хранится в виде указателя на символ, ее длина не может быть определена, как в случае строковых литералов, завершаемых пустым символом. В отсутствие пустого символа невозможно понять, где кончается строка.

Символьные массивы

Как вы видели, массивы символов – это еще один способ представления строк. Массив символов можно инициализировать строковым литералом, как показано в следующем примере:

```
char s[] = "Здравствуй, мир!";
```

Поскольку нет необходимости указывать длину строки в операторе индекса, этот способ объявления строки является почти столь же удобным, как применение класса `string` стандартной библиотеки. C++ выделяет под хранение массива `s` (включая пустой (нулевой) символ в конце строки) достаточный объем памяти.

Вычисление длины строки

Чтобы получить длину строки `s[]` (из предшествующего раздела «Символьные массивы»), можно воспользоваться стандартным способом определения длины массива, то есть оператором `sizeof`. Для строки `s` из примера предшествующего раздела вызов может выглядеть так:

```
cout << sizeof(s);
```

Этот фрагмент кода выводит на экран число `17`. Мы не выполняли деление на размер типа `char`, как это обычно делается при вычислении длины массива, поскольку размер типа `char` всегда равен 1 байту.

У этого метода есть недостаток. Длина строки на самом деле `16` символов, а не `17`. При вычислении размера посредством оператора `sizeof` в длину строки включается пустой символ. Чтобы вычислить длину строки без пустого символа, можно вычесть единицу из полученного значения либо воспользоваться стандартной библиотекой.

Функция по имени `strlen()` возвращает длину любой строки. Например, чтобы выяснить длину строки `s` (`char s[] = "Здравствуй, мир!";`), воспользуйтесь конструкцией:

```
cout << strlen(s);
```

Этот код отображает корректную длину строки, `16`, а не `17`, поскольку `strlen()` учитывает только элементы до первого пустого символа. Для строки вроде этой:

```
char weird_string = "Здравствуй\0 мир";
```

`strlen()` возвращает `10`, учитывая лишь символы, предшествующие пустому. Но если вычислять длину этой строки при помощи оператора `sizeof`, мы получим `16` (два пустых символа).

Прототип функции `strlen()` выглядит следующим образом:

```
int strlen(const char*)
```

Чтобы воспользоваться функцией, включите в свою программу библиотеку `<string>`.

Прочие функции для строк в стиле C

Стандартная библиотека содержит и многие другие функции для работы со строками в стиле C. В этом разделе мы рассмотрим некоторыми из них, просто чтобы познакомить читателей с потенциальными возможностями. Речь пойдет о функции `strcopy()`, предназначенной для копирования строк, и `strcat()`, выполняющей слияние строк.

Чтобы воспользоваться этими функциями, следует включить `<cstring>` или `<string.h>`. Чтобы скопировать одну строку в другую есть функция `strcpy()` (string copy). Ее прототип:

```
char* strcpy(char* p, const char* q);
```

Эта функция копирует все элементы `q` в `p`. Например, следующий вызов:

```
char s[7];
strcpy(s, "Привет");
```

приводит к сохранению в `s` значения `"Привет"` (включая и завершающий пустой символ). Значение `char*`, возвращаемое этой функцией, является значением `q` (скопированной строки).

Эта функция не проверяет, имеет ли массив `p` размер, достаточный для сохранения всех значений `q`. Она просто копирует. Как следствие, всегда следует проверять, что массив `p` имеет размер, по меньшей мере равный размеру `q`.

Пример копирования строк посредством функции `strcpy()`:

```
char s[7];
char t[] = "Привет";
cout << strcpy(s, t);
```

Этот код выводит `"Привет"` на экран и копирует строку «Привет» в массив `s`.

Чтобы произвести *слияние* (*сцепление*, *конкатенацию*) двух строк (добавить одну в конец другой), воспользуйтесь функцией `strcat()`. Ее прототип выглядит следующим образом:

```
char* strcat(char* p, const char* q);
```

Функция добавляет строку `q` в конец строки `p`. После выполнения такого кода:

```
char s[16] = "Здравствуй,";
cout << strcat(s, " мир");
```

`s` хранит значение `"Здравствуй, мир"` (включая завершающий пустой символ). Значение `char*`, возвращаемое этой функцией, является результатом слияния строк. В этом примере на экран выводится текст `"Здравствуй, мир"`.

Функция strncpy() действует так же, как strcpy(), но копирует лишь определенный объем элементов q в p. Прототип функции strncpy():

```
char* strncpy(char* p, const char* q, int n);
```

Функция копирует n символов из q в конец p. В результате выполнения кода:

```
char s [8]= "Ну и ";
char t[]= "ну";
cout << strncpy(s, t, 2);
```

s принимает значение «Ну и ну». Кроме того, эта строка выводится на экран, поскольку дополнительно возвращается функцией strncpy().

Преобразование строк в числа

Существует также пара функций, позволяющих производить преобразование строк, содержащих численные значения, в численные значения (например, "5" в 5). Эти функции объявлены в файлах <cstdlib> и <stdlib.h>.

Чтобы преобразовать строковое представление целого числа в целое число, воспользуйтесь функцией atoi(). Прототип этой функции:

```
int atoi(const char* p);
```

Аргументом функции является строка, возвращаемым значением – целое число. Пример:

```
char s[] = "567";
int x = atoi(s) + 3;
cout << x;
```

В результате выполнения кода на экран выводится число 570.

Существуют также функции для преобразования строки в значение double и строки в значение long: atof() и atol() соответственно. Их прототипы почти идентичны прототипу atoi():

```
double atof(const char* p);
long atol(const char* p);
```

Эти функции работают точно так же, как atoi().

Если строка не содержит числа (скажем, строка "Привет"), возвращается значение 0.

Эти функции могут использоваться для обработки ввода пользователя. Не полагаясь на корректный ввод чисел пользователями, мы можем использовать для хранения ввода строки, а затем проверять, могут ли эти строки быть преобразованы в числа. Такой способ получения ввода пользователя является существенно более безопасным.

Знакомимся со ссылками

Вы проделали долгий путь. Поздравляем! Теперь вы готовы узнать о работе со ссылками. *Ссылка* – это псевдоним, или альтернативное имя, отдельной переменной. Ссылку можно считать неизменяемым указателем, который заранее разыменован. Ссылки похожи на константы в том смысле, что должны инициализироваться при объявлении, а их значения не могут изменяться впоследствии.

Для создания ссылок используется оператор `&`. Не путайте этот оператор с оператором взятия адреса. Определять семантику в каждом конкретном случае можно исходя из контекста. Синтаксис объявления ссылки:

```
тип_данных& имя_ссылки;
```

Здесь *тип_данных* – это тип переменной, на которую указывает ссылка, а *имя_ссылки* – соответственно имя ссылки. Чтобы инициализировать ссылку (действие обязательное при объявлении), достаточно присвоить ей в качестве значения переменную типа *тип_данных*:

```
тип_данных& имя_ссылки = переменная;
```

Пример создания и инициализации ссылки:

```
int x ;
int& rx = x;
```

В результате выполнения этого кода мы получаем `rx`, ссылку на переменную `x`.

Может показаться необычным, но операторы действуют не на ссылку. Они действуют на переменную, связанную со ссылкой. Такое приращение `rx`:

```
rx++;
```

приводит к увеличению значения `x` на единицу. `rx` – просто еще одно имя для `x`.

Ссылки используются в основном в качестве аргументов функций и возвращаемых значений.

Ссылки в параметрах функций

Любой параметр функции может стать ссылкой на законных основаниях. Это может быть полезно для создания функций, изменяющих полученные аргументы. Например, можно создать функцию `decrement` с параметром-ссылкой:

```
void decrement(int& x)
{
```

```
    x--;
}
```

А затем воспользоваться функцией следующим образом:

```
int a = 5;
decrement(a);
cout << a;
```

Вывод:

```
4
```

Поскольку в момент вызова этой функции x становится ссылкой на a, уменьшение x изменяет значение a. Как видите, такой способ гораздо удобнее, чем применение указателей для тех же целей.

Ссылки в качестве возвращаемых значений функций

Как можно догадаться, функция может возвращать ссылку на переменную. Однако последствия подобных действий не всегда очевидны. Рассмотрим пример.

```
//6.4 - Пример: ссылки - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
using namespace std;
class Point
{
     int X,Y;
     public:
          Point(int lX, int lY):X(lX),Y(lY) {}
          int& GetX() {return  X;}
          int& GetY() {return Y;}
};
int main(void)
{
     Point p(5,3);
     p.GetX() = 3;
     p.GetY() = 5;
     cout << p.GetX() << p.GetY();
}
```

Вывод:

```
35
```

Поскольку GetX() и GetY() возвращают ссылки на компонентные переменные X и Y, вызовы этих функций могут использоваться в качестве именующих выражений (*lvalues*) (именующее выражение – это значение, или переменная, которое может изменяться). Изменение значений этих ссылок изменяет только значения компонентных переменных X и Y.

Динамическая память

До сих пор мы хранили данные только в статической и автоматической памяти. *Статическая память* – это область хранения всех глобальных и статических переменных. Переменные статической памяти объявляются лишь единожды и уничтожаются по завершении программы. *Автоматическая память* хранит аргументы функций и локальные переменные. Элементы данных, хранимые в автоматической памяти, создаются и уничтожаются по необходимости.

Третий вид памяти: *память свободного хранения* (или *динамическая память*). Программа должна явным образом запросить память для элементов, хранимых в этой области, а затем освободить память, если она больше не нужна. Навыки эффективного применения динамической памяти приобрести нелегко, но они исключительно полезны.

Чтобы успешно пользоваться памятью свободного хранения, необходимо понимать некоторые тонкие моменты. Во-первых, следует явным образом запрашивать *выделение* памяти (память выделяется под хранение переменной и отмечается как таковая, если найдено свободное пространство в памяти) для хранения данных при помощи оператора `new`. Во-вторых, следует *освобождать* выделенную память (в результате с области памяти снимается маркер использования) при помощи оператора `delete`.

Все элементы данных, хранимых в динамической памяти, существуют до момента освобождения памяти либо до конца программы. Как следствие, хранимые в динамической памяти переменные не уничтожаются при выходе из области видимости. Это полезная особенность, поскольку позволяет создать переменную внутри функции, а использовать за пределами функции. Большинство программистов-профессионалов используют динамическую память практически постоянно, по привычке.

Выделение динамической памяти, операторы new и delete

Как мы упоминали, выделение памяти выполняется с помощью оператора `new`. Этот оператор возвращает указатель на выделенную память. Синтаксис оператора `new`:

```
new тип_данных;
```

Здесь *тип_данных* – любой допустимый тип данных или класс. Чтобы воспользоваться памятью, необходимо присвоить полученное значение переменной-указателю. Именно в работе с динамической памятью указатели могут себя проявить. Например, создать целое число в области свободного хранения можно следующим образом:

```
int* a = new int;
```

Вот, собственно говоря, и все. Оператор new исключительно прост в применении. С-эквивалентом этого оператора является функция malloc(), объявленная в файле <cstdlib> (или <stdlib.h>):

```
int* a = (int*)malloc(sizeof(int));
```

Как видите, этот вариант значительно более запутанный. Особого смысла использовать функцию malloc() нет, но следует знать, что такая функция существует – на случай встречи с ней в чужой программе.

Вся память, выделенная оператором new, должна освобождаться оператором delete. В противном случае эта память не будет доступна для выполнения других операций. Когда программист забывает освободить выделенную динамическую память, возникает так называемая *утечка памяти*, которой следует избегать всеми доступными способами.

Оператор delete выполняется над указателем на раздел памяти, выделенный оператором new, либо нулевым указателем. Во всех остальных случаях возникает ошибка. Синтаксис оператора delete:

```
delete указатель;
```

Чтобы освободить память, выделенную в предыдущем примере, выполните:

```
delete a;
```

Освободив всю память, выделенную оператором new, обнулите значение указателя. Это гарантирует, что вы случайно не попытаетесь освободить уже освобожденную память. Выполнение оператора delete над нулевым указателем безопасно, поскольку не связано с какими-либо действиями.

В случае создания динамических объектов в классе лучшая точка для освобождения памяти этих объектов – код деструктора класса. Необходимость освобождения динамически выделенной памяти является одной из главных причин существования деструкторов в C++.

Создание динамических массивов

Обычно при создании массива необходимо указать константное выражение для числа элементов. Если попытаться использовать выражение с переменными, возникнет ошибка компиляции. Например, на следующий код:

```
int x = 5;
char s[x];
```

CodeWarrior среагирует ошибкой, сообщив, что ожидается константа.

При создании динамического массива можно использовать переменное значение для числа элементов:

```
int x = 5;
char* s = new char[x];
```

Чтобы освободить память созданного массива, воспользуйтесь оператором `delete[]`. `delete[]` работает в точности как `delete`:

```
delete[] s;
```

Этот код освобождает память всего массива, и нет необходимости думать о каждом элементе в отдельности.

Воссоздаем крестики-нолики

А сейчас вам предстоит воспользоваться приобретенным опытом работы с массивами и указателями: мы вместе создадим нестареющую классику: игру «Крестики-нолики». Чтобы сделать код более прозрачным, мы не старались предельно оптимизировать игру. Позже вы можете вернуться к этой игре и попробовать ее усовершенствовать.

```
//6.5 - Крестики-нолики - Дирк Хенкеманс и Марк Ли - Premier Press
#include <iostream>
#include <string>

using namespace std;      //introduces namespace std
enum SquareState { blank = ' ', X = 'X', O='O'};
class gameBoard
{
     private:
         const int WIDTH;
         const int HEIGHT;
         int* GameBoard;
     public:
          gameBoard() : WIDTH(3), HEIGHT(3)
          {
               GameBoard = new int[9];
               for (int i = 0; i < 9; i++)
                    *(GameBoard + i) = blank;
          }
         ~gameBoard() {delete[] GameBoard;}
         void setX(int h, int w);
         void setO(int h, int w);
         bool isTaken(int h, int w);
         SquareState isLine();
         void draw();
};

void gameBoard::setX(int h, int w)
{
     *(GameBoard + h*HEIGHT + w) = X;
}

void gameBoard::setO(int h, int w)
{
     *(GameBoard + h*HEIGHT + w) = O;
```

```
}

bool gameBoard::isTaken (int h, int w)
{
     return *(GameBoard + h*HEIGHT + w) != ' ';
}

SquareState gameBoard::isLine()
{
     if(*GameBoard==X && *(GameBoard +1)==X && *(GameBoard +2)==X)
          return X;
     if(*GameBoard==0 && *(GameBoard +1)==0 && *(GameBoard +2)==0)
          return 0;
     if(*(GameBoard +3)==X && *(GameBoard +4)==X && *(GameBoard +5)==X)
          return X;
     if(*(GameBoard +3)==0 && *(GameBoard +4)==0 && *(GameBoard +5)==0)
          return 0;
     if(*(GameBoard +6)==X && *(GameBoard +7)==X && *(GameBoard +8)==X)
          return X;
     if(*(GameBoard +6)==0 && *(GameBoard +7)==0 && *(GameBoard +8)==0)
          return 0;

     if(*GameBoard==X && *(GameBoard +3)==X && *(GameBoard +6)==X)
          return X;
     if(*GameBoard==0 && *(GameBoard +3)==0 && *(GameBoard +6)==0)
          return 0;
     if(*(GameBoard +1)==X && *(GameBoard +4)==X && *(GameBoard +7)==X)
          return X;
     if(*(GameBoard +1)==0 && *(GameBoard +4)==0 && *(GameBoard +7)==0)
          return 0;
     if(*(GameBoard +2)==X && *(GameBoard +5)==X && *(GameBoard +8)==X)
          return X;
     if(*(GameBoard +2)==0 && *(GameBoard +5)==0 && *(GameBoard +8)==0)
          return 0;

     if(*GameBoard==X && *(GameBoard +4)==X && *(GameBoard +8)==X)
          return X;
     if(*GameBoard==0 && *(GameBoard +4)==0 && *(GameBoard +8)==0)
          return 0;
     if(*(GameBoard +2)==X && *(GameBoard +4)==X && *(GameBoard +6)==X)
          return X;
     if(*(GameBoard +2)==0 && *(GameBoard +4)==0 && *(GameBoard +6)==0)
          return 0;
     return blank;

}

void gameBoard::draw()
{
     cout << endl;
     for(int i=0; i < HEIGHT; i++)
     {
          cout << (char)*(GameBoard + i*HEIGHT);
          for(int c=1; c < WIDTH; c++)
```

```
                cout << " | " << (char)*(GameBoard + i*WIDTH + c);
            cout << endl << "-----------" << endl;
        }
}

class Game
{
public:
    gameBoard* doInput(string player, gameBoard* gb);
    bool inRange(int test);
};

gameBoard* Game::doInput(string player, gameBoard* gb)
{
    gb->draw();

    string letter;
    if (player.compare("один") == 0)
        letter = "X";
    else if (player.compare("два") == 0)
        letter = "O";
    else return gb;

    int input1, input2;

    do {

        do {
             cout << "\nИгрок " << player.c_str()
                 << ", пожалуйста, введите номер ряда для "
                 << letter.c_str() << ": ";
            cin >> input1;
        }while(!inRange(input1));

        do {
            cout << "\nПожалуйста, введите номер колонки для "
                 << letter.c_str() << ": ";
            cin >> input2;
        }while(!inRange(input2));

    }while (gb->isTaken(input1,input2));

    if (player.compare("один") == 0)
        gb->setX(input1, input2);
    else gb->setO(input1, input2);

    return gb;

}

bool Game::inRange(int test)
{
    return test > -1 && test < 3;
}

int main( void )
{
```

```
    gameBoard* gb = new gameBoard;
    Game g;
    string player1, player2;
    cout << "Добро пожаловать в Крестики-нолики!"
         << "\nИгрок один, введите свое имя: ";
    cin >> player1;
    cout << "\nИгрок два, введите свое имя: ";
    cin >> player2;

    while (gb->isLine() == C C)
    {
         gb = g.doInput("один",gb);
         gb = g.doInput("два",gb);
    }
    gb->draw();
    if(gb->isLine() == X)
         cout << "\nИгрок один, вы победили!"
              << "\nКонец игры.";
    else cout << "\nИгрок два, вы победили!"
              << "\nКонец игры.";
    return 0;
}
```

Резюме

В этой главе вы изучили много новых понятий. Прежде всего, вы узнали об указателях и о том, каким образом они являются основой массивов, затем изучили ссылки и динамическую память. И наконец, снова вернулись к строкам, чтобы узнать разницу между строками в стиле C и уже привычными строками C++. Но приключение еще не завершено. Настало время продолжить его в главе 7 «Градостроение и пространства имен».

Задания

1. Какой размер имеет строка “Здравствуй, мир”? Какова длина массива s?

```
char s[] = "Здравствуй, мир";
```

2. Приведите пять причин, по которым необходимо использовать указатели.
3. Какие проблемы существуют в игре «Крестики-нолики», приведенной в конце главы? Как можно улучшить игру?
4. Приведите три причины, по которым целесообразно использовать динамическую память.

7

Градостроение и пространства имен

Представьте, что вы отправились в неизведанные земли и в пути приобрели последователей. Последователи устали и решили основать новый город на берегах великой реки. Но вы не знаете, как назвать поселение, поскольку не знакомы с другими городами этой страны и не желаете выбирать уже существующее имя. Очевидное решение – провозгласить новое государство, чтобы имя нового поселения было гарантированно уникальным в его пределах. В программировании точно так же можно давать разным переменным одинаковые имена, разделяя их пространствами имен.

В этой главе вы узнаете:

- О назначении пространств имен
- Как объявлять пространства имен
- О действии дублирующихся пространств имен
- Об именованных и безымянных пространствах
- Какую пользу приносят пространства имен
- О стандартном и глобальном пространствах имен

Пространства имен

Самый простой способ разобраться в концепции пространств имен – думать о них, как о классах. С помощью класса (см. главу 5 «Боевые свойства ООП») можно создавать небольшую частную зону (область видимости), в которой объявляются общие (public) и частные (private) переменные. Затем для указания принадлежности объекта классу или глобальной области видимости используется оператор разрешения контекста (::), который описан в главе 4 «Пишем функции». Точно так же пространства имен используются для разбиения области видимости на несколько зон.

Объявление пространства имен

Синтаксис объявления пространства имен схож с синтаксисом объявления класса. Объявление пространства имен – это назначение имени области видимости, в которую будут входить компоненты пространства имен. В качестве аналогии можно привести код района для телефонного номера, который позволяет различать одинаковые телефонные номера.

Общий синтаксис для объявлений пространств имен:

```
namespace имя
{
     состав
}
```

Здесь *состав* может включать функции, классы и переменные – по сути дела, все, что может быть частью глобального пространства имен.

В одной области видимости не могут существовать два одинаковых идентификатора. Например, в одной программе невозможно создать две функции `fire()`, одна из которых зажигала бы факел, а вторая производила стрельбу из арбалета, кроме как с помощью пространств имен. Пространства имен позволяют разместить две функции в различных областях видимости, не создавая сложностей для компилятора. Более того, можно разместить все функции, связанные с определенной функциональностью, в отдельном пространстве имен. Например, функция `fire()`, зажигающая факел, попадет в пространство имен `exploration` (исследование) наряду с другими функциями исследования, а функция `fire()`, ответственная за стрельбу из арбалета, попадет в пространство имен `combat` (боевое).

Вот пример кода для двух различных функций `fire()`:

```
//хранит все боевые функции, классы и переменные
namespace combat
{
     void fire()
     {
          cout << "Ты стреляешь из арбалета." << endl;
     }
}

//хранит все исследовательские функции, классы и переменные
{
     void fire()
     {
          cout << "Ты зажег факел." << endl;
     }
}
```

Другой пример организации пространств имен приведен на рис. 7.1. Целочисленная переменная `subGlobalInt` объявляется в пространстве

имен `subGlobal`. Это объявление делает `subGlobalInt` частью пространства имен `subGlobal`. По этой причине при обращении к `subGlobalInt` извне `subGlobal` следует предварять переменную идентификатором пространства имен (`subGlobal::subGlobalInt`).

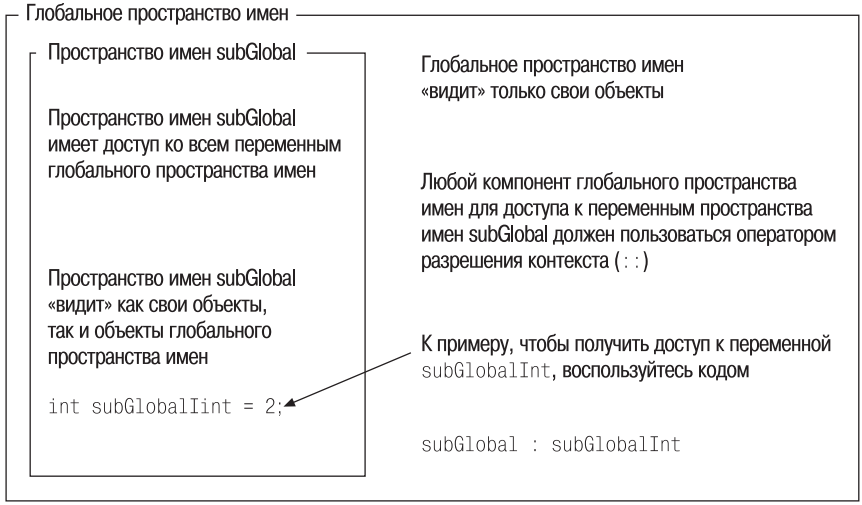


Рис. 7.1. *В глобальной области видимости существует пространство имен subGlobal. Чтобы получить доступ к данным subGlobal из глобального пространства имен, необходимо воспользоваться оператором разрешения контекста, но из пространства subGlobal можно получать доступ к любым объектам*

Пространства имен в действии

Доступ к компонентам пространств имен (функциям, классам и переменным) организован посредством оператора разрешения контекста. Синтаксис идентичен синтаксису для классов:

```
пространство_имен::компонент;
```

Например, чтобы вызвать обе функции `fire()` из предыдущего примера, можно написать нечто похожее на такой код:

```
//7.1 - Функции Fire - Дирк Хенкеманс - Premier Press
#include <iostream>
using namespace std;

//хранит все боевые функции, классы и переменные
namespace combat
{
     void fire()
     {
          cout << "Стрела со свистом"
               " срывается с твоего арбалета." << endl;
     }
```

```
}

//хранит все исследовательские функции, классы и переменные
namespace exploration
{
     void fire()
     {
          cout << "Пылает зажженный тобой факел." << endl;
}
}

int main(void)
{
     combat::fire();
     exploration::fire();
     return 0;
}
```

Вывод:

```
Стрела со свистом срывается с твоего арбалета.
Пылает зажженный тобой факел.
```

Знакомьтесь, глобальное пространство имен

Глобальное пространство имен – это официальное название области видимости самого высокого уровня (той, в которой существуют глобальные переменные). Чтобы обратиться к компоненту глобального пространства имен, иногда приходится пользоваться оператором разрешения контекста (::). Это происходит в случаях, когда существуют совпадающие идентификаторы в локальном и глобальном пространствах имен, поскольку по умолчанию всегда выбирается переменная с наименьшей областью видимости. Синтаксис обращения к компоненту глобального пространства имен следующий:

```
::глобальныйКомпонент;
```

Истории из жизни

Помимо очевидной пользы пространств имен для разделения кода игр на более управляемые сегменты их можно применять и для создания универсальных баз данных. Например представьте, что вы работаете в библиотеке и появилась необходимость создать базу данных по книгам библиотечного фонда. Вы используете компьютерное приложение для инвентаризации, в котором для сортировки книг различных жанров используются различные функции сортировки, и столкнулись с проблемой – невозможно назвать обе версии метода именем `sort()`. Проблема решается размещением двух функций в различных пространствах имен.

Повторные объявления пространств имен

Если два пространства имен имеют одинаковые имена, второе считается продолжением первого. Компьютер считает идентичными конструкции:

```
namespace x
{
     func1() {}
}
namespace x
{
     func2() {}
}
```

и

```
namespace x
{
     func1() {}
     func2() {}
}
```

C++ помещает оба пространства имен в одну область видимости при компиляции программы, и это означает, что в дублирующихся пространствах имен не могут существовать компоненты с одинаковыми именами.

Прямой доступ к пространствам имен

Если вы точно знаете, что пространство имен не содержит дублирующих идентификаторов, то можете произвести слияние с глобальным пространством имен. Это позволяет сэкономить время, поскольку исчезает необходимость постоянно набирать идентификатор пространства имен и оператор разрешения контекста.

Существует два метода прямого доступа к пространству имен:

- Объявление `using`
- Директива `using`

Объявление using

Объявление `using` показывает, что вы намереваетесь работать с определенным компонентом области видимости более низкого уровня. В результате отпадает необходимость в явном указании области видимости. Синтаксис объявления `using` следующий:

```
using имяПространства::компонент;
```

Пример использования объявления `using`:

```
namespace dragon{int gold = 50;}
```

```
int main(void)
{
     using dragon::gold;
     cout << gold;
}
```

Объявление `using` в данном случае позволяет не уточнять имя `gold` идентификатором `dragon`.

Директива using

Директива `using` показывает, что вы собираетесь работать со всеми компонентами конкретного пространства имен. Директива работает так же, как объявление, с той разницей, что обеспечивает прямой доступ ко всем компонентам пространства имен. Воспользовавшись ею, вы избавляетесь от необходимости уточнять имена компонентов этого пространства до конца программы. Перед ее применением следует убедиться, что во включаемом пространстве имен отсутствуют идентификаторы, дублирующие идентификаторы глобального пространства имен. Синтаксис директивы `using`:

```
using namespace имяПространства;
```

Пример использования этой конструкции вы уже неоднократно встречали, он присутствует во всех написанных к этому моменту программах:

```
using namespace std;
```

Более подробно мы расскажем об этой строке чуть позже в разделе «И снова пространство имен std».

Создаем функции меню

В каждой программе вы создавали меню с нуля. Настало время создать общую программу меню, которую можно будет использовать многократно. Способность к повторному использованию кода – это основа *полиморфизма* (принципа ООП, согласно которому каждый объект может использоваться более чем в одной программе; за дополнительными сведениями обращайтесь к главе 5). Функция `menu()` содержится в пространстве имен `menuNamespace` потому, что функция `menu` встречается часто, а мы хотим избежать конфликтов вспомогательной функции `menu()` с функцией `menu()` основной программы.

Приведенный здесь код состоит из трех файлов. Файлы отмечены комментариями, позволяющими определить их имена и границы. Код должен содержаться в трех различных файлах, иначе он не будет работать. Включите первый файл, hello.cpp, в свой проект, и при компиляции проекта остальные файлы будут включены автоматически благодаря операторам `#include`.

```
//7.2 - Функции меню, полигон (hello.cpp)
//Дирк Хенкеманс - Premier Press
#include <iostream>
#include "MenuUtility.h"
using namespace std;      //introduces namespace std

int main( void )
{
     using namespace menuNamespace;

     string example[] = {"атаковать","отступать"};
     menu(example, 2);

     return 0;
}
```

Далее следует код заголовочного файла для функции меню. Сохраните этот код в файле MenuUtility.h.

```
//7.3 - MenuUtility.h - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
using namespace std;

namespace menuNamespace
{
     int menu(string* strArray, int size);
}
```

И наконец, исходный текст собственно функции меню. Сохраните его под именем MenuUtility.cpp.

```
//7.4 - MenuUtility.cpp - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
using namespace std;

namespace menuNamespace
{
     int menu(string* strArray, int size)
     {
          int userResponse;

          cout << "Варианты:"
          while(userResponse < 1 || userResponse > size)
          {
               for(int i = 0; i < size; i++)
               {
                    cout<< i + 1 << ")" << strArray[i] << endl;
               }
               cin>> userResponse;
          }
          return userResponse;
     }
}
```

Попробуйте поработать с этим кодом, поскольку в будущем мы будем многократно к нему обращаться.

Создание безымянных пространств имен

Каким образом можно гарантировать корректность идентификатора, присвоенного пространству имен? Создав пространство имен без имени. Звучит противоестественно? Поверьте, дела обстоят именно так. Если создать безымянное пространство имен, C++ автоматически даст ему уникальное имя. Синтаксис объявления безымянного пространства имен следующий (просто воспользуйтесь ключевым словом `namespace`):

```
namespace
{
     компоненты
}
```

В этом случае C++ позволяет обращаться к компонентам безымянного пространства имен, добавляя неявное объявление `using namespace` после объявления безымянного пространства имен. Так что на деле код выглядит примерно так:

```
namespace a
{
     компоненты
}
using namespace a;
```

Здесь *a* – уникальный идентификатор, о котором вы ничего не знаете и не можете узнать.

Пример работы с безымянным пространством имен:

```
#include <iostream>
using namespace std;

namespace
{
     void func(void) {cout << "::func" << endl; }
}

int main(void)
{
     ::func();
}
```

Вывод:

```
::func
```

И снова пространство имен std

Стандартное пространство имен охватывает всю стандартную библиотеку C++, включая библиотеку iostream и библиотеку работы со строками, которые мы активно использовали в программах. Директива `using namespace std` включает стандартное пространство имен в глобальное пространство имен.

Это очень полезная строка, поскольку она позволяет использовать код, не задумываясь об уточнении областей видимости и пространств имен. Возьмем для примера достаточно простую программу:

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
     string name;
     cout<< "Как ваше имя, милорд?" <<endl;
     cin>> name;
     cout<< "\nПриветствую вас, сэр " << name.c_str() << endl;
     return 0;
}
```

Чтобы создать такую же программу, не выполняя слияния стандартного и глобального пространств имен, мы должны будем воспользоваться оператором разрешения контекста. Сохранение функциональности этого простого примера обойдется нам в восемь таких операторов.

```
#include <iostream>
#include <string>

int main(void)
{
     std::string name;
     std::cout<< "Как ваше имя, милорд?" <<std::endl;
     std::cin>> name;
     std::cout<< "\nПриветствую вас, сэр " << name.c_str() << std::endl;
     return 0;
}
```

Мораль этого рассказа – пользуйтесь стандартными пространствами имен, они существенно облегчают жизнь.

Пишем игру «Пиратский город»

После долгих месяцев грабежа в Карибском море твой корабль «Кровавый ветер» вошел в доки Сент-Мари. Что ты сделаешь, впервые сойдя с корабля? Некоторые из вариантов звучат достаточно странно. Что ж, вот игра, которая тебе подойдет.

Изучая игру, обратите внимание, что каждый крупный раздел заключен в отдельное пространство имен. В по-настоящему хорошей игре каждый из разделов будет существенно длиннее. Разобравшись с этим кодом, подумайте, как его можно улучшить. Не забудьте включить файлы меню в свой проект. Счастливого грабежа!

Прежде всего, сохраните следующий код в заголовочном файле pirateTown.h.

```
//7.5 - "Пиратский город"  (pirateTown.h) - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <string>
#include "MenuUtility.h"
using namespace std;

//объявления функций
namespace street
{
     void menu(void);
}

namespace weaponShop
{
     void menu(void);
}

namespace wharf
{
     void menu(void);
}

namespace tavern
{
     void menu(void);
}
```

Теперь наберите и выполните следующий исходный текст (файл pirateTown.cpp).

```
//7.6 - "Пиратский город" (pirateTown.cpp) - Дирк Хенкеманс
//Premier Press
#include <iostream>
#include <string>
#include "MenuUtility.h"
#include "pirateTown.h"
using namespace std;

//включает функции меню в глобальное пространство имен
using menuNamespace::menu;

//этот код
//отвечает за все события на пристани
namespace wharf
{
     void menu()
```

```
     {
          string options[] =
          {"Прыгнуть в воду",
          "Взять гребную шлюпку и уплыть в закат.",
          "Взойти на борт \"Кровавого ветра\".",
          "Отправиться в город"};
          int userResponse = ::menu(options, 4);

          switch(userResponse)
          {
               case 1:
                    cout << "Ты прыгаешь в воду."
                         << " Внезапно ты слышишь"
                         << " смех, и тут же \n"
                         << "осознаешь, что забыл"
                         << " снять одежду."
                         << " Ты вылезаешь \n"
                         << "из воды, промокший насквозь. \n" ;
                         menu();
                    break;
               case 2:
                    cout << "Ты хватаешь небольшую красную лодчонку"
                         << " и уплываешь в закат."
                         << "Ах, слава." << endl;
                    break;
               case 3:
                    cout << "Ты возвращаешься на борт \"Кровавого ветра\""
                         << " и ждешь, когда вернутся твои друзья,"
                         << " вдоволь навеселившись. \n";
                    break;
               case 4:
                    street::menu();
                    break;
          }
     }
}

//обработка событий в таверне
namespace tavern
{
     void menu(void)
     {
          string options[] =
          {"Заказать выпивку",
          "Начать шумную драку.",
          "Выйти на улицу."};
          int userResponse = ::menu(options, 3);

          switch(userResponse)
          {
               case 1:
                    cout << "Ты заказал коктейль"
```

```
                    << " \"Лентяй\"\n";
                menu();
                break;
            case 2:
                cout << "Ты начал замечательную шумную драку.\n";
                menu();
                break;
            case 3:
                street::menu();
                break;
        }
    }

}
//обработка событий, происходящих на улице
namespace street
{
    void menu(void)
    {
        string options[] =
        {"Двинуться к причалу",
        "Войти в таверну.",
        "Войти в оружейный магазин.",
        "Начать драку."};
        int userResponse = ::menu(options, 4);

        switch(userResponse)
        {
            case 1:
                wharf::menu();
                break;
            case 2:
                tavern::menu();
                break;
            case 3:
                weaponShop::menu();
                break;
            case 4:
                cout<< "Ты начал отличную, полезную для здоровья"
                    << " потасовку прямо на улице. \n";
                street::menu();
                break;
        }
    }

}

//покупка оружия
namespace weaponShop
{
    void menu(void)
    {
        string options[] =
```

```
            {"Купить кинжал, инкрустированный драгоценными камнями, за 300.",
            "Купить великолепный кремневый мушкет за 300",
            "Купить стандартную английскую боевую саблю за 100",
            "Уйти из магазина."};
            int userResponse = ::menu(options, 4);

            switch(userResponse)
            {
                 case 1:
                      cout << "Ты купил кинжал"
                           << " и спрятал его в карман, уплатив \n"
                           << "костлявому мужчине"
                           << " за прилавком \n";
                      menu();
                      break;
                 case 2:
                      cout << "Заплатив за мушкет, "
                           << " ты сразу опробовал его. Он "
                           << "превосходно действует!!! \n";
                      menu();
                      break;
                 case 3:
                      cout << "Взмахнув саблей пару "
                           << "раз, для пробы, "
                           << " ты понимаешь, что она \n"
                           << "не стоит потраченного"
                           << " золота \n";
                      menu();
                      break;
                 case 4:
                      street::menu();
                      break;
            }
       }

}

//начало игры
int main( void )
{
       cout<< "Твой корабль, Кровавый ветер, вошел в доки Сент-Мари."
            << " Ты сошел с \n"
            << "корабля и стоишь на причале. \n \n";
       wharf::menu();

       return 0;
}
```

Резюме

В этой главе вы научились разделять контекст на меньшие, более управляемые части. Затем вы изучили стандартное пространство имен и

причины, по которым его следует использовать на постоянной основе. Кроме того, вы узнали, как получить доступ к пространству имен и как создать безымянное пространство. И хотя в небольших программах пользы от пространств имен немного, в крупных проектах они позволяют избежать ошибок, связанных с дублирующимися именами, и затрат времени на их исправление.

Задания

1. Объясните, зачем можно использовать пространства имен.
2. Какие преимущества имеет безымянное пространство имен перед обычным?
3. Назовите два способа организации прямого доступа к пространствам имен. Чем они различаются?
4. Каким образом для приведенного ниже кода можно вызвать функцию `breathFire()` из:
   a. глобального пространства имен?
   b. пространства имен dragon?
   c. из другого пространства имен?

```
namespace dragon
{
    void breathFire() {cout<< ¨Дракон дышит огнем \n¨; }
}
using dragon::breathFire();
```

8

Наследование

Предшествующие главы были посвящены основам объектно-ориентированного программирования (ООП) и переходным темам C++. В этой главе мы изучим очень важное понятие: наследование. *Наследование* – это создание одного класса на базе другого. Разобравшись с наследованием, вы выйдете на финишную прямую, которая ведет к свободному чтению кода на C++ и проектированию эффективных, хороших программ. В этой главе:

- Понятие наследования
- Доступ к компонентам базового класса
- Множественное наследование
- Снова о полиморфизме
- Виртуальные функции
- Абстрактные классы

Как работает наследование

В главе 5 «Боевые качества ООП» мы уже говорили, что классы обычно моделируют понятия или сущности. Но мы не сказали, что эти понятия и сущности имеют определенные взаимоотношения. Подобно прочим взаимоотношениям, эти определяют и проясняют логический порядок объектов мира.

При проектировании классов для своих программ не забывайте сфокусироваться на этих отношениях из соображений удобства и логики. В этой главе мы расскажем, как выражать их в коде.

Когда две вещи связаны, у них есть общие свойства. Например, и кошки и собаки являются животными, у них есть хвосты, мех, усы. Менеджер и рекламный агент получают зарплату, ходят на работу по расписанию, имеют номера социального страхования.

Чтобы выразить эту похожесть в C++, следует определить три класса – два для связанных сущностей и один для их общих свойств. Если брать в качестве примера животных, мы создадим классы `Animal`, `Dog` и `Cat`, чтобы выразить отношения между кошками и собаками. Затем эти классы необходимо связать при помощи наследования.

Но прежде необходимо изучить новый тип отношений, называемый *производной от*. В качестве примера опять же можно привести отношение между собакой и понятием животного. Переход от понятия животного к понятию собаки производится добавлением нескольких простых идей. Это отношение также известно как отношение типа *является*. Собака является животным.

В C++ создается *базовый класс* (называемый также *суперклассом* или *родительским классом*), который обычно моделирует общую идею или род сущности (в нашем примере базовым классом является класс `Animal`). Прочие классы являются производными базовых классов. *Производные классы* (*подклассы*, или *порожденные классы*) наследуют все компоненты базовых классов и другие свойства. Рис. 8.1 иллюстрирует изложенное.

Не переживайте, если что-то непонятно. Когда вы разберетесь с созданием кода наследования, полученная информация существенно прояснится.

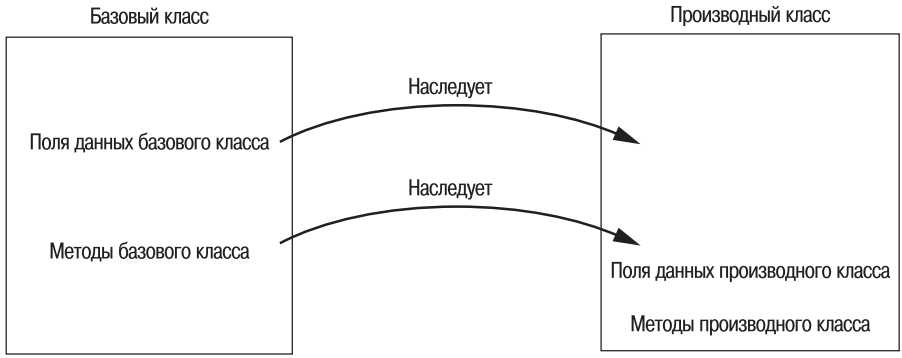


Рис. 8.1. *Производный класс наследует компонентную информацию от своего родительского, или базового класса*

Пишем код наследования

Класс с одним полем данных, такой как

```
class aClass
{
     public:
          int anInt;
};
```

можно использовать для порождения другого класса следующим образом:

```
class aDerivedClass : public aClass
{
     protected:
          float aFloat;
};
```

Класс aDerivedClass имеет два поля данных: объявленный aFloat, а также anInt, который порожденный класс унаследовал от класса aClass.

Общий синтаксис создания производных классов:

```
class имя_порожденного_класса : модификатор_доступа имя_базового_класса
```

Здесь *имя_порожденного_класса* — это имя производного класса, а *имя_базового_класса* — имя класса-предка. *модификатор_доступа* может иметь значение public, private или protected; пока что мы остановимся на public. Результаты применения различных модификаторов доступа мы обсудим позже в этой главе в разделе «Разграничение доступа к базовым классам».

Порожденный класс наследует все компоненты базового класса. Можно считать, что эти компоненты объявлены в порожденном классе (хотя это не совсем точно, как мы увидим в следующем разделе).

Наследование никак не влияет на базовый класс. Он остается обычным классом со своими собственными компонентами. Единственным классом, на который влияет наследование, является порождаемый класс. Поясним на таком примере:

```
class Base
{
     public:
          int base_int;
};

class Derived : public Base
{
     public:
          int derived_int;
};

int main ( void )
{
     Derived d;
     d.base_int = 5; // допустимо
     d.derived_int = 10; //допустимо
     Base b;
     b.base_int = 6; //допустимо
     b.derived_int = 12; //недопустимо - в классе Base только одно поле данных
     return 0;
}
```

Как видите, базовый класс Base ничего не наследует. Базовый класс содержит только одно поле данных, base_int, поэтому оператор присваивания значения 6 переменной b.derived_int является недопустимым – в b не существует поля данных с именем derived_int. С другой стороны, порожденный класс derived_int содержит *два* поля данных, компонент базового класса base_int и компонент, объявленный в самом порожденном классе, derived_int.

Наследование и доступ к членам класса

Правила доступа для производных классов очень просты, но новичку зачастую трудно в них разобраться (поскольку даже у профессионалов они иногда вызывают затруднения). В этом разделе мы постараемся максимально сократить путаницу и увеличить удовольствие от учебы (или, в крайнем случае, выполнить хотя бы одну из этих задач).

Прежде всего, вспомните, что как общие, так и частные компоненты принадлежат классу. Существует и третий вид компонентов – *protected*, защищенные. В этом разделе мы затронем применения и ограничения этого модификатора доступа.

Порожденный класс имеет доступ:

- К собственным компонентам
- Ко всем глобальным переменным
- Ко всем общим и защищенным компонентам родительского класса

Как видите, ничего сложного нет. Если возникают сомнения, просто обратитесь к этим правилам. Порожденный класс имеет почти такой же доступ к компонентам, как базовый, поскольку наследует большую часть его функциональности. Порожденный класс не имеет доступа только к частным компонентам базового класса.

Из предшествующего списка следует, что порожденный класс имеет доступ ко всем общим и защищенным компонентам базового класса, но не к частным компонентам. Вот пример:

```
class Base
{
     private:
          int private_int;
     protected:
          int getInt() { return private_int;}
};

class Derived : public Base
{
     protected:
          //ошибка - нет доступа к private_int
          int getInt() { return private_int; }
          int getInt() { return Base::getInt(); } // Так правильно
}
```

В этом примере класс Derived не может обращаться к полю private_int, поскольку оно является частным. Однако порожденный класс может воспользоваться защищенной компонентной функцией getInt(). Именно такой способ доступа (посредством общих и защищенных функций) к частным данным базового класса является правильным. Помните, что в объекте класса Derived все-таки существует поле данных private_int, просто оно недоступно напрямую.

В большинстве других случаев модификатор protected имеет такое же значение, как private. Например, если создать объект Derived, извне класса доступа к функции getInt() нет. В табл. 8.1 приведена информация по модификаторам доступа.

Таблица 8.1. Доступ к компонентам класса

Модификатор доступа	Внутри класса	Извне класса	В порожденном классе
private	Есть доступ	Нет доступа	Нет доступа
protected	Есть доступ	Нет доступа	Есть доступ
public	Есть доступ	Есть доступ	Есть доступ

В большинстве случаев для полей данных следует использовать модификатор доступа private.

Переопределение функций

Иногда в порожденном классе требуется несколько иная реализация метода, унаследованного от родительского класса. Такое требование может быть вполне обоснованным. Скажем, порожденному классу требуется более точная реализация либо необходимо расширение функциональности метода базового класса. Разумеется, в C++ существует способ сделать это, и называется он *переопределением*.

Чтобы переопределить функцию в порожденном классе, достаточно просто создать функцию с таким же заголовком. После этого при вызове функции будет выполняться реализация порожденного класса. Таким образом, переопределенная функция просто замещает соответствующую функцию базового класса. Пример:

```
//8.1 - Переопределение функций - Марк Ли - Premier Press
#include <iostream>
using namespace std;
class Base
{
     public:
          void display() { cout << "Базовый класс\n"; }
};

class Derived : public Base
{
```

```
    public:
        void display() { cout << "Порожденный класс\n"; }
};

int main( void )
{
    Derived d;
    d.display();
    Base b;
    b.display();
    return 0;
}
```

Вывод:

```
Порожденный класс
Базовый класс
```

В этом примере при вызове метода `display` объекта `d` используется реализация метода из класса `Derived`. Прежде всего, производится поиск метода в порожденном классе, и если метод не найден, поиск продолжается в родительском классе. Обратите внимание, что реализация функции в базовом классе не изменяется при переопределении в порожденном классе. Эта операция не затрагивает базовый класс.

Что еще умеет оператор разрешения контекста

Иногда после переопределения функции по-прежнему требуется доступ к реализации функции из базового класса. В таком случае следует воспользоваться оператором разрешения контекста (`::`) и явным образом указать, к компонентам какого класса происходит обращение. Синтаксис оператора разрешения контекста:

имя_класса`::`*компонент*

где *компонент* — имя компонента, а *имя_класса* — имя класса, в котором существует компонент. Приведем пример:

```
//8.2 - Применение оператора разрешения контекста - Марк Ли
//Premier Press
#include <iostream>
using namespace std;
class Base
{
    public:
        void display() { cout << "Здравствуй, мир.\n"; }
};

class Derived : public Base
{
    public:
        void display()
        {
            // выполняет функцию display() базового класса
```

```
            Base::display();
            cout << "Прощай, мир.";
        }
};

int main(void)
{
    Derived d;
    d.display();    // выполняет функцию display() порожденного класса
    return 0;
}
```

Вывод:

```
Здравствуй, мир.
Прощай, мир.
```

В этом примере в строке `Base::display()` происходит явный вызов метода `display()` базового класса. Помните, что для доступа к компонентам посредством оператора разрешения контекста требуется иметь к ним допуск. Этот оператор не позволит обращаться к структурам, которые обычно недоступны.

Конструкторы и деструкторы в порожденных классах

Конструкторы и деструкторы классов, вовлеченных в процесс наследования, предоставляют собой специальный случай, который мы рассмотрим отдельно.

Первое правило звучит так: если в базовом классе отсутствует конструктор или существует конструктор без аргументов, порожденному классу конструктор не требуется. По умолчанию используется пустой конструктор для базового класса. Пример:

```
class Base
{
    protected:
        int an_int;
};

class Base2
{
    Base2() { cout << "Привет"; }
    protected:
        int an_int;
};

class Derived : public Base
{
    //конструктор не требуется
};
```

```
class Derived2 : public Base2
{
     //конструктор обязателен
     public:
     Derived2() {}
};

int main( void )
{
     Derived2 d2;
     return 0;
}
```

Второе: если все конструкторы базового класса требуют наличия аргументов, порожденный класс обязан иметь конструктор, причем конструктор, вызывающий конструктор базового класса в списке инициализации. Пример:

```
class Base
{
     public:
          Base(int a) {}
};

class Derived : public Base
{
     public:

          Derived(int a, int b) : Base(a) {} //верно
          Derived(int a, int b) {Base(a)} //верно
          Derived(int a, int b) {} //недопустимо
};
```

Третье: конструктор порожденного класса не может инициализировать переменные, унаследованные от родительского класса, в списке инициализации. Следует воспользоваться конструктором базового класса либо инициализировать переменные внутри фигурных скобок ({ и }). Пример:

```
class Base
{
     public:
          Base (int a);
     protected:
          int an_int;
          int an_int2;
};
//конструктор базового класса:
Base::Base(int a) : an_int(a) {}

class Derived : public Base
{
     public:
          Derived(int a, int b);
```

```
    protected:
        int an_int3;
};
//конструктор класса Derived
//верно
Derived::Derived(int a, int b) : Base(a), an_int3(b) {}
//тоже верно
Derived::Derived(int a, int b) : Base(a) {an_int2 = b;}
//ошибка
Derived::Derived(int a, int b) : Base(a), an_int2(b) {}
```

В первом варианте конструктора класса `Derived` вызывается конструктор базового класса, после чего порожденный класс инициализирует собственные переменные. Именно такой вариант получил широкое распространение и является корректным способом инициализации класса.

Однако второй вариант тоже является верным. Компоненты базового класса могут быть инициализированы в теле конструктора, а не только в списке инициализации.

Порядок создания объектов следующий: сначала базовый класс, затем порожденный. Уничтожение (вызов деструкторов) происходит в обратном порядке. Порожденный класс уничтожается первым, затем базовый класс. Поясним на примере:

```
class Base
{
    public:
        Base() { cout << "Конструктор класса Base\n"; }
        ~Base() { cout << "Деструктор класса Base\n"; }
};

class Derived : public Base
{
    public:
        Derived() { cout << "Конструктор класса Derived\n"; }
        ~Derived() { cout << "Деструктор класса Derived\n"; }
};

int main ( void )
{
    Derived d;
    return 0;
}
```

Вывод:

```
Конструктор класса Base
Конструктор класса Derived
Деструктор класса Derived
Деструктор класса Base
```

Вот видите, все очень просто! Усвоив основы, вы не встретите сложностей на своем пути.

Усложняем наследование

Простое наследование, о котором мы говорили до сих пор, не особенно полезно, но является фундаментом для возведения небоскреба знаний. Понятия, о которых речь пойдет далее, предельно просты, и если у вас не возникло проблем в предшествующих разделах, то и здесь их быть не должно.

Во-первых, порожденный класс также может являться родительским классом для другого порожденного класса. Многоуровневое порождение позволяет создавать цепи наследования. *Цепь наследования* – это последовательность порожденных классов, каждый из которых является базовым для последующего. Идея заключается в том, что степень охвата понятий классами уменьшается по мере продвижения к концу цепи.

Следующий код иллюстрирует цепи наследования в C++:

```
class Vehicle
{};

class LandVehicle : public Vehicle
{};

class Car : public LandVehicle
{};

class HondaInsight : public Car
{};
```

Как и в случае простого наследования, конструкторы выполняются от первого базового класса к последнему классу цепи, деструкторы – в обратном порядке. Например, для приведенной цепи наследования порядок будет таким: конструктор `Vehicle`, конструктор `LandVehicle`, конструктор `Car`, конструктор `HondaInsight`; затем деструктор `HondaInsight`, деструктор `Car`, деструктор `LandVehicle`, деструктор `Vehicle`.

На рис. 8.2 приведена концептуальная схема цепи наследования `Vehicle`. Обратите внимание, что стрелки направлены к началу цепи. Они показывают направления доступа одних классов к другим. Базовый класс не имеет доступа к порожденному классу, но обратное верно.

При вызове метода класса компьютер производит поиск метода в указанном классе, затем в родительском классе, затем в базовом классе для родительского класса и так далее вверх по цепи, пока не будет найден метод.

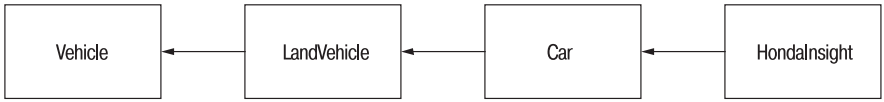


***Рис. 8.2.** Стрелки на схеме показывают направление организации доступа для классов*

Базовый класс может служить родителем нескольких порожденных классов. Таким образом, наследование позволяет создавать древовидные структуры. Эти структуры называются *иерархиями классов*. Примером крупной, развитой иерархии классов является библиотека MFC (Microsoft Foundation Classes), которую вам придется изучить, чтобы создавать сложные приложения для Windows.

Иерархия классов не является чем-то запредельно сложным. Главное, что следует помнить, – *потомки одного уровня* (то есть имеющие общего предка) являются независимыми; их компоненты ни при каких условиях не могут стать общими. Иерархия классов для транспортных средств приведена на рис. 8.3.

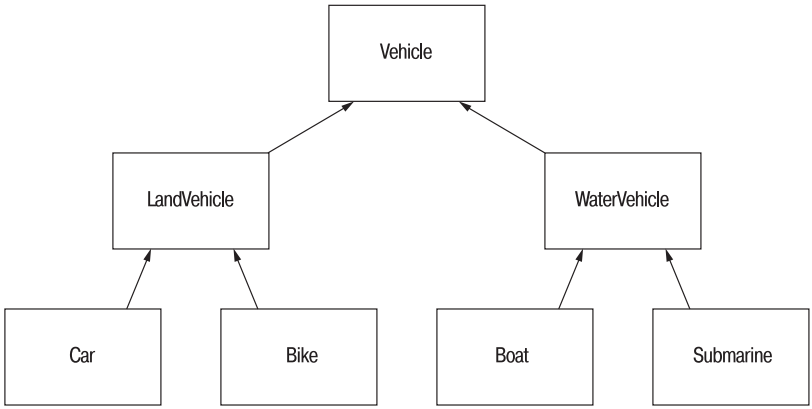


Рис. 8.3. *Пример проектирования иерархии классов транспортных средств*

Следующий код отражает эту иерархическую структуру на C++:

```
class Vehicle
{};

class LandVehicle : public Vehicle
{};

class WaterVehicle : public Vehicle
{};

class Car : public LandVehicle
{};

class Bike : public LandVehicle
{};

class Boat : public WaterVehicle
{};

class Submarine : public WaterVehicle
{};
```

Эту причудливую иерархию классов можно представить в виде ряда цепей наследования, как показано на рис. 8.4.

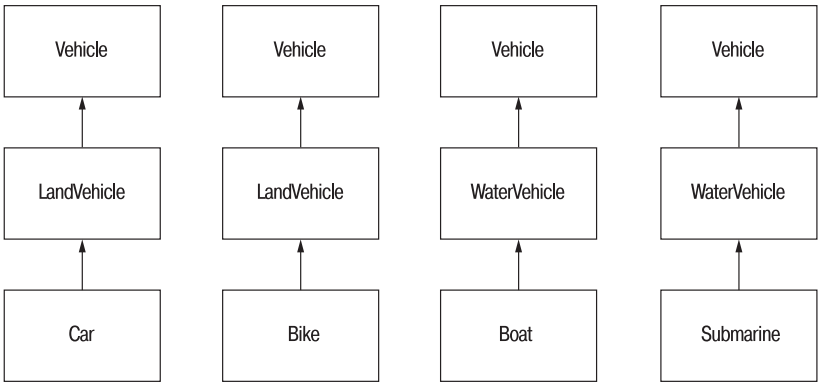


Рис. 8.4. *Иерархия состоит из многих цепей наследования*

Разграничение доступа к базовым классам

Помните, мы говорили, что модификатор доступа перед именем базового класса может принимать значения `public`, `private` и `protected`? Настало время проверить теорию практикой. Модификатор `public` означает, что все общие (public) и защищенные (protected) компоненты базового класса свободно доступны в порожденном классе и объектах порожденного класса.

Однако в случае модификаторов `private` и `protected` все выглядит совсем иначе. Модификатор `protected` делает все общие компоненты базового класса защищенными для класса-наследника. Помните, что базовый класс при этом не меняется. Изменяются только версии компонентов базового класса для класса-наследника.

Модификатор `private` делает все компоненты базового класса частными. Доступ к этим компонентам имеют только методы порожденного класса. В табл. 8.2 содержится сводка по модификаторам доступа и их влиянию на наследование.

Таблица 8.2. Модификаторы доступа и наследование

Модификатор доступа	Общие компоненты базового класса	Защищенные компоненты базового класса	Частные компоненты базового класса
`public`	Остаются общими	Остаются защищенными	Остаются частными
`protected`	Становятся защищенными	Остаются защищенными	Остаются частными
`private`	Становятся частными	Становятся частными	Остаются частными

Приведем пример применения модификаторов доступа в коде:

```
class Base
{
private:
     int x;
protected:
     int y;
public:
     int z;
};

class Derived1 : public Base
{
     void f()
     {
          x = 5; //ошибка - x частное поле base
          y = 10; //ok
          z = 15; //ok
};

class Derived2 : protected Base
{
     void f()
     {
          x = 5; //ошибка
          y = 10; //ok
          z = 15; //ok
};

class Derived3 : private Base
{
     void f()
     {
          x = 5; //ошибка
          y = 10; //ошибка
          z = 15; //ошибка
};

int main(void)
{
     Derived1 d1;
     Derived2 d2;
     Derived3 d3;
     //ошибка - нет доступа к частным компонентам извне класса
     d1.x = 4;
     //ошибка - нет доступа к защищенным компонентам извне класса
     d1.y = 8;     d1.z = 12; //ok
     d2.x = 3; //ошибка - частное поле данных
     d2.y = 6; //ошибка - защищенное поле данных
     d2.z = 9; //ошибка - защищенное поле данных
     d3.x = 2; //ошибка - частное поле данных
     d3.y = 4; //ошибка - частное поле данных
```

```
    d3.z = 6; //ошибка - частное поле данных
    return 0;
}
```

Представьте, что модификаторы доступа работают подобно переключателю яркости света. В случае public-наследования свет горит ярче всего, и не видно лишь предметы в самых темных углах комнаты. Наследование с модификатором protected устанавливает среднюю яркость света, так что к некоторым предметам приходится подходить, чтобы увидеть их. Наследование с модификатором private выключает свет, так что можно видеть лишь предметы с исключительными характеристиками отражения света – большинство вещей остаются невидимыми.

Множественное наследование

До сих пор в тексте главы речь шла только об одиночном наследовании. *Одиночное наследование* происходит, когда порожденный класс создается на основе единственного класса-родителя. Но существует и другой вид наследования – *множественное наследование*. При множественном наследовании порожденный класс создается на основе двух или более базовых классов. Даже для опытных программистов понятие множественного наследования может быть сложным, так что мы будем путешествовать по этому темному лесу очень осторожно.

Идея множественного наследования заключается в интеграции функциональности базовых классов в один подкласс. И как бы невероятно это не звучало, множественное наследование может применяться на практике.

Код для множественного наследования выглядит достаточно прозрачно. При объявлении порождаемого класса указывается не один базовый класс, а несколько. Пример:

```
class A
{
    int a;
};

class B
{
    int b;
};

class AB : public A, public B
{
    int ab;
};
```

В результате получается порожденный класс AB с тремя частными компонентами: a, b и ab. Отметим, что базовые классы в объявлении разде-

лены запятыми. Порожденный класс включает все компоненты всех базовых классов, плюс свои собственные.

Представьте, что необходимо создать приложение для работы с записями сотрудников компании, создающей игры. Один вид сотрудников может быть представлен классом `Programmer`, а другой – классом `Artist`. Но что если сотрудник выполняет задачи того и другого рода? Программист-художник может принадлежать как к классу `Programmer`, так и к классу `Artist`. Выбрав один из классов, мы потеряем информацию. Решение проблемы – использовать множественное наследование. Мы можем создать класс `Programmer_Artist`, порожденный от классов `Programmer` и `Artist`, что позволит точно отразить роли сотрудников.

Разрешение неоднозначностей

При множественном наследовании базовые классы могут иметь компоненты с одинаковыми именами, что создает неоднозначности, подлежащие разрешению. Взгляните на такой пример:

```
class A
{
     public:
          int x;
};

class B
{
     public:
          int x;
};

class AB : public A, public B
{};

int main( void )
{
     AB ab;
     ab.x; // о котором x идет речь?
     return 0;
}
```

Как видите, строка `ab.x` может относиться к `x` класса `A` и к `x` класса `B`.

Простейший способ избавиться от неоднозначности – воспользоваться оператором разрешения контекста (`::`). С его помощью можно явным образом сообщить компьютеру, о каком из классов идет речь:

```
class A
{
     public:
          int x;
};
```

```
class B
{
     public:
          int x;
};

class AB : public A, public B
{};

int main( void )
{
     AB ab;
     ab.A::x = 5; // использовать x класса A
     ab.B::x = 6; // использовать x класса B
     return 0;
}
```

Оператор разрешения контекста позволяет различать экземпляры поля x. Однако необходимость использовать его при каждом обращении к компонентам может быстро утомить автора программы.

Более правильное решение – избавиться от неоднозначностей в порожденном классе, чтобы пользователи класса вообще не знали об их существовании. Процесс разрешения неоднозначностей для всех классов выглядит по-разному, и следующий код является примером выполнения этой операции.

```
//8.3 - Разрешение неоднозначностей - Марк Ли - Premier Press
#include <iostream>
using namespace std;
class Hello
{
     public:
          void g() { cout << "Здравствуй, мир\n";}
};

class Goodbye
{
     public:
          void g() { cout << "Прощай, мир\n";}
};

class HelloGoodbye : public Hello, public Goodbye
{
     public:
          void g() //переопределение Hello::g() и Goodbye::g()
          {
               Hello::g();
               Goodbye::g();
          }
};

int main( void )
```

```
{
     HelloGoodbye hg;
     hg.g();
     return 0;
}
```

Вывод:

```
Здравствуй, мир
Прощай, мир
```

В этом примере порожденный класс переопределяет методы `g()` классов `Hello` и `Goodbye`. Эта техника может эффективно применяться для разрешения неоднозначностей. Теперь пользователи класса `HelloGoodbye` будут знать лишь об одной версии функции.

Дубликаты базовых классов

Множественное наследование допускает существование более чем одной копии базового класса. К примеру, предположим, что классы `Hello` и `Goodbye` наследуют класс `Display`, реализующий функцию печати строк. Класс `Display` может выглядеть следующим образом:

```
class Display
{
     protected:
          void output(string s) { cout << s.c_str();}
};
```

А классы `Hello` и `Goodbye` так:

```
class Hello : public Display
{
     protected:
          void g() { output("Здравствуй, мир.\n"); }
};

class Goodbye : public Display
{
     protected:
          void g() { output("Прощай, мир.\n");}
};
```

И наконец, класс `HelloGoodbye`:

```
class HelloGoodbye : public Hello, public Goodbye
{
     public:
          void g()
          {
               Hello::g();
               Goodbye::g();
          }
};
```

Пока что неплохо. Следует только не забыть, что класс HelloGoodbye содержит две различных копии класса Display. Чтобы воспользоваться компонентами класса Display, следует указать конкретную копию этого класса. Диаграмма структуры классов приведена на рис. 8.5.

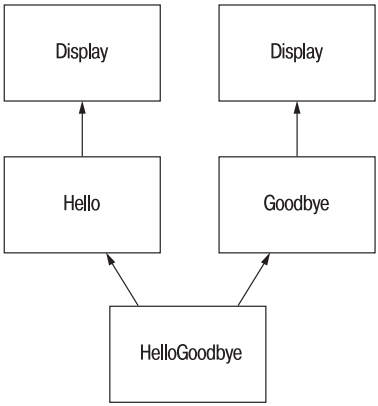


Рис. 8.5. *Вот так выглядит класс HelloGoodbye с дубликатами базового класса*

Для доступа к компонентам конкретной копии класса Display потребуется дважды воспользоваться оператором разрешения контекста: один раз, чтобы указать принадлежность класса Display (Hello или Goodbye), и еще раз, чтобы указать компонент класса. Пример:

```
class HelloGoodbye : public Hello, public Goodbye
{
     public:
          void g() { Hello::Display::output("Привет.\n");}
};
```

Очевидно, такое применение оператора разрешения контекста может быстро стать утомительным. Множественное наследование может быть опасной областью для программистов, но в умелых руках становится важным инструментом.

Некоторые специалисты полагают, что множественное наследование не следует использовать вообще, и неопытным программистам мы рекомендуем принять это мнение к сведению.

Доступ к объектам иерархии

Научившись создавать порожденные классы и подробно пройдясь по внутренним правилам и опасностям наследования, мы готовы рассмотреть вторую сторону медали. Необходимо знать, как работать с объектами, вовлеченными в наследование, и на какие моменты обращать при этом внимание. В следующих разделах содержатся важные понятия и новые идеи, так что будьте настороже. Вам предстоит на-

учиться работать с иерархиями классов и создавать истинный полиморфизм.

И снова полиморфизм

Одной из важнейших характеристик порождаемых классов является их способность в определенных ситуациях действовать так, как действовал бы любой из базовых классов. Эта способность, получившая название *полиморфизма*, является одной из главных составляющих ООП, позволяя использовать каждый объект в различных контекстах.

Вы часто будете использовать полиморфизм при работе с указателями на объекты базовых классов. В следующем разделе вы узнаете, как и когда проявляется подобный полиморфизм.

Указатели на базовые классы

Одним из наиболее актуальных примеров является проявление полиморфизма при работе с указателями на объекты из иерархии классов. Указатель на базовый класс может хранить адрес производного объекта. Пример:

```
Base* b;
Derived d;
b = &d; //порядок
```

Эта способность к хранению двух видов адресов является важной составляющей наследования и делает C++ невероятно мощным языком. В частности, она может применяться в массивах. Массив указателей на объект `Vehicle` позволяет хранить любые типы объектов, представляющих транспортные средства. В ином случае понадобилось бы создавать отдельный массив для каждого типа транспортных средств, что вряд ли можно назвать интересным занятием.

Полиморфизм имеет некоторые минусы. К примеру, если сохранить объект `Car` в массиве указателей на `Vehicle`, вернуться к типу `Car` становится проблематично. Иными словами, сложно определить, о каком типе транспортного средства идет речь в случае конкретного элемента массива указателей на `Vehicle`. Пример:

```
void f(Vehicle* v)
{
     //v может быть объектом Car, Boat, Bike и т. д.
}
```

Вторая проблема – компьютер предполагает, что объект, сохраненный посредством указателя на базовый класс, является объектом базового класса. Если в классе-наследнике функция переопределена, указатель не позволит ее использовать. Поясним на примере:

```
class Base
{
```

```
    public:
        void Display() { cout << "Класс Base, функция Display";}
};

class Derived : public Base
{
    public:
        void Display() { cout << "Класс Derived, функция Display";}
};

int main( void )
{
    Derived d;
    Base* b = &d;
    b->Display();
    return 0;
}
```

Вывод:

```
Класс Base, функция Display
```

Как видите, компьютер не может определить, является ли объект, хранимый в `b`, объектом порожденного класса или объектом базового класса. По этой причине компьютер предполагает, что хранимый объект имеет тип `Base`.

Эти правила могут показаться сложными. Если говорить проще, объекты порожденных классов могут храниться в качестве объектов базовых классов, но в этом случае они теряют опознавательные знаки (и в дальнейшем считаются объектами базовых классов).

Виртуальные функции

По счастью, существует решение проблемы недоступности переопределенных функций. Оно связано с использованием *виртуальных функций*. Виртуальные функции позволяют компьютеру определять, какую функцию следует вызвать, даже если неизвестен тип объекта, о котором идет речь.

Для создания виртуальных функций служит ключевое слово `virtual`. Общий синтаксис этой функции:

```
virtual тип_возвращаемого_значения имя_функции(список_аргументов);
```

Заголовок функции следует предварять ключевым словом `virtual` только в момент объявления. Если объявление и определение функции разнесены в коде, `virtual` предшествует только объявлению. Если функция определена в точке объявления, ключевое слово предшествует определению.

Ключевое слово `virtual` достаточно использовать только для той версии функции, которая принадлежит базовому классу. Создание виртуальной функции предписывает компьютеру выполнять поиск кор-

ректной версии функции при вызове. Взгляните на следующий пример:

```
//8.4 - Виртуальные функции - Марк Ли - Premier Press
#include <iostream>
using namespace std;
class Base
{
     public:
          void Display() { cout << "Класс Base, функция Display\n";}
};

class Derived : public Base
{
     public:
          void Display() { cout << "Класс Derived, функция  Display\n";}
};

class Base2
{
     public:
          virtual void Display() { cout << "Класс Base2, функция Display\n";}
          virtual void f(); // допустимая конструкция
};

void Base2::f() // здесь слово virtual не требуется
{}

class Derived2 : public Base2
{
     public:
          void Display() { cout << "Класс Derived2, функция Display\n";}
};

int main ( void )
{
     Derived d;
     Base* b = &d;
     b->Display();

     Derived2 d2;
     Base2* b2 = &d2;
     b2->Display();
     return 0;
}
```

Вывод:

```
Класс Base, функция Display
Класс Derived2, функция Display
```

Как можно видеть по результатам работы программы, применение виртуальной функции решило проблему. В классе `Base` использовалась обычная функция `Display()`, что привело к вызову неправильной версии функции. Функция `Display()` класса `Base2` является виртуальной, поэтому вызов `b2->Display()` приводит к желаемым результатам.

Виртуальные функции являются важной частью ООП C++. Без виртуальных функций полиморфизм в C++ стал бы не столь эффективным средством (или вообще невозможным).

Абстрактные классы

Некоторые из классов спроектированной иерархии могут не иметь смысла в качестве объектов. Скажем, несмотря на полезность класса `Vehicle` в качестве родителя, определяющего общие свойства наследников, создание объекта типа `Vehicle` бессмысленно.

Другой пример – иерархия классов геометрических форм, произрастающая из общего класса `Shape`. Создание объекта `Shape` не имеет смысла. Эта геометрическая форма не имеет формы. Такие классы представляют абстрактные понятия. Напротив, класс `Car` представляет конкретный объект. Еще одна возможность связана с созданием класса `Comparable`, реализующего базовый интерфейс, необходимый для сравнения одного объекта с другим. Наследуя классы от этого базового класса, мы получаем возможность сравнивать их. Однако создание объекта класса `Comparable` особого смысла не имеет.

В C++ существует способ представления абстрактных концепций, не позволяющий по случайности принять их за нормальные объекты. *Абстрактный класс* – это класс, содержащий чистую виртуальную функцию. (Создание объекта абстрактного класса невозможно.)

Чистая виртуальная функция – это функция, которая не реализована и используется только для создания абстрактного класса (то есть в базовом классе нет определения этой функции). Синтаксис чистой виртуальной функции дополняется конструкцией `= 0`:

```
virtual тип_возвращаемого_значения имя_функции(список_аргументов) = 0;
```

Помните, что не следует создавать реализации чистых виртуальных функций. Это приведет к ошибкам.

Если в числе компонентов класса есть чистая виртуальная функция (либо класс не переопределяет чистую виртуальную функцию, унаследованную от базового класса), класс является абстрактным классом. В C++ запрещено создавать объекты абстрактных классов. Чистая виртуальная функция гарантирует, что автор программы не попытается по ошибке создать объект базового класса. Таким образом, базовый класс существует только для поддержания структуры классов-наследников. Пример:

```
class Abstract
{
     public:
          virtual void draw() = 0;
};

int main(void)
{
```

```
    Abstract a; //недопустимо - возникает ошибка
    return 0;
}
```

Если в класс входит абстрактный базовый класс, можно переопределить все чистые виртуальные функции, чтобы избавиться от качества абстракции. Пример:

```
class Abstract
{
    public:
        virtual void draw() = 0;
};
class Derived : public Abstract
{
    public:
        //переопределение Abstract::draw()
        void draw() { cout << "Здравствуй, мир."; }
};

int main( void )
{
    Derived d; // допустимо
    Abstract a; // недопустимо
    return 0;
}
```

Если порожденный класс не переопределяет все чистые виртуальные функции родителя, он наследует их и становится абстрактным классом. Пример:

```
class Abstract
{
    virtual int f() = 0;
    virtual float g(float) = 0;
};

class Derived : class Abstract
{
    int f(); //переопределение чистой виртуальной функции родителя
    //g() не переопределяется
};

int main (void)
{
    Abstract a; //недопустимо - абстрактный класс
    Derived d; // недопустимо - также абстрактный класс
}
```

Пишем игру «Лорд-Дракон»

Путешествуя по великому королевству, ты повстречал высокопоставленных вестников. «О благородный рыцарь, – молвили они, – Ты дол-

жен нам помочь!» И прежде чем ты успел объяснить, что не имеешь к рыцарям никакого отношения, рассказали следующее: «Ужасный дракон похитил нашу принцессу! Ты должен спасти ее. Мы в отчаянии.»

Ты не можешь отказать в помощи, а потому оказываешься втянут в спасательную операцию и твой путь лежит к логову дракона. Сможешь ли ты победить его? Скомпилируй программу и узнай. Обратите внимание, что программа состоит из нескольких файлов.

Прежде всего, следует создать и сохранить файлы Dragon.cpp, RedDragon.cpp, BlueDragon.cpp и BlackDragon.cpp в каталоге проекта. Затем необходимо заменить .cpp-файл, созданный CodeWarrior, файлом DragonLord.cpp (удалите старый файл из проекта и добавьте DragonLord.cpp). Затем включите все файлы в проект. Наконец, скомпилируйте и выполните проект, как любую другую программу.

```
//8.5 - Лорд-Дракон - Марк Ли - Premier Press
// Dragon.cpp
#include <string>
#include <ctime>
#include <cstdlib>

#define MAX(a,b) a>b? a:b

using namespace std;
class Dragon
{
private:
     int speed;
     string name;
     int hitPoints;
     int armour;
     int treasure;
     int clawDamage;
     int size;
protected:
     Dragon(int theSize);
     int getArmour() {return armour;}
     int& getHitPoints() {return hitPoints;}
     int getClawDamage() { return clawDamage;}
     int getSize() { return size;}
     virtual int attack(int targetArmour, int specialDamage);
public:
     virtual int attack(int targetArmour) = 0;
     virtual void defend(int damage) = 0;
     int getTreasure() {return treasure;}
     virtual string getName() {return name;}
     int getSpeed() {return speed;}
     bool isAlive() { return hitPoints >0;}
};
```

```
Dragon::Dragon(int theSize) :
     size(theSize)
{
     if (size < 1 || size > 4)
          size = 3;
     clawDamage = 2 * size;
     speed = 3 * size;
     hitPoints = 4 * size;
     armour = size;
     treasure = 1000 * size;
     srand(time(0));
}

int Dragon::attack(int targetArmour, int specialDamage)
{
     int useSpecial = rand() % 2; // 0 или 1
     int damage;
     if (useSpecial)
          damage = specialDamage;
     else damage = getClawDamage();
     return MAX(damage - targetArmour,0);
}
//RedDragon.cpp
class RedDragon : public Dragon
{
private:
     int fireDamage;

public:
     RedDragon(int theSize);
     int attack(int targetArmour);
     void defend(int damage);
     string getName() {return "Красный дракон";}
};

RedDragon::RedDragon(int theSize) :
     Dragon(theSize)
{
     fireDamage = 4 * getSize();
}

int RedDragon::attack(int targetArmour)
{
     return Dragon::attack(targetArmour, fireDamage);
}

void RedDragon::defend(int damage)
{
     getHitPoints() -= (damage - getArmour())/3;
}
//BlueDragon.cpp
class BlueDragon : public Dragon
{
```

```
private:
     int iceDamage;

public:
     BlueDragon(int theSize);
     int attack(int targetArmour);
     void defend(int damage);
     string getName() {return "Синий дракон";}
};

BlueDragon::BlueDragon(int theSize) :
     Dragon(theSize)
{
     iceDamage = 3 * getSize();
}

int BlueDragon::attack(int targetArmour)
{
     return Dragon::attack(targetArmour, iceDamage);
}

void BlueDragon::defend(int damage)
{
     getHitPoints() -= (damage - getArmour())/2;
}
//BlackDragon.cpp
class BlackDragon : public Dragon
{
private:
     int poisonDamage;

public:
     BlackDragon(int theSize);
     int attack(int targetArmour);
     void defend(int damage);
     string getName() {return "Черный дракон";}
};

BlackDragon::BlackDragon(int theSize) :
     Dragon(theSize)
{
     poisonDamage = getSize();
}

int BlackDragon::attack(int targetArmour)
{
     return Dragon::attack(targetArmour, poisonDamage);
}

void BlackDragon::defend(int damage)
{
     getHitPoints() -= damage - getArmour();
}
//DragonLord.cpp
```

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "Dragon.cpp"
#include "RedDragon.cpp"
#include "BlueDragon.cpp"
#include "BlackDragon.cpp"

using namespace std;

int menuChoice();

int main (void)
{
     srand(time(0));
     Dragon* dragons[3];
     int hp = 15;
     int armour = 2;
     int tempArmour;
     int tempAttack;
     dragons[0] = new RedDragon(rand()%4+1);
     dragons[1] = new BlackDragon(rand()%4+1);
     dragons[2] = new BlueDragon(rand()%4+1);
     Dragon* d = dragons[rand()%3];
     cout << "Добро пожаловать, благородный рыцарь.\n"
          << "Ты должен спасти принцессу."
          << " Ее похитил "
          << d->getName() << ".\n"
          << "Ты должен победить дракона.\n";
     cout << "Твоя сила: " << hp << endl;
     while (d->isAlive() && hp>0)
     {
          int choice = menuChoice();
          if (choice == 3) goto RUN;
          else if (choice == 1)
          {
               tempAttack = rand()%16+5;
               tempArmour = armour;
          }
          else {
               tempAttack = rand()%11;
               tempArmour = armour + 4;
          }
          hp -= d->attack(armour);
          d->defend(rand()%16-5);
          cout << "\nТы наносишь мощный удар и ущерб "
               << tempAttack << " damage.\n";
          cout << "Твоя сила: " << hp;
     }
     if (d->isAlive())
          cout << "\nТы побежден "
               << " могущественным драконом.\n";
```

```
        else
            cout << "\n\nТы убил дракона!"
                 << " Поздравляем.\n"
                 << "Принцесса спасена.\n";
        return 0;
        RUN:
        cout <<"\nТы струсил и убежал.\n";
        return 0;
}

int menuChoice()
{
        int choice;
        do {
            cout << endl
                 << "[1]Атаковать\n"
                 << "[2]Обороняться\n"
                 << "[3]Бежать\n";
            cin >> choice;
        } while (choice < 1 && choice > 3);
        return choice;
}
```

Резюме

В этой главе на вас обрушился ураган новых понятий. Вы узнали об основах наследования, модификаторе доступа `protected`, множественном наследовании, виртуальных функциях и абстрактных классах. Понимание и умение воспользоваться информацией из этой главы является решающим на пути становления профессионального программиста на языке C++. Если вам случится работать с другими языками программирования, вы обнаружите, что понятия из этой главы применимы не только к C++.

Задания

1. Создайте иерархию классов оружия, в которой по меньшей мере четыре класса наследуются от базового. Какие поля данных должен содержать каждый из классов? Какие методы?
2. Приведите пример ситуации, когда может оказаться полезным множественное наследование. Легче ли решить задачу при помощи одиночного наследования?
3. Когда следует использовать наследование по моделям `protected` и `private`?
4. Спроектируйте и реализуйте абстрактный класс `Shape`. Что следует включить в класс? Что следует оставить для реализации в порожденных классах?

9

Шаблоны

Если вы читали книгу с самого начала, то находитесь на верном пути к вершинам мастерства. Вы постигли основы программирования на C++ и готовы к изучению более сложных концепций. В этой главе мы расскажем о шаблонах, и эта тема послужит трамплином для изучения стандартной библиотеки C++. Вы постигнете некоторые из важнейших ее составляющих. Для тех, кто планирует создавать библиотеки, которыми будут пользоваться другие программисты, эта глава является обязательной. Темы этой главы:

- Создание шаблонов классов
- Создание шаблонов функций
- Прозрачное использование шаблонов
- Работа со стандартной библиотекой
- Работа со строками
- Работа с векторами

Создание шаблонов

Вообразите, что вы – знаменитый оружейник, прославленный своим великолепным оружием по всей округе. Явившись лично, король заказывает лук, который позволял бы использовать любой вид стрел. По глупости вы соглашаетесь изготовить такой лук, не осознавая, что задача невыполнима. Такой лук невозможен. Но, к несчастью, сдать работу надо уже завтра. Когда-то, изучая древние руководства по оружейной науке, вы наткнулись на волшебное заклинание со странным именем – *шаблон*. «Произнеси это заклинание над оружием, – гласила книга, – и оно станет абсолютно универсальным, сможет действовать с любым видом боеприпасов.» Вот оно, спасение! Теперь вы запросто закончите лук к завтрашнему утру.

Шаблоны C++ позволяют обобщать класс или функцию, избавляясь от специализации на конкретном типе данных, и точно так же, как оружейник из (только что выдуманной) байки создал универсальный лук, создавать при помощи магии шаблонов функции и классы, работающие со многими типами данных.

К примеру, каждый вариант функции сложения, которую мы создавали в главе 4 «Пишем функции», работает только с конкретным типом данных (с двумя целыми числам, с двумя с плавающей точкой и т. д.). Шаблоны позволяют создать один вариант, который будет работать для всех типов данных C++ (и даже пользовательских типов данных, таких как классы). Звучит заманчиво? Мы так и думали.

Теория гласит, что при использовании шаблонов над данными выполняются одинаковые операции вне зависимости от типа данных. Как следствие, нет смысла заниматься реализацией кода повторно для различных типов. Шаблоны позволяют обобщать операции и предотвращать повторное программирование.

А теперь мы поделимся с читателями основами программирования с применением шаблонов.

Создание шаблонов классов

Шаблон класса – это класс, в котором шаблон используется в целях обобщения. Надо думать, эта фраза не послужит для вас источником просветления?

Прежде всего, взгляните на класс `Storage`, который позволяет хранить массив целых чисел (и не использует шаблоны):

```
class Storage
{
     int* data;
     int numElements;
     Storage();
     Storage(int* array, int num) :
          data(array), numElements(num) {}
     Storage(int num) {data = new int[num];}
     void AddElement(int newElement);
     int ElementAt(int location);
     int RemoveElementAt(int location);
     ~Storage();
};
```

Этот класс хранит массив целых чисел и содержит методы, которые позволяют добавлять, просматривать и удалять элементы. Но что делать, если необходимо хранить массив чисел с плавающей точкой или символов? Придется скопировать приведенный код, а затем заменить целые на числа с плавающей точкой или на символы, и это будет не очень эффективно. Либо можно каким-то образом передавать инфор-

мацию о хранимом типе данных в качестве параметра. Тогда, скорее всего, возникнет оператор switch для обработки различных вариантов. Код быстро станет сложным и плохо управляемым.

Лучшее решение этой задачи – шаблоны. Шаблонная версия класса Storage способна хранить любой тип данных. Кроме того, шаблоны являются простым способом передавать тип данных в качестве параметра, не усложняя код методов. Вот как выглядит класс Storage в случае использования шаблона:

```
template<class T> class Storage
{
     T* data;
     int numElements;
     Storage();
     Storage(T* array, int num) :
          data(array), numElements(num) {}
     Storage(int num);
     void AddElement(T newElement);
     T ElementAt(int location);
     T RemoveElementAt(int location);
     ~Storage();
};
```

Как видите, код существенно отличается. Сейчас он, возможно, смотрится несколько загадочно, но вы вскоре привыкнете. Здесь только два нововведения: ключевые слова int (не все, отметим) замещены буквой T, а непосредственно перед объявлением класса добавилась конструкция template<class T>.

Назначение буквы T объяснить очень просто. Она является заполнителем и обозначает произвольный тип данных, который может понадобиться хранить пользователю класса. Таким образом, если пользователь предпочтет хранить целые числа, все вхождения T будут заменены типом int. При этом отдельные вхождения int никак не связаны с хранением данных, а потому по-прежнему присутствуют в коде шаблона класса. В частности, нумерация элементов всегда основана на целых числах.

Конструкция template<class T> сообщает компьютеру, что следом объявляется класс-шаблон, а в качестве заполнителя используется идентификатор T. При этом, несмотря на ключевое слово class, T может являться любым типом данных, не обязательно объектным.

Шаблон можно также считать списком аргументов. В предыдущем примере был только один аргумент, class T. При использовании класса тип хранимых данных передается почти как аргумент. T заменяется указанным типом данных. Как видите, конструкция class T во многом похожа на объявление аргумента. Ключевое слово class используется здесь в значении «тип данных». По желанию можно вместо class ис-

пользовать ключевое слово typename (которое в данном контексте имеет то же значение).

После начального объявления T можно использовать в пределах класса аналогично любому другому имени типа (такому как int или float). То есть объявлять новые переменные, создавать массивы и указатели и применять все те конструкции, в которые входит имя типа.

При создании объекта класса-шаблона следует указать тип хранимых данных. В следующем примере мы намереваемся хранить целые числа:

```
Storage<int> intStore;
```

Как видите, указание типа данных для шаблона очень похоже на передачу аргумента. В момент создания intStore все вхождения T в классе заменяются типом int. Storage<int> можно считать равноценной любому другому имени класса. Таким же образом можно хранить и другие типы данных:

```
Storage<float> floatStore; //хранит значения float
Storage<string> stringStore; //хранит строки
Storage<Shape*> shapeStore; //хранит указатели на объекты Shape
```

Изучите синтаксис создания шаблона класса:

```
template<class ИмяТипа> class ИмяКласса
{
     ОбъявлениеКласса
};
```

Здесь *ИмяТипа* – произвольный заполнитель для типа данных, *ИмяКласса* – имя класса, а *ОбъявлениеКласса* соответственно объявление класса (стандартное, мы изучали его в главе 5 «Боевые качества ООП»). Помните, что *ИмяТипа* может быть представлено любым допустимым идентификатором, хотя обычно используется одна прописная буква, скажем, T или C.

Если метод реализуется вне шаблона класса, для него следует использовать синтаксис шаблонов, приведенный ниже:

```
template<class параметр> типВозвращаемогоЗначения
     ИмяКласса<параметр>::имяМетода(аргументы)
{
     реализацияМетода
}
```

В данном случае *параметр* – это имя параметра (например, T), а *реализацияМетода* – реализация метода.

В реализации конструктора имя класса используется дважды (имя класса совпадает с именем метода), и логично предположить, что конструкция <*параметр*> должна следовать за каждым вхождением имени класса. Это будет несколько избыточно, поэтому для имени метода не

является обязательным. Синтаксис для конструктора, реализуемого вне шаблона класса:

```
template<параметр>
     ИмяКласса<параметр>::ИмяКласса<параметр>(аргументы)
{
     реализацияКонструктора
}
```

Можно воспользоваться и более распространенным вариантом синтаксиса:

```
template<параметр> ИмяКласса<параметр>::ИмяКласса(аргументы)
{
     реализацияКонструктора
}
```

Оба варианта абсолютно равноценны. Автор класса может руководствоваться собственными предпочтениями. Первый вариант ближе к формальным определениям, второй чаще применяется.

При создании класса-шаблона мы рекомендуем в процессе проектирования работать с конкретным типом данных. После устранения всех ошибок и доработки класса его можно обобщить в шаблоне. Этот подход значительно упрощает программирование.

Работа с параметрами шаблона

Объявление в угловых скобках (например, `<class T>`) является *параметром шаблона*. Как вы видели, параметр шаблона ведет себя почти так же, как аргумент функции. Параметров, как и аргументов функции, может быть несколько. А говоря точнее, сколько угодно.

Множественные параметры шаблона могут быть полезны, например, при создании ассоциативного массива. *Ассоциативный массив* схож с обычным, но каждый его элемент состоит из двух объектов. Один из этих двух объектов может быть строкой, а второй – целым числом, отражающим число вхождений строки в блок текста. Ассоциативные массивы можно считать сдвоенными массивами с равным числом элементов. Вот так может выглядеть класс ассоциативного массива:

```
template<class T, class C> class AssociativeArray;
AssociativeArray<string, int> wordFrequency;
```

Кроме того, параметр шаблона может иметь любой тип, не только общий, а, например, целочисленный или с плавающей точкой. Пример использования конкретного типа параметра шаблона:

```
template<class T, int i> class Array
{
     T data[i];
     //...
};
```

Использовать класс можно следующим образом:

```
Array<int, 10> intArray;
```

Приведенный фрагмент кода создает массив из десяти целых чисел. Обратите внимание, что размер массива передается в качестве одного из параметров шаблона. Не торопитесь удивляться. Динамические массивы не могут создаваться на основе аргументов функции, поскольку размер массива должен определяться постоянным значением (за исключением случаев, когда память выделяется динамически). Параметр шаблона, в отличие от аргумента функции, должен быть представлен константой. Параметр шаблона может быть константой или указателем. Кроме того, в пределах класса параметр шаблона также считается константой (его значения нельзя изменить). Проиллюстрируем сказанное примером:

```
int i = 10;
//ошибка - параметр шаблона должен быть константой
Array<int, i> intArray;
Array<int, 10> intArray; //порядок - константа
```

Компилятор должен знать размеры всех создаваемых массивов до того, как программа будет выполнена. Выражение, состоящее из констант, такое как `20 - 10`, может быть вычислено компилятором, а потому также может использоваться в качестве значения параметра шаблона.

При этом компилятор способен обнаружить только ошибки синтаксиса в момент компиляции шаблона. В частности, проверить, что код шаблона будет работать для конкретного типа данных, компилятор может только тогда, когда шаблон используется для этого типа данных. Момент времени, когда шаблон применяется для конкретного типа, называется *точкой конкретизации*.

Представьте, что при проектировании шаблона класса вы ошибочно предположили, что он будет использоваться только для типа `string`, и задействовали функцию `c_str()`. В конечном итоге это может привести к проблемам. Если объект будет создан на базе типа `string`, это не вызовет ошибок. Однако ошибки могут появиться при использовании других типов. Подобные недосмотры приводят к осложнениям не всегда, но довольно часто, так что будьте осторожны. При разработке шаблона класса, который работает с ограниченным диапазоном типов, не забудьте донести эту информацию до пользователей класса.

Создание шаблонов функций

Функции интегрируются с шаблонами – так же, как и классы. Хорошим примером будет функция сложения из главы 4. Конечно, функция сложения примитивна (она всего лишь складывает два числа), но принципы работы подобных функций можно распространить и на решение более сложных задач.

Шаблоны функций также создаются при помощи ключевых слов `template` и параметров. Функции-шаблоны похожи на перегруженные функции, но они позволяют создать единственную функцию для обработки всех вариантов, вместо того, чтобы создавать отдельную функцию для каждого варианта. Вот так может выглядеть шаблон функции сложения:

```
template<class T> T add (T a, T b)
{
     return a + b;
}
```

К сожалению, эта функция сложения не будет универсальной. Она не позволяет использовать два аргумента различного типа. В разделе «Перегрузка шаблонов функций» ниже по тексту главы вы узнаете о способах передачи произвольных аргументов с целью решения поставленной задачи.

Параметры шаблона могут использоваться в любой точке определения функции наравне с другими типами данных. Пример:

```
template<class T> void swap(T* a, T* b)
{
     T temp = *a;
     *a = *b;
     *b = temp;
}
```

Приведенная функция обменивает значения двух указателей. Ее можно использовать так:

```
int x = 5;
int y = 6;
swap<int>(&x, &y);
```

После вызова `x` содержит значение `6`, а `y` содержит значение `5`. В библиотеке `<iostream>` присутствует стандартная библиотечная версия этой функции, которая называется `swap`. Она принимает пару аргументов одного типа (причем неважно какого, поскольку функция реализована с помощью шаблона).

Разрешение аргументов

Шаблоны функций отличаются от шаблонов классов. Параметр шаблона не всегда должен указываться при вызове функции-шаблона. В некоторых случаях компьютер может самостоятельно определить, какими должны быть параметры, на основе полученных аргументов. Пример:

```
add(5,3);
```

Не обманитесь, отважные читатели, это вовсе не функция сложения из главы 4. В данном случае вызывается вариант с шаблоном. Поскольку оба аргумента являются целыми числами, компьютер способен сделать вывод, что параметр `T` должен иметь значение `int`. Вы стоите в начале пути настоящего программирования; но пока необязательно даже знать, что у функции `add` есть шаблон. Ее можно вызывать как всякую другую функцию.

Если аргументы, переданные функции, не столь очевидны (например, один целочисленный и один с плавающей точкой), компилятор сообщает об ошибке. Умственные способности компьютеров ограничены, так что они неспособны решить, следует ли преобразовать целое в число с плавающей точкой или же, наоборот, число с плавающей точкой в целое. Опуская параметры при вызове функций-шаблонов, позаботьтесь о том, чтобы аргументы четко определяли, какими должны быть параметры.

Кроме того, если не все параметры содержатся в списке аргументов (скажем, один из параметров определяет только тип возвращаемого значения), при вызове функции следует обязательно передать полный список параметров.

А теперь в качестве упражнения вы уже готовы создать функцию, которая возьмет на себя приведение типов. (В главе 2 «Продолжаем погружение: переменные» мы определили приведение типов как действие преобразования данных из одного типа в другой с сохранением исходного значения (по возможности).)

Эта функция иллюстрирует случай, когда один из параметров не входит в список аргументов:

```
template<class C, class T> C cast(T a)
{
     return (C)a;
}
```

Функция работает только для случаев, когда возможно приведение традиционным способом. Вот несколько примеров ее применения:

```
cout << cast<int, float>(3.5); //преобразует 3.5 в 3
cout << cast<float, int>(9); //преобразует 9 в 9.0
//ошибка – невозможно выполнить приведение int к char*
cout << cast<char*, int>(35769);
```

В подобных случаях исходя из аргументов функции можно определять второй параметр шаблона. При этом действует то же правило, что и для аргументов функции по умолчанию: опускать можно только параметры шаблона, расположенные в конце списка параметров. Скажем, для трех параметров можно опустить последний, два последних либо все параметры – разумеется, если они могут быть автоматически определены. Примеры:

```
//преобразует 3 в 3.0 - второй параметр автоматически устанавливается в int
cout << cast<float>(3);
//преобразует 3.5 в 3 - второй параметр - float
cout << cast<int>(3.5);
cout << cast(35); //ошибка - к какому типу приводить?
```

Уточнение шаблонов

В некоторых случаях шаблон функции должен особым образом обрабатывать конкретный тип данных, который является исключением из правила, верного для других типов. Как следствие, для этого типа необходимо создать отдельную версию шаблона. Такие версии носят название *уточненных* и используются только для целевых типов данных.

В разделе «Векторы» ниже по тексту главы мы расскажем о векторах. *Вектор* – это шаблон класса. Базовое определение шаблона класса `Vector` выглядит так:

```
template<class T> class Vector;
```

Это объявление кажется простым – обычный шаблон класса. Но помимо такого общего объявления у класса `Vector` есть еще и уточненное. Вектор должен особым образом работать с массивами указателей на `void`. Объявление уточненного шаблона класса `Vector`:

```
template<> class Vector<void*>;
```

Предшествующее объявление обозначает уточненную версию класса `Vector`, которая используется в случаях, когда значением параметра типа является `void*`. Уточненная версия используется вместо обычной. Для всех прочих типов используется первый вариант шаблона. Вот пример применения уточненных шаблонов:

```
#include <iostream>
#include <string>
using namespace std;
template<class T>class SayHi
{
     T data;
public:
     SayHi(T a) : data(a) {}
     void display() { cout << data << endl; }
};

template<> class SayHi<string>
{
     string data;
public:
     SayHi(string a) : data(a) {}
     void display() {cout << data.c_str() << endl; }
};
```

```
int main (void)
{
     SayHi<int> hi(5);
     hi.display();
     SayHi<string> hs("Здравствуй, мир");
     hs.display();
     return 0;
}
```

Вывод:

```
5
Здравствуй, мир
```

Шаблоны функций можно уточнять точно таким же образом. Синтаксис следующий:

```
template<> типВозвращаемогоЗначения имяФункции <особыйТип>(аргументы) {}
```

Здесь *особыйТип* – тип данных, с которым работает уточненный шаблон.

Упрощаем работу с шаблонами

Шаблоны могут становиться неудобными для пользователей классов и функций. По счастью, факт использования шаблонов можно просто скрыть.

Первый и самый простой способ – использовать `typedef`. Поскольку конструкцию `ИмяКласса<параметры>` можно использовать наравне с именем обычного класса, можно сократить имя шаблона класса, переименовав его при помощи `typedef`. Например, заменить `Vector<int>` на `IntVec`. Когда пользователю понадобится массив целых чисел, он воспользуется типом `IntVec`. Пользователям не обязательно знать, что `IntVec` входит в состав шаблона. Выполнить смену имени можно так:

```
typedef Vector<int> IntVec;
```

Примером такого использования `typedef` может послужить класс `string` стандартной библиотеки. Действительное имя шаблона класса для строк – `basic_string`. Этот шаблон имеет только один параметр – тип хранимых символов. Вот определение `typedef` для этого класса:

```
typedef basic_string<char> string;
```

Интересно, правда? Это означает, что при желании (приобрести бесполезные неудобства) можно использовать `basic_string<char>` вместо `string`.

Второй распространенный метод упрощения работы с шаблонами – создание значений по умолчанию для параметров шаблонов. Подобно аргументам функций, параметры шаблонов могут иметь значения по умолчанию, причем формат их задания точно такой же:

```
template<class A, class B = int, class C = float> class ABC;
```

В этом примере B и C по умолчанию равны int и float соответственно, если при создании объекта типы не указаны. Некоторые примеры применения этого шаблона:

```
ABC<int> a1; //A - int, B - int, C - float
ABC<int, string, char> a2; //A - int, B - string, C - char
//A - double, B - double, C - float
ABC<double, double> a3;
```

Как и в случае с аргументами функций, все параметры шаблона, имеющие значения по умолчанию, должны размещаться в конце списка.

Ранее в этой главе в разделе «Разрешение аргументов» мы рассказывали о третьем способе упрощения работы с шаблонами функций.

Если при вызове функции параметры отсутствуют, компьютер в некоторых случаях может сделать логический вывод о том, какими они должны быть. Таким образом, функция-шаблон может быть замаскирована под обычную функцию.

Четвертый способ применим только для шаблонов функций и описан в следующей главе.

Перегрузка шаблонов функций

Шаблоны функций могут перегружаться точно так же, как и обычные функции. Более того, можно создать много вариантов функции с одним именем, некоторые из которых будут шаблонами функций, а некоторые – обычными функциями.

Секундочку! Ведь если все сказанное о шаблонах – правда (они обобщают функции, позволяя им работать с любыми типами данных), зачем может понадобиться перегрузка шаблона функции? Как это не удивительно, причины тому существуют. И одна из основных причин – заставить шаблон функции вести себя в точности так, как ведут себя обычные функции. Можно создать перегруженные версии функции-шаблона, которые позволяют избежать применения параметров шаблона в некоторых случаях. Это гарантирует, что всегда вызывается нужная версия.

Предположим, складывая два целых числа, мы хотим получить результат с плавающей точкой (а потому что). Новый вариант функции сложения:

```
template<class T> T add (T a, T b)
{
     return a + b;
}
float add (int a, int b)
{
     return (float) a+b;
}
```

Если теперь вызвать функцию add для двух целых чисел, выполнится перегруженная версия (вторая). Для всех прочих типов данных вторая версия функции просто игнорируется. Откуда компьютер знает, какую из версий вызвать? Для принятия решений он пользуется определенным набором правил. Для каждого вызова перегруженного шаблона функции компьютер выполняет следующие шаги:

1. Находит все версии функции, которые потенциально могут быть вызваны для указанных аргументов. Для вызова add(5,3) это правило выбирает обе версии функции сложения. Для вызова add(75, 'a') – ни одной, поскольку обе версии функции имеют аргументы одинакового типа.
2. Если один из шаблонов является уточнением другого, предпочтение отдается уточненному шаблону.
3. Для выбранных функций применяются обычные правила перегрузки.
4. Если осталась функция и шаблон функции, предпочтение отдается функции. Для вызова add(5,3) кандидатами на выполнение вплоть до этого шага были обе версии. Будет выполнена функция.
5. Если возможности исчерпаны, возникает ошибка. Если осталось два или более вариантов, вызов считается неоднозначным, возникает ошибка.

Ранее мы уже говорили, что если значения параметров не очевидны из списка аргументов, они должны указываться при вызове функции – например, при вызове функции сложения для целого и значения с плавающей точкой. К счастью, есть способы обойти это правило. Первый способ – создать перегруженные версии функции сложения, вызывающие функцию-шаблон с корректными параметрами. В таком случае пользователю уже не надо беспокоиться о параметрах шаблона. Пример подобной перегрузки для функции add:

```
inline float add(int a, float b) { return add<float>(a,b);}
inline float add(float a, int b) { return add<float>(a.b); }
```

Если теперь вызвать функцию add для целого и числа с плавающей точкой, не указав параметр, компьютер сам разберется, что следует делать. Обратите внимание, что последние функции объявлены как подставляемые (inline). Это позволяет получить удобство обращения к функции без параметра – без необходимости расплачиваться памятью. Мы говорим компьютеру, что при сложении целого и числа с плавающей точкой следует возвращать результат с плавающей точкой. Вот примеры использования новой версии функции сложения:

```
cout << add(3.5, 7); //возвращает 10.5
cout << add(7, 3.5); //возвращает 10.5
```

Пусть вас не пугает перегрузка шаблонов функций. В действительности это всего лишь стандартная перегрузка за тем исключением, что об-

работка происходит по чуть более сложным правилам. Столкнувшись с затруднениями, просто обращайтесь к списку правил.

Работа со стандартной библиотекой

Стандартная библиотека C++ задумывалась как средство, способное пригодиться любому программисту на C++. Вследствие этого обстоятельства она должна быть как можно более общей. Например, будь векторы стандартной библиотеки спроектированы для работы лишь с одним конкретным типом, их полезность была бы серьезно ограничена.

При создании стандартной библиотеки разработчики языка C++ установили следующие требования:

- Библиотека должна быть удобной, эффективной, полезной и безопасной в применении для всех пользователей.
- Чтобы извлечь пользу из функциональности библиотеки, пользователь не должен писать дополнительный код.
- Библиотека должна иметь простой интерфейс.
- Библиотека должна до конца выполнять возложенные на нее задачи.
- Библиотека должна хорошо интегрироваться со встроенными типами данных и ключевыми словами.

На основе этих требований и была спроектирована библиотека. По нашему мнению, разработчикам полностью удалось выполнить каждое из требований.

Стандартная библиотека представлена набором заголовочных файлов, каждый из которых служит определенной цели. Функциональность библиотеки разделена на десять категорий: контейнеры, общие вспомогательные функции, итераторы, алгоритмы, диагностика, строки, потоки, региональные настройки, поддержка языка, работа с числами.

В этом разделе мы расскажем о некоторых контейнерах и типе string, который используем с главы 1 «Путешествие начинается». *Контейнер* – это тип данных, позволяющий хранить набор значений. *Строки* (как вам известно) хранят текст.

Поскольку для обобщения кода используются именно шаблоны, стандартная библиотека является богатым источником примеров применения шаблонов.

Строки

Все стандартные средства C++ для работы со строками содержатся в библиотеке `<string>`. Как мы уже упоминали в этой главе, `string` является синонимом для `basic_string<char>`. `basic_string` – это шаблон класса, который способен работать с любым видом символов. Напри-

мер, можно создать класс для хранения египетских иероглифов и дать ему имя E_char. Затем можно создать строковый объект из египетских иероглифов следующим образом:

```
basic_string<E_char> egyptian;
```

Все символы типа, передаваемого в качестве параметра basic_string, должны иметь ряд свойств, определяемых другим шаблоном класса по имени char_traits. Базовое объявление char_traits:

```
template<class Ch> struct char_traits{};
```

Структура (struct) не отличается от класса, но все ее компоненты по умолчанию являются общими, а не частными. По странному стечению обстоятельств этот класс-шаблон никогда не используется. Каждому типу символов соответствует уточненный шаблон (в том числе и в нашем примере):

```
template<> struct char_traits<char>;
```

Это уточненный шаблон для типа символов char. Данный шаблон класса char_traits хранит свойства конкретного типа символов. Взгляните на упрощенную версию реализации char_traits<char>:

```
template<> struct char_traits<char> {
     typedef char char_type; //стандартизация имени
     //стандартизация имени для целочисленного представления
     typedef int int_type;
     //копирует второй аргумент в первый
     static void assign(char_type& a, const char_type& b);
     //преобразование int в char
     static char_type to_char_type(const int_type& a);
     //преобразование char в int
     static int_type to_int_type(const char_type& a);
     //являются ли два целых числа одним и тем же символом?
     static bool eq_int_type(const int_type& i,
          const int_type& j);
     //одинаковы ли два символа?
     static bool eq(const char_type& a, const char_type& b);
     //первый символ меньше второго?
     static bool lt(const char_type& a, const char_type& b);
     //поместить n копий s2 в s
     static char_type* move(char_type* s,
          const char_type* s2, size_t n);
     //поместить n копий s2 в s
     static char_type* copy(char_type* s,
          const char_type* s2, size_t n);
     //скопировать n a в s
     static char_type* assign(char_type* s,
          size_t n, char_type a);

     //сравнивает 2 символа
     static int compare(const char_type* s,
```

```
        const char_type* s2, size_t n);
    //возвращает длину массива символов
    static size_t length(const char_type* c);
    //поиск c в s
    static const char_type* find(const char_type* s,
        int n, const char_type& c);
};
```

Вот и первое знакомство с библиотеками. Поначалу подобный код кажется совершенно невразумительным, но по мере набора опыта вы привыкнете к подобным конструкциям. Основное затруднение вызывает применение в большинстве библиотек идентификаторов вроде `_E`, которые затрудняют запоминание семантики. Использование подобных имен в библиотеках связано с необходимостью исключить конфликты с любыми другими именами.

Как видите, `char_traits<char>` определяет базовые операции для работы с типом `char` и символьными массивами.

Первые несколько строк кода приведенного определения содержат две конструкции `typedef`. Они дают стандартные имена символьному типу и целочисленному типу, чтобы класс строк имел возможность обращаться к значениям `char_traits::char_type` и `char_traits::int_type` независимо от того, о какой разновидности символов идет речь.

Раньше вы еще не встречались с типом `size_t`, и он может вызывать затруднения. Считайте его типом беззнакового целого.

Реализация методов этого класса-шаблона достаточно проста. Вот, например, реализация первого варианта функции `assign`:

```
template<> void char_traits<char>::assign(char_type& a,
    char_type& b)
{
    a = b;
}
```

Для каждого типа символа, используемого совместно с `basic_string`, должны быть определены уточненные шаблоны `char_traits` и реализованы все перечисленные функции. Это гарантирует, что класс `basic_string` сможет выполнить подобные операции для каждого символа.

Вторым по распространенности типом символа является длинный символ (`wchar_t`). Он похож на обычный тип, но занимает два байта (что позволяет хранить в наборе символов гораздо больше знаков).

Базовое объявление для шаблона класса `basic_string`:

```
template<class C, class T = char_traits<C>,
    class A = allocator<C>> class std::basic_string;
```

Как видите, класс является частью пространства имен `std` (как и вообще все сущности стандартной библиотеки), а шаблон имеет три параметра. Два из них имеют значения по умолчанию, а следовательно,

обязательным является только один. C определяет тип символа, T — это шаблон char_traits, связанный с этим типом, A — тип, называемый *распределителем*, он отвечает за выделение памяти для хранения символа. По умолчанию T и A являются уточненными версиями соответствующих типу символа классов. Практически во всех случаях эти параметры можно не указывать.

Класс basic_string, подобно классу char_traits, дает стандартным сущностям имена при помощи конструкций typedef. Вот эти конструкции:

```
//тип используемого char_traits (как правило, char_traits<char>)
typedef T traits_type;
//тип менеджера памяти (как правило, allocator<char>)
typedef typename A allocator_type;

//тип хранимых символов (как правило, char)
typedef typename T::char_type value_type;
//какой-либо беззнаковый тип (возможно, unsigned int)
typedef typename A::size_type size_type;

//ссылка на отдельный символ (как правило, char&)
typedef typename A::reference reference;
//указатель на отдельный символ (как правило, char*)
typedef typename A::pointer pointer;

//тип для стандартного итератора
typedef compiler_dependant iterator;
//тип для обратного итератора
typedef std::reverse_iterator<iterator> reverse_iterator;

//используется для представления символов строки
static const size_type npos;
```

В начале параметра шаблона следует использовать typename, чтобы получить доступ к его сущностям. Например, чтобы определить value_type, необходимо обратиться к char_type из класса char_traits. *compiler_dependant* говорит о том, что размещенный в этой точке код зависит от применяемого компилятора. Для каждого компилятора код будет иным, но в любом случае речь идет о какой-либо разновидности стандартного итератора. npos представляет «все символы». Когда необходимо передать число символов для обработки какой-то операцией, можно использовать npos для представления их всех (с определенной точки и до конца). Имейте в виду, npos не обязательно означает «строка целиком». Наиболее точное определение — «вся эта строка».

Создание строки

Стандартная библиотека позволяет создать объект string различными способами. Можно использовать другой строковой объект, массив символов, литеральное значение, пустой конструктор и другие варианты. Мы рассмотрим конструкторы по очереди. Если необходимо создать одну строку на основе другой:

```
//создать строку на основе другой, используя ее полностью или частично
basic_string(const basic_string& s, size_type pos = 0,
     size_type n = npos, const A& a = A() );
```

Первый аргумент представляет другую строку. Здесь можно использовать любой вариант basic_string. Второй аргумент определяет позицию (строка в данном случае считается массивом символов), с которой начинается копирование значений. Третий аргумент задает количество копируемых символов. Он по умолчанию имеет значение npos, что примерно соответствует максимально возможной величине строки. Это означает – «до конца s». pos = 0 означает «начать с начала», а n = npos означает – копировать все символы. Четвертый аргумент определяет способ выделения памяти для хранения строки. И хотя мы не будем здесь подробно останавливаться на четвертом аргументе – эта тема слишком сложна для новичка – вот некоторые примеры использования данного конструктора:

```
string s = "Здравствуй, мир"; //обьявляем строку, с которой можно работать
string s1(s); //строки s1 и s содержат значение "Здравствуй, мир"
string s2(s, 6, 2); //начинаем с позиции 6, копируем 2 символа - s2 содержит "тв"
string s3(s, 6); //ошибка - необходим третий аргумент
```

Последний вызов конструктора является ошибочным, поскольку компьютер путает его с другим конструктором (речь о котором пойдет ниже). Второй и третий конструкторы создают строку на основе символьного массива:

```
basic_string(const C* p, size_type n, const A& a = A());
basic_string(const C* p, const A& a = A());
```

Первый аргумент, p, представляет символьный массив. n – число копируемых символов, а параметр a связан со способом выделения памяти под хранение строки (оставьте значение по умолчанию, если не занимаетесь целенаправленным изучением распределителей). Если опустить значение n, для создания строки будут использованы все символы массива (интерпретируемые в этом случае как строка в стиле C). Некоторые примеры применения этих конструкторов:

```
char* p = "Здравствуй, мир";
string s1(p); //s1 имеет значение "Здравствуй, мир"
string s2(p, 5); //s2 имеет значение "Здрав"
string s3(p+12, 3); //s3 имеет значение "мир"
```

Последний конструктор создает строку из последовательности символов. Последовательность символов передается вместе с итератором. В следующем разделе мы подробно расскажем об итераторах, а пока что лишь самые краткие сведения. В случае контейнера или строки существует возможность вызывать методы begin() и end() с целью получения указателей на первый и последний элементы, хранимые контейнером или строкой. Эти значения могут использоваться для копи-

рования всего набора символов в новую строку. Конструктор выглядит следующим образом:

```
//использует все символы, от первого до последнего
template<class I> basic_string(I first, I last,
     const A& a = A());
```

Вот некоторые варианты применения этого конструктора:

```
string s = "Здравствуй, мир";
char* c = "Здравствуй, мир";
string s1(s.begin(), s.end()); //s1 имеет значение "Здравствуй, мир"
string s2(c, c+5); //s2 имеет значение "Здрав"
string s3(c+12, c+15); //s3 имеет значение "мир"
```

При вызове этого конструктора можно пользоваться любой парой указателей. Будут скопированы все символы между двумя адресами. Не забудьте убедиться, что первый аргумент предшествует второму в памяти и что все промежуточные значения являются символьными.

Помимо конструкторов класс `basic_string` имеет деструктор, выполняющий необходимые действия по очистке памяти:

```
~basic_string();
```

Перебор символов строки

Итераторы являются общей сущностью для всех контейнеров и строк. Итераторы позволяют перебирать значения, хранимые контейнером или строкой. Все итераторы имеют похожий интерфейс, и это обстоятельство является одним из главных достоинств стандартной библиотеки. Изучив интерфейс на примере одного контейнера, вы будете с легкостью применять его и для других. Значения, к которым позволяет обращаться итератор для `basic_string`, представлены отдельными символами строки. Код итераторов содержится в библиотеке `<iterator>`.

Некоторые контейнеры (в контексте этой темы будем считать строку контейнером символов) позволяют эффективно выполнять определенные операции, а некоторые нет. Например, список не всегда позволяет получить прямой доступ к любому из хранимых значений. Иначе говоря, списки не реализуют *прямой доступ*. С другой стороны, векторы именно такой доступ реализуют. Существует пять категорий итераторов, каждая из которых связана с определенным набором выполнимых операций доступа. Эти категории: ввод, вывод, прямой, двунаправленный, произвольный доступ.

Итератор категории ввода позволяет только читать значения элемента и перебирать элементы контейнера (ввод от контейнера к пользователю). Итератор категории вывода позволяет только изменять значения элементов и перебирать элементы контейнера (вывод от пользователя в контейнер). Прямой итератор позволяет читать и изменять значения переменных, а также выполнять прямой перебор элементов. Двуна-

правленный позволяет читать и изменять значения элементов, а также выполнять прямой и обратный перебор элементов. Итератор произвольного доступа позволяет читать и изменять значения элементов, а также перебирать элементы в любом порядке.

Операции, доступные в каждой из категорий, сведены в табл. 9.1.

Таблица 9.1. Операции итераторов

Тип операции	Вывод	Ввод	Прямой	Прямой и обратный	Произвольный
Прямой перебор (`++`)	Да	Да	Да	Да	Да
Обратный перебор (`--`)	Нет	Нет	Нет	Да	Да
Произвольный доступ (`[], +=, -=, +, -`)	Нет	Нет	Нет	Нет	Да
Чтение (`*`*Итератор*)	Нет	Да	Да	Да	Да
Запись (`*Итератор =`)	Да	Нет	Да	Да	Да
Компонентный доступ (`->`)	Нет	Да	Да	Да	Да
Сравнение (`==, !=`)	Нет	Да	Да	Да	Да
Сравнение (`<, >, <=, =>`)	Нет	Нет	Нет	Нет	Да

Как можно понять из табл. 9.1, итераторы схожи с указателями на массивы. По сути дела, указатель на массив является итератором этого массива. Все итераторы предоставляет операторы, соответствующие операторам для указателей. Поначалу итератор может казаться указателем, поскольку применяется точно так же.

Описанные категории итераторов составляют иерархию:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag :
     public forward_iterator_tag {};
struct random_access_iterator_tag :
     public bidirectional_iterator_tag {};
```

Эта иерархия определяет возможности для каждого типа. Итератор произвольного доступа может выступать в роли итератора ввода, но не наоборот. Итераторы вывода не входят в эту иерархию – они несколько выпадают из общего ряда. Включение этих итераторов потребовало бы создания второго базового класса для `forward_iterator_tag`, но это не дает никаких существенных преимуществ, а потому не имеет смысла.

Итак, вы готовы к изучению простого класса итератора. Через несколько минут итераторы будут казаться вам уже не столь сложными:

```
template <class C, class T, class Dist = ptrdiff_t,
```

```
        class Ptr = T*, class Ref = T&>
struct iterator {
        //класс из приведенной выше иерархии
        typedef C iterator_category;
        //тип элемента, с которым работает итератор
        typedef T value_type;
        //тип расстояния между парой элементов
        typedef Dist difference_type;
        typedef Ptr pointer; //указатель на тип элемента
        typedef Ref reference; //ссылка на тип элемента
};
```

Здесь `C` определяет категорию итератора (с помощью типа из приведенной иерархии), `T` – тип элемента, `Dist` – это тип, который используется для измерения расстояния между элементами, `Ptr` – тип указателя на элемент, а `Ref` – ссылочный тип для элементов. `ptrdiff_t` по определению является стандартным расстоянием между двумя указателями. Как видите, этот класс лишь определяет имена и является базовым классом, который наследуется каждым контейнером. Используйте этот класс для определения перечисленных типов в соответствии с собственными задачами.

С классом символов связан шаблон `char_traits`, а с каждым итератором – шаблон `iterator_traits`. Класс для итераторов выглядит так:

```
template<class Iter> struct iterator_traits {
        typedef typename
             Iter::iterator_category iterator_category;
        typedef typename Iter::value_type value_type;
        typedef typename Iter::difference_type difference_type;
        typedef typename Iter::pointer pointer;
        typedef typename Iter::reference reference;
};
```

Проектируя алгоритм с итераторами, программист может воспользоваться категорией итератора `iterator_traits<Iter>::iterator_ctegory` для реализации различного поведения различных типов итераторов. В этом случае пользователю кода необязательно даже знать об итераторах. Подходящая ситуации версия кода используется автоматически. Это еще один мощный способ сокрытия реализации от пользователя.

Для всех строк и контейнеров существует стандартный способ генерации итератора элементов. Методы итерации для класса `basic_string`:

```
//создает итератор, указывающий на первый элемент
iterator begin();
//итератор, указывающий на следующий за последним элемент
iterator end();
```

В случаях когда итератор используется для получения последовательности символов, требуются оба этих метода. Например, конструк-

тор типа `string`, работающий на основе входного диапазона, может воспользоваться результатами работы этих методов для создания строки:

```
string s = "Здравствуй, мир";
string s1(s.begin(), s.end()); //s1 имеет значение "Здравствуй, мир"
```

Иногда требуется перебрать элементы контейнера в обратном порядке. В основе подобных действий лежит шаблон класса `reverse_iterator`. Он наследует класс итератора. Шаблон класса `reverse_iterator`:

```
template<class Iter> class reverse_iterator :
public iterator<iterator_traits<Iter>::iterator_category,
iterator_traits<Iter>::value_type,
iterator_traits<Iter>::difference_type,
iterator_traits<Iter>::pointer,
iterator_traits<Iter>::reference> {

protected:
     //скрытый перебор в обратном порядке
     //с помощью обычного итератора
     Iter current;
public:
     //стандартное имя для типа итератора
     typedef Iter iterator_type;

     reverse_iterator() : current() {} //конструктор по умолчанию
     //конструктор, аргументом которого является обычный итератор
     reverse_iterator(Iter x) : current(x) {}
     //конструктор, аргументом которого является другой обратный итератор
     template<class U> reverse_iterator(
          const reverse_iterator<U>& x) : current(x.base()) {}

     //возвращает обычный итератор, используемый классом
     Iter base() const {return current;}

     //разыменование
     reference operator* () const { Iter tmp = current;
          return *--tmp;}
     pointer operator-> () const; //оператор компонентного доступа
     reference operator[] (difference_type n) const;

     //возврат (обратный)
     reverse_iterator& operator++ () {--current;
          return *this; }
     reverse_iterator operator++ (int) {
          reverse_iterator t = current; --current; return t;}
     //перемещение вперед (обратное)
     reverse_iterator& operator-- () {++current;
          return *this;}
     reverse_iterator operator-- (int) {
          reverse_iterator t = current; ++current; return t;}

     reverse_iterator operator+ (difference_type n) const;
     reverse_iterator operator+= (difference_type n);
```

```
        reverse_iterator operator- (difference_type n) const;
        reverse_iterator operator -= (difference_type n);
    };
```

`reverse_iterator` содержит итератор `current`, который используется для перебора элементов. `reverse_iterator` самостоятельно изменяет значение этого итератора и использует его, когда пользователь обращается к элементу. Оператор ++ для итератора имеет значение «перейти к следующему элементу». Однако обратный итератор перебирает элементы в обратном направлении, поэтому оператор ++ для обратного итератора эквивалентен оператору -- для нормального (перемещение в контейнере происходит в одном направлении). Поэтому `current` получает отрицательное приращение в реализации оператора ++.

Обычно, когда итератор достигает конца контейнера, он указывает на элемент, следующий за последним. Но обратный итератор имеет другое направление движения, и для него конец элементов – это элемент, предшествующий первому значению. За пределами контейнера элементы не существуют, и обращение к ним может иметь неприятные последствия. Чтобы избежать подобных осложнений, `current` указывает на элемент, следующий за элементом, на который указывает итератор. Иначе говоря, когда итератор достигает конца, `current` указывает на первый элемент контейнера. Такая система более надежна, чем вариант с элементом, предшествующим первому.

Класс `reverse_iterator`, как и все итераторы, поддерживает операторы, позволяющие обращаться с ним в точности как с указателем.

Чтобы создать обратный итератор на основе контейнера или строки, следует воспользоваться методами `rbegin()` и `rend()`. Вот эти методы для класса `basic_string`:

```
//обратный итератор - начинает работу в конце строки
//и движется к ее началу
reverse_iterator rbegin();
//обратный итератор, начинающий работу в начале строки
reverse_iterator rend();
```

Вот некоторые примеры использования `reverse_iterator`:

```
string s = "Здравствуй, мир";
cout << *s.rbegin(); //отображается "р"
cout << *++s.rbegin(); //отображается "и"
cout << s.rbegin()[4]; //отображается ","
```

Доступ к строкам

Стандартная библиотека предоставляет многочисленные способы для доступа к данным, хранящимся в строке. В этом разделе мы расскажем о каждом из них, чтобы обеспечить читателей твердыми знаниями о классе `basic_string`.

По сути дела, класс basic_string является всего лишь доработанным символьным массивом, и разумно будет предположить, что с ним можно работать как с массивом. И это, по счастью, действительно так. Доступ к отдельным элементам массива можно получить двумя способами: с помощью оператора индекса ([]) и с помощью метода at(). Вот их объявления:

```
reference operator[] (size_type n);
reference at(size_type n);
```

n — это индекс, определяющий позицию в строке в обоих случаях (нумерация начинается с нуля, как в массивах). Различие между вариантами заключается в том, что метод at() проверяет существование индекса строки (в случае недопустимого индекса возникает ошибка). Оператор индекса подобной проверкой не затрудняется. Он предполагает, что символ существует, и просто обращается по указанному адресу памяти. Примеры использования этих методов:

```
string s = "Asus - лучшие материнские платы!";
cout << s[3]; //четвертый символ - 's'
cout << s.at(2); //третий символ - 'u'
```

Как видите, эти методы весьма прозрачны и просты в применении.

Второй распространенный метод доступа к строкам связан с преобразованием их в строки в стиле языка C. В главе 1 вы узнали об одном предназначенном для этого методе, c_str(). Два других метода называются data() и copy(). Приведем объявления всех трех методов:

```
const C* c_str() const;
const C* data() const;
size_type copy(C* p, size_type n, size_type pos = 0) const;
```

Метод data() возвращает указатель на константный массив символов. Строка сохраняет права на этот массив, поэтому его нельзя изменить. Он предназначен только для просмотра. Метод c_str() делает то же, но добавляет нулевой символ в конец строки (после чего массив становится корректной строкой в стиле C). Метод copy() используется для копирования элементов в массив с целью последующей работы с ними. Метод копирует n элементов в массив p начиная с элемента pos. Метод возвращает число скопированных элементов. Несколько примеров применения этих методов:

```
string s = "SyncMaster 3";
char array[15];
s.copy(array, 4, 7); //массив содержит "ster"
array[4] = 0; //теперь это строка в стиле C
cout << s.c_str(); //старый знакомый
const char* p = s.data();
p[4] = 'a'; //ошибка, массив нельзя изменять
```

Область применения этих функций ограничена в основном строками в стиле C.

Дополнительную информацию о строке можно получить, сравнив ее с другой строкой. Строку basic_string можно сравнить с другой строкой basic_string или с символьным массивом (того же символьного типа). Для сравнения строк служит метод compare(). Он существует в нескольких перегруженных вариантах:

```
int compare(const basic_string& s) const;
int compare(const C* p) const;
int compare(size_type pos, size_type n,
     const basic_string& s) const;
int compare(size_type pos, size_type n,
     const basic_string& s, size_type pos2,
     size_type n2 = npos) const;
```

Каждая версия метода compare() возвращает целое число. Это число 0, если строки равны (по версии метода char_trait<C>::compare()), отрицательное, если первая строка меньше второй, и положительное, если первая строка больше второй. Мысль о том, что одна строка больше другой, может показаться странной, но на деле сравниваются целочисленные значения символов (поэтому буква a меньше буквы b).

С первой парой вариантов все достаточно просто. Достаточно передавать строку или символьный массив, и метод compare() выполнит сравнение. Однако иногда требуется сравнение фрагментов строк. Здесь на сцене появляются две последних версии метода. pos – эта позиция, с которой начинается сравнение, n – число символов для сравнения. В третьей версии метода часть строки сравнивается со строкой s. В четвертой – сравниваются части обеих строк. Примеры:

```
string s1 = "Get the Fish";
string s2 = "Getting Started";
char array[] = "5-Port Starter Kit";
cout << s1.compare(s2); //выводит -1
cout << s2.compare(array); //выводит 1
cout << s1.compare(0,3,s2); //выводит -1
cout << s1.compare(0.3.s2.0,3); //выводит 0
```

Эти методы вам очень пригодятся, если понадобится отсортировать набор строк (например, чтобы расположить имена в алфавитном порядке).

Кроме того, в стандартной библиотеке существует функциональность стандартных операторов сравнения (==, !=, <, >, <= и >=). Они определены вне класса basic_string. Вот некоторые из объявлений:

```
template<class C, class T, class A> bool operator==(
     const basic_string<C,T,A>&, const basic_string<C,T,A>&);
template<class C, class T, class A> bool operator==(
     const C*, const basic_string<C,T,A>&);
template<class C, class T, class A> bool operator==(
     const basic_string<C,T,A>&, const C*);
```

Эти функции позволяют определить равенство двух строк или строки и массива. Вот пример сравнения строки и символьного массива:

```
string continent = "Европа";
if (continent == "Европа") //true
     cout << "Этот континент - Европа!";
```

Не стоит задаваться вопросом, как именно компьютер определяет равенство строк. По сути дела, строки равны, если содержат одинаковые последовательности символов.

Временами возникает необходимость найти определенную последовательность символов (*подстроку*) в строке. В классе `basic_string` существует двадцать четыре различных способа сделать это. Первый способ – использовать метод `find()`. Он производит поиск начиная с конкретной позиции строки и возвращает позицию (индекс), если подстрока найдена. В противном случае возвращается значение `npos` (представляющее недопустимый индекс). Искомая последовательность символов может быть строкой, массивом символов либо отдельным символом. Вот объявления всех версий метода `find()`:

```
size_type find(const basic_string& s, size_type i = 0) const;
size_type find(const C* p, size_type i, size_type n) const;
size_type find(const C* p, size_type i = 0) const;
size_type find(C c, size_type i = 0) const;
```

Здесь `i` – это позиция, с которой начинается поиск, а `n` – число символов из массива, для которых следует выполнять поиск. Примеры:

```
string s = "The basic_string class is extremely versatile.";
string s1 = "basic_string";
cout << s.find(s1); //выводит 4
//выводит 4294967295 - свидетельствуя о том, что
//строка не найдена
cout << s.find(s1, 5);
//отображает 17 - найдено вхождение строки "class"
cout << s.find("classes", 17, 5);
//выводит 4294967295 - строка "classes" не найдена
cout << s.find("classes", 17);
//выводит 19 - поиск начат за первым символом 'a'
cout << s.find('a', 6);
```

Иногда бывает удобнее (или более эффективно) производить поиск от конца строки к ее началу. В этом случае следует воспользоваться методом `rfind()`. Метод производит поиск последовательности символов так же, как и `find()`, только в обратном направлении. Объявления функций `rfind()`:

```
size_type rfind(const basic_string& s,
     size_type i = npos) const;
size_type rfind(const C* p, size_type i, size_type n) const;
```

```
size_type rfind(const C* p, size_type i = npos) const;
size_type rfind(C c, size_type i = npos) const;
```

Как видите, кроме имен методов мало что изменилось. Исключением являются значения по умолчанию для аргументов. Для `find()` значением по умолчанию начальной позиции является `0` (поиск начинается с начала строки), а для `rfind()` — `npos` (поиск с конца). Несколько примеров:

```
string s = "The string class sure has a lot of methods.";
string s1 = "of";
cout << s.rfind(s1); //выводит 32
//выводит 4294967295 - подстрока не найдена
cout << s.rfind(s1, 31);
//выводит 35 - позиция подстроки "meth"
cout << s.rfind("methane", string::npos, 4);
```

Кроме того, можно производить поиск конкретного символа из последовательности. Эту задачу выполняет метод `find_first_of()`. Метод выполняет прямой поиск от указанной позиции и, если находит символ, входящий в последовательность, возвращает его позицию. Метод `find_last_of()` работает аналогичным образом, но выполняет поиск в обратном направлении (как `rfind()`). Объявления функций `find_first_of()`:

```
size_type find_first_of(const basic_string& s,
     size_type i = 0) const;
size_type find_first_of(const C* p, size_type i,
     size_type n) const;
size_type find_first_of(const C* p, size_type i = 0) const;
size_type find_first_of(C c, size_type i = 0) const;

size_type find_last_of(const basic_string& s,
     size_type i = npos) const;
size_type find_last_of(const C* p, size_type i,
     size_type n) const;
size_type find_last_of(const C* p, size_type i = npos) const;
size_type find_last_of(C c, size_type i = npos) const;
```

Некоторые примеры для этой группы функций:

```
string s = "We need a bigger gun.";
string s1= "Mercenaries.";
//выводит 1 - позиция символа 'e'
cout << s.find_first_of(s1);
//выводит 15 - позицию 'r' из слова 'bigger'
cout << s.find_last_of("tree.", 16);
```

Иногда необходимо выполнить обратное действие. Можно производить поиск первого (или последнего) символа строки, который не принадлежит определенной последовательности символов. Эта задача возложена на методы `find_first_not_of()` и `find_last_not_of()`. Объявления:

```
size_type find_first_not_of(const basic_string& s,
```

```
     size_type i = 0) const;
size_type find_first_not_of(const C* p, size_type i,
     size_type n) const;
size_type find_first_not_of(const C* p,
     size_type i = 0) const;
size_type find_first_not_of(C c, size_type i = 0) const;

size_type find_last_not_of(const basic_string& s,
     size_type i = npos) const;
size_type find_last_not_of(const C* p, size_type i,
     size_type n) const;
size_type find_last_not_of(const C* p,
     size_type i = npos) const;
size_type find_last_not_of(C c, size_type i = npos) const;
```

Примеры для этой группы функций:

```
string s = "The sock is on the floor.";
string s1 = "These rocks";
//выводит 9 - позицию буквы 'i' из слова 'is'
cout << s.find_first_not_of(s1);
//выводит 20 - позицию 'l' из слова "floor."
cout << s.find_last_not_of("roof.");
```

Новые строки могут создаваться из фрагментов (подстрок) существующих. К примеру, на основе строки, в которой хранится значение `"Здравствуй, мир"`, можно создать новую строку со значением `"мир"`. С этой целью можно использовать метод `substr()`. Указав начальную позицию и длину, можно определить, какой именно фрагмент строки использовать. Объявление метода `substr()`:

```
basic_string substr(size_type i = 0, size_type n = npos) const;
```

Здесь `i` — позиция, с которой начинается копирование, `n` — число копируемых символов. Пример:

```
string s = "уничтожение";
string s1 = s.substr(4,2);
cout << s1.c_str(); //выводит "то"
```

Обратите внимание, что по умолчанию метод создает копию полной строки.

В качестве дополнения к методу `substr()` стандартная библиотека предоставляет способ сформировать новую строку из двух существующих. Для этой цели используется оператор +. Сложение двух строк называется *слиянием* (сцепкой, конкатенацией). Все функции слияния объявлены вне класса `basic_string`. Вот объявления функций слияния для строк:

```
template<class C, class T, class A> basic_string<C,T,A>
     operator+ (const basic_string<C,T,A>&,
```

```
    const basic_string<C,T,A>&);
template<class C, class T, class A> basic_string<C,T,A>
    operator+ (const C*, const basic_string<C,T,A>&);
template<class C, class T, class A> basic_string<C,T,A>
    operator+ (C, const basic_string<C,T,A>&);
template<class C, class T, class A> basic_string<C,T,A>
    operator+ (const basic_string<C,T,A>&, const C*);
template<class C, class T, class A> basic_string<C,T,A>
    operator+ (const basic_string<C,T,A>&, C);
```

При помощи этих функций можно создавать строку из двух существующих строк, из строки и символьного массива либо строки и символа. Примеры:

```
string s = "Здравствуй, ";
string t = "мир";
cout << s + ' ' + t; //выводит "Здравствуй, мир"
cout << s + ' ' + "народ!"; // выводит "Здравствуй, народ!"
```

Кроме того, класс `basic_string` реализует функции для работы с памятью данных строки и статистической информации. Одна из статистических характеристик доступна посредством вызова метода `size()`, который возвращает число символов в строке. То же делает метод `length()`. Метод `empty()` возвращает значение `true`, если строка пуста (и `false` в противном случае). Метод `max_size()` возвращает максимально возможный для строки размер. Объявления этих методов:

```
size_type size() const;
size_type length() const {return size();}
bool empty() const { return size == 0;}
size_type max_size() const;
```

Работа со строками

Возможность работы со строками разнообразными способами является неотъемлемой характеристикой класса строк. По счастью, `basic_string` более чем соответствует этому требованию. Он предоставляет мощные средства для работы со строками.

Самый простой и наиболее полезный способ изменения значения строки связан с оператором присваивания (`=`). Класс `basic_string` перегружает эти операторы, чтобы тип `string` больше соответствовал встроенным типам. Объявления методов присваивания:

```
basic_string& operator= (const basic_string& s);
basic_string& operator= (const C* p);
basic_string& operator= (C c);
```

Эти функции позволяют выполнить присваивание строки, символьного массива либо отдельного символа строковому объекту. Помните, что присвоение нового значения уничтожает предыдущее.

Для выполнения тех же операций существует и обычный метод assign(), с более гибкими аргументами. Вот его объявления:

```
basic_string& assign(const basic_string&); //полная строка
//фрагмент строки из n символов, начинается
//с позиции pos
basic_string& assign(const basic_string& s, size_type pos, size_type n);
//первые n элементов из p
basic_string& assign(const C* p, size_type n);
basic_string& assign(const C* p); //p целиком
basic_string& assign(size_type n, C c); //n копий c
//все символы, с первого до последнего
typedef<class I> basic_string& assign(I first, I last);
```

Этот метод подражает универсальности конструктора. Было бы странно иметь возможность создать объект определенным способом, но не иметь возможности тем же способом выполнять присваивание. Возвращаемое значение basic_string& идентично присвоенному значению, и его можно в большинстве случаев игнорировать.

Добавление символов в конец строки – распространенная операция над строками. В классе basic_string она реализована в виде оператора += и метода append. Объявления функций добавления:

```
basic_string& operator+= (const basic_string& s);
basic_string& operator+= (const C* p);
basic_string& operator+= (C c);

basic_string& append(const basic_string& s);
//добавление фрагмента строки
basic_string& append(const basic_string& s, size_type pos,
     size_type n);
//добавление фрагмента массива символов
basic_string& append(const C* p, size_type n);
//добавление полного массива символов
basic_string& append(const C* p);
basic_string& append(size_type n, C c); //добавление n копий c
//добавление значений из итератора
template<class I> basic_string& append(I first, I last);
```

Оператор += позволяет добавлять строки, символьные массивы либо отдельные символы к строке. С помощью метода append можно добавлять строки, фрагменты строк, фрагменты символьных массивов, символьные массивы, множественные копии отдельных символов и значения из итераторов.

Добавляемые значения не обязательно должны помещаться в конец строки благодаря существованию методов вставки. Объявления метода insert:

```
//вставка строки непосредственно перед символом в позиции pos
basic_string& insert(size_type pos, const basic_string& s);
```

```
//вставка фрагмента строки непосредственно перед символом в позиции pos
basic_string& insert(size_type pos, const basic_string& s,
     size_type pos2, size_type n);
//вставка фрагмента массива в позиции pos
basic_string& insert(size_type pos, const C* p, size_type n);
//вставка массива в позиции pos
basic_string& insert(size_type pos, const C* p);
//вставка n копий с в позиции pos
basic_string& insert(size_type pos, size_type n, C c);

//вставка с непосредственно перед символом в позиции  pos - возвращается
итератор в с
iterator insert(iterator pos, C c);
//вставка n копий  с в позиции pos
void insert(iterator pos, size_type n, C c);
//вставка последовательности  в позиции pos
template<class I> void insert(iterator pos, I first, I last);
```

Каждый из вариантов метода `insert` **выполняет вставку указанных символов перед символом в позиции** `pos`**. Каждый из последующих символов сдвигается на соответствующее число позиций.**

Замена фрагментов строк производится с помощью метода `replace()`**. Его объявления:**

```
//замена символов с pos до pos+n
basic_string& replace(size_type pos, size_type n,
     const basic_string& s);
basic_string& replace(size_type pos, size_type n,
     const basic_string& s, size_type i2, size_type n2);
basic_string& replace(size_type pos, size_type n,
     const C* p, size_type n2);
basic_string& replace(size_type pos, size_type n, const C* p);
basic_string& replace(size_type pos, size_type n,
     size_type n2, C c);
//замена символов с pos до pos2
basic_string& replace(iterator pos, iterator pos2,
     const basic_string& s);
basic_string& replace(iterator pos, iterator pos2,
     const C* p, size_type n);
basic_string& replace(iterator pos, iterator pos2,
     const C* p);
basic_string& replace(iterator pos, iterator pos2,
     size_type n, C c);
template<class I>basic_string& replace(iterator pos,
     iterator pos2, I first, I last);
```

Кроме того, при желании можно удалить подстроку из строки с помощью метода `erase()`**. Объявления:**

```
//удалить все символы с позиции i до позиции i+n
basic_string& erase(size_type i = 0, size_type n = npos);
//удалить один символ в позиции i
```

```
iterator erase(iterator i);
//удалить все символы, с первого до последнего
iterator erase(iterator first, iterator last);
```

Первая версия метода по умолчанию очищает строку.

Векторы

По мере усложнения ваших программ будет возникать потребность в более совершенных структурах хранения данных. Массивы являются неплохим вариантом для хранения данных, но это структура достаточно низкого уровня. Иначе говоря, со многими неудобствами приходится бороться самостоятельно.

Оптимальным решением этой проблемы являются контейнеры стандартной библиотеки. Различные контейнеры служат различным целям. В своей работе вам придется применять большинство из них. И для начала в этом разделе мы познакомим читателей с одним из наиболее применяемых контейнеров – с *вектором*. Вектор обладает свойствами, которые делают его идеальным вариантом для одних случаев и довольно неэффективным решением для других. Какой из типов контейнеров соответствует потребностям программы, решает ее автор.

Векторы схожи с массивами. Вектор хранит набор значений, с каждым из которых можно работать как с отдельной переменной (которая не является частью структуры данных). На деле данные в векторе хранятся в одномерном массиве. По этой причине доступ к любому элементу вектора можно получить, указав его индекс. Такой вид доступа к элементам называется *произвольным* или *прямым доступом*, поскольку позволяет легко обращаться к случайным образом выбранным переменным.

Истории из жизни

Многие программисты игнорируют вспомогательные средства стандартной библиотеки либо просто не знают о них. Когда я (Марк) начал писать свою первую компьютерную игру, то столкнулся с необходимостью хранить объекты игровых единиц. Число игровых единиц невозможно предсказать (по крайней мере, простыми способами), так что мне требовалось их динамическое создание и хранение; а в то время я еще плохо разбирался в этой теме. В результате я потратил уйму времени на исследование этого вопроса. Когда я наткнулся на векторы в стандартной библиотеке, то был приятно удивлен. Другие программисты не только многократно встречались с моей проблемой, но даже разработали ее решение. Вне всякого сомнения, стандартная библиотека позволит вам сэкономить массу бесценного времени, она достойна внимательного изучения.

При добавлении элементов вектор автоматически изменяет свой размер. Эта операция связана с созданием нового, более крупного массива и копированием элементов из исходного массива в новый. Именно эта неэффективная операция является одним из крупных недостатков вектора.

Вставка и удаление элементов из вектора связаны с необходимостью сдвигать элементы с целью заполнения образовавшихся пробелов или расширения диапазона. Такой сдвиг элементов в любом из направлений может существенно влиять на производительность.

Вектор – это класс-шаблон, который хранится в библиотеке `<vector>`. Его объявление похоже на объявление `basic_string`:

```
template <class T, class A = allocator<T>> class std::vector;
```

Распределитель вектора ведет себя во многом так же, как распределитель `basic_string` (то есть в большинстве случаев нет необходимости с ним работать). `T` представляет тип данных, хранимых вектором.

Подобно `basic_string`, класс `vector` определяет стандартные имена посредством `typedef`:

```
typedef T value_type; //тип хранимых данных
typedef A allocator_type; //тип распределителя памяти
typedef typename A::size_type size_type;

//способ перебора элементов
typedef compiler_dependent iterator;
//обратный итератор
typedef std::reverse_iterator<iterator> reverse_iterator;

typedef typename A::pointer pointer; //указатель на элемент
//ссылка на элемент
typedef typename A::reference reference;
```

Эти имена могут пригодиться, но используются преимущественно кодом реализации класса, так что можете не особо о них беспокоиться (достаточно просто помнить, что они означают).

Конструкторы класса `vector` предоставляют массу способов создать объект вектора:

```
vector(const A& = A());
//вектор из n копий val
vector(size_type n, const T& val = T(), const A& a = A());
//вектор, скопированный из входной последовательности
template<class I> vector(I first, I last, const A& a = A());
vector(const vector& a); //вектор, созданный на основе другого вектора
```

Первый конструктор создает указанное число элементов. Каждый элемент инициализируется вызовом конструктора по умолчанию (либо указанным). Второй конструктор действует точно так же, как конст-

руктор basic_string, работающий с последовательностью символов. Он принимает последовательность элементов того типа данных, которые хранит вектор (что вполне очевидно). Третий конструктор используется для создания вектора на основе другого вектора. Приведем несколько примеров:

```
vector<float> fv; //конструктор по умолчанию - пустой вектор
vector<string> sv(10); //вектор из десяти строк
//вектор из 10 целых чисел, каждое имеет значение 3
vector<int> iv(10,3);
string s = "Здравствуй, мир";
vector<string> sv2(s.begin(), s.end());
vector<int> iv2(iv);
```

Кроме того, в классе vector существует деструктор:

```
~vector();
```

Доступ и работа с векторами

Векторы используются аналогично строкам. Большинство интерфейсов совпадают. Вот, например, методы получения итераторов для вектора:

```
iterator begin(); //указывает на первый элемент
iterator end(); //указывает на элемент за последним
reverse_iterator rbegin(); //указывает на последний элемент
reverse_iterator rend(); //указывает на элемент, предшествующий первому
```

Как видите, итераторы работают так же, как и в случае строк. Отметим, что эти итераторы являются итераторами произвольного доступа, поскольку для строк и векторов реализован именно произвольный доступ к элементам.

Чтобы получить доступ к отдельным элементам вектора, можно воспользоваться оператором индекса или методом at(), как и в случае строк. Кроме того, первый и последний элементы вектора доступны посредством методов front() и back(). Объявления этих методов:

```
reference front();
reference back();
```

Вектор предоставляет два способа присваивать значения: с помощью оператора присваивания или метода assign(). Объявления:

```
vector& operator=(const vector& x); //скопировать вектор
//присвоить все значения первого вектора второму
template<class I> void assign (I first, I last);
//присвоить n копий val
void assign(size_type n, const T& val);
```

Методы assign() вектора аналогичны версиям для строк, а потому достаточно прозрачны.

Две операции над векторами известны в качестве *стековых операций*. Речь идет о методах `push_back()` и `pop_back()`. `push_back()` добавляет значение в конец вектора, а `pop_back()` удаляет последний элемент из вектора. Объявления:

```
void push_back(const T& x);
void pop_back();
```

Еще три операции известны в качестве операций над *списком*. Речь идет о методах `insert()`, `erase()` и `clear()`. `insert()` вставляет набор значений в определенной позиции вектора, `erase()` удаляет последовательность элементов из вектора, `clear()` удаляет все элементы. Вот объявления этих методов:

```
iterator insert(iterator pos, const T&); //вставить x в позиции pos
//вставить n значений x  в позиции pos
void insert(iterator pos, size_type n, const T& x);
//вставить последовательность в позиции pos
template <class I> void insert(iterator pos, I first, I last);
iterator erase(iterator pos); //удалить элемент в позиции  pos
//удалить последовательность элементов
iterator erase(iterator first, iterator last);
void clear(); //удалить все элементы
```

Класс вектора также реализует методы `size()`, `empty()` и `max_size()`, которые ведут себя идентично одноименным методам класса `basic_string`.

И наконец, подобно классу `string`, класс `vector` реализует операторы сравнения (`==`, `!=`, `<`, `>`, `<=`, `>=`).

Игра «Таинственный магазин»

Блуждая по диким просторам долины Вектор, ты наткнулся на небольшой городок. Он совсем не похож на другие городки. Такое ощущение, что у всех местных жителей все есть. Припасов хватает всем. Пытаясь разрешить эту загадку, ты зашел в крохотный магазинчик. Похоже, именно этот магазин является рогом изобилия. Изучив собственные припасы, ты понял, что не мешало бы их пополнить, и потому без лишних раздумий вошел внутрь магазина. Скомпилируйте следующий код и посмотрите, что получится.

```
//9.1 - Таинственный магазин - Марк Ли - Premier Press
#include <iostream>
#include <vector>
#include <string>

using namespace std;

#define MAX(a,b) a<b ? b: a //общая макроподстановка
```

```
struct Item //используется для хранения информации о товаре магазина.
{
     string name;
     int price;
};

class Store //используется для работы магазина
{
     vector<Item> inventory;
     vector<Item> forSale;
     int money;
public:
     Store(Item* itemList, int n);

     ~Store() {}

     string BuyItem(int item);

     string viewInventory();

     string ListItems();

     int getMoney()
     {
          return MAX(money,0);
     }

};
Store::Store(Item* itemList, int n)
{

     for(int i = 0; i < n; i++)
          forSale.push_back(itemList[i]);
     money = 20;
}

string Store::BuyItem(int item)
{
     money -= forSale[item-1].price;
     if (money < 0)
          return "\nИзвините, у вас недостаточно денег.\n\n";
     inventory.push_back(forSale[item-1]);
     return "Вы купили " + forSale[item-1].name + '\n';

}

string Store::ListItems()
{
     string s;
     for(int i = 0; i < forSale.size(); i++)
     {
          s += "[";
          s += i + 49;
          s+="]Купить ";
          s+= forSale[i].name;
          s+= " ($";
```

```
            s+= forSale[i].price + 48;
            s+= ")\n";
        }
        return s;
}

string Store::viewInventory()
{
        string s;
        for(int i = 0; i < inventory.size(); i++)
            s += inventory[i].name + '\n';
        return s + '\n';
}

int main(void)
{
        int input;
        Item f[3];
        f[0].name = "Clown";
        f[0].price = 2;
        f[1].name = "Cracker Jack";
        f[1].price = 6;
        f[2].name = "Camel";
        f[2].price = 9;
        Store s = Store(f,3);
        while(true)
        {
            do {
                cout << "Добро пожаловать в магазин.\n"
                    << "У вас " << s.getMoney()
                    << " долларов.\n"
                    << "\nЧто желаете сделать?\n"
                    << s.ListItems()
                    << "[4]Просмотреть каталог припасов\n"
                    << "[5]Уйти\n";
                cin >> input;
            }while(input<1||input>5);
            switch(input)
            {
                case 4:
                    cout << s.viewInventory();
                    break;
                case 5:
                    goto END;
                default:
                    cout << s.BuyItem(input);
            }
        }
END:
        cout << "Счастливо!";
        return 0;
}
```

Резюме

В этой главе мы изучили ряд важных тем, включая шаблоны, строки и векторы. Считайте эту главу своим первым серьезным знакомством со стандартной библиотекой C++, мощным инструментом, которому с течением времени вы будете находить все больше применений. Необязательно помнить наизусть все методы класса `basic_string` или все детали работы с векторами, но очень важно знать, где можно найти нужную информацию.

Задания

1. Создайте вектор, содержащий набор векторов, каждый из которых хранит набор целых чисел.
2. Создайте класс-шаблон `store`, позволяющий хранить массив элементов `T` (`T` является параметром шаблона).
3. Создайте итератор `random_iterator`, который использует другой итератор для произвольного доступа к элементам контейнера.
4. Назовите три места (не считая этой книги), где можно быстро получить информацию о структурах стандартной библиотеки.

10

Потоки и файлы

Потоки и файлы входят в число самых сложных тем языка C++. В этой главе мы постараемся безболезненно преподать читателям соответствующие уроки. И хотя на эти темы можно говорить до бесконечности, к концу этой главы читатели, вне всякого сомнения, смогут решать насущные задачи при помощи потоков.

В этой главе:

- Терминология потоков ввода-вывода
- Манипуляторы
- Бинарные и текстовые файлы
- Деление данных на битовые поля
- Магия битовых сдвигов
- Создание программы шифрования

Терминология ввода-вывода

Основой для понимания этой главы является понятие ввода-вывода. Ввод-вывод означает отправку (вывод) и получение (ввод) данных от различных аппаратных устройств, таких как жесткий диск, модем и клавиатура.

В C++ операции ввода-вывода не являются частью языка, но они доступны в составе стандартной библиотеки. Несмотря на это, C++ оставляет применение потоков на совести программистов. Это означает, что поддержка ввода-вывода в C++ реализована на достаточно низком уровне, и для применения системы в реальных приложениях приходится подстраивать ее под конкретные нужды.

Чтобы изучить подсистему ввода-вывода, необходимо прежде всего определиться с терминологией. Вот определения некоторых терминов, используемых на протяжении главы:

- **Объект потока.** Действует в качестве источника и адресата байтов. Объект потока работает с упорядоченными последовательностями байтов. Эти последовательности могут являться экранами, файлами и любыми другими сущностями, которые могут представляться в виде байтов. Классы для работы с потоками расположены в нескольких библиотечных файлах: `<fstream>`, `<iomanip>`, `<ios>`, `<iosfwd>`, `<iostream>`, `<istream>`, `<ostream>`, `<sstream>`, `<streambuf>` и `<strstream>`.
- **Манипулятор.** Манипулирует данными потока определенным образом. Скажем, манипулятор может изменить регистр всех символов или преобразовать числа из десятичной системы в шестнадцатеричную.
- **Вставка.** Запись байтов в поток.
- **Извлечение.** Чтение байтов из потока.

Разбираемся с файлами заголовков

В списке определений предыдущего раздела мы перечислили несколько заголовочных файлов. Каждый из них содержит сегмент архитектуры потоков C++. Эти файлы совместно работают на осуществление поддержки ввода-вывода для языка C++. Каждый из них отвечает за решение определенных задач, что в результате дает языку C++ самую богатую функциональность ввода-вывода в сравнении с другими языками программирования. Назначение файлов заголовков следующее:

- `<fstream>`. Содержит определения нескольких классов-шаблонов, поддерживающих работу кода файла `iostream` с последовательностями, хранимыми во внешних файлах.
- `<iomanip>`. Содержит несколько манипуляторов, работающих с единственным аргументом.
- `<ios>`. Содержит большинство форматных манипуляторов, необходимых для работы классов из файла `iostream`.
- `<iosfwd>`. Содержит предварительные объявления классов файла `iostream`.
- `<iostream>`. Содержит объявления стандартных глобальных потоковых объектов, таких как `cin` и `cout`.
- `<istream>`. Содержит методы извлечения данных из потоков и класс-шаблон `basic_istream`. Иными словами, `<istream>` отвечает за ввод.
- `<ostream>`. Содержит методы вставки, предназначенные для вывода последовательностей байтов, а также класс-шаблон `basic_ostream`. По существу, `<ostream>` отвечает за вывод.
- `<sstream>`. Содержит классы-шаблоны, позволяющие классам `iostream` работать с последовательностями, хранимыми в символьных массивах.

- `<streambuf>`. Определяет класс-шаблон `basic_streambuf`.
- `<strstream>`. Определяет классы, позволяющие классам `iostream` работать с последовательностями, хранимыми в символьных массивах в стиле C.

За исключением класса `ios_base`, каждый класс-шаблон имеет уточненную версию для работы с символами. На рис. 10.1 приведены наиболее востребованные классы и их уточненные версии. Однако список неполон, поскольку отдельные классы потоков выходят за пределы настоящей книги.

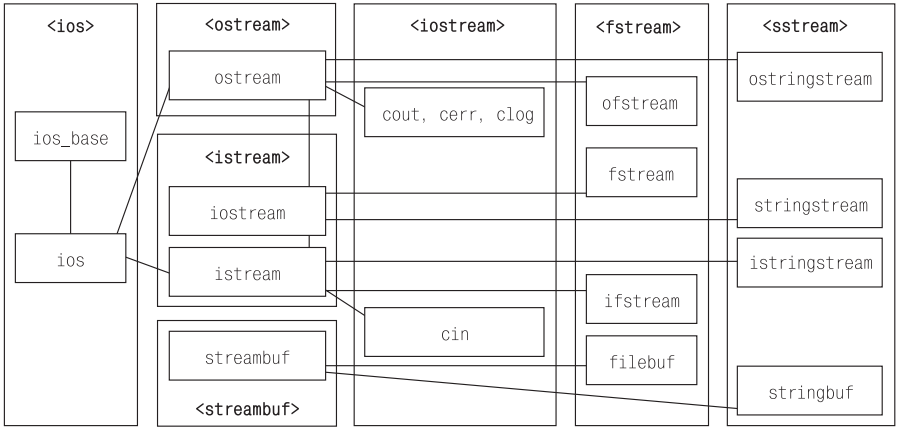


Рис. 10.1. *Классы отдельных файлов заголовков и их классы-предки. Жирным шрифтом выделены имена файлов заголовков, прочие имена представляют классы. Связующие линии отражают структуру наследования классов*

Класс ios_base

Класс `ios_base` реализует хранилище байтов, общее для всех потоков и методов, не зависящих от параметров шаблонов. Байты ввода и вывода содержатся в хранилище, пока не будут переданы по назначению. Основу управления потоками составляют различные типы данных класса `ios_base`.

В следующем списке описано назначение некоторых из таких типов:

- `event`. Используйте информацию о событии из типа данных `event` для хранения обратных вызовов. *Обратные вызовы* – это указатели на методы, зарегистрированные и выполняемые операционной системой при возникновении определенных событий.
- `fmtflags`. Используйте флаги формата типа данных `fmtflags` для указания форматных параметров потока.
- `iostate`. Используйте информацию состояния ввода-вывода из типа данных `iostate` для наблюдения за состоянием операций ввода-вы-

вода. Информацией `iostate` можно пользоваться для проверки целостности потоков.

- `openmode`. Используйте для указания режима доступа (чтения/записи) для потоков.
- `seekdir`. Используйте направление поиска для указания способа поиска в данных потоков и сохраняйте этот параметр в типе-псевдониме `seekdir`.

Кроме того, доступен ряд методов класса `ios_base` и порожденных от него классов (всех прочих классов потоков). Вот эти методы, в алфавитном порядке:

- `flags()`. Позволяет устанавливать или получать значения флагов форматов.
- `getloc()`. Возвращает копию внутреннего локального объекта, инкапсулирующего параметры потока.
- `imbue()`. Сохраняет локальный объект, инкапсулирующий параметры потока, и возвращает предыдущую копию объекта.
- `ios_base()`. Конструктор.
- `iword()`. Возвращает ссылку на указанный элемент расширяемого массива типа `long`.
- `operator =()` (перегруженный оператор присваивания). Копирует содержимое объекта в новый потоковый объект – включая информацию форматирования и расширяемые массивы.
- `precision()`. Определяет точность отображения в количестве цифр после десятичной запятой.
- `pword()`. Возвращает ссылку на конкретный элемент массива `void*`.
- `register_callback()`. Сохраняет параметры обратного вызова для последующего использования.
- `setf()`. Устанавливает новые флаги для потока и возвращает параметры существующих.
- `unsetf()`. Сброс всех флагов.
- `width()`. Позволяет устанавливать или получать ширину поля потока.
- `xalloc()`. Возвращает уникальную статическую переменную для использования в качестве индекса массива с функциями `iword()` или `pword()`.

О классе `ios_base` можно написать еще очень много. Однако чтобы читатели не потеряли интерес, а книга не стала чрезмерно толстой, мы решили сфокусироваться на самых востребованных применениях потоков. Странные люди, подобные нам, занимаются изучением потоков постоянно, и таким людям мы рекомендуем обратиться к разделу Library (библиотека) веб-сайта MSDN по адресу *http://www.msdn.microsoft.com*.

Знакомьтесь, файловые потоки

Для повышения функциональности игр зачастую требуется хранить миллионы элементов информации – скажем, информацию карт или сохраненные игры. Хранение таких данных организуется при помощи специализированных классов ввода-вывода: `ofstream`, `ifstream` и `fstream`, представленных соответственно файлами заголовков `ofstream`, `ifstream` и `fstream`.

Открываем файлы

Как правило, первое, что необходимо сделать с объектом файлового потока, порожденного от класса `ofstream`, `ifstream` или `fstream`, это связать его с рабочим файлом. Этой цели служит метод `open()` или конструктор любого из классов.

Прототип метода `open()` выглядит так:

```
void open (const char * filename, openmode mode);
```

filename – это строка символов, представляющая имя файла, а *mode* – это комбинация флагов, перечисленных в табл. 10.1.

Флаги объединяются поразрядным оператором «или» (`|`). Предположим, необходимо открыть файл `"level1.map"` на вывод в бинарном режиме. Вызов функции `open` может выглядеть следующим образом:

```
ofstream file;
file.open ("level1.map", ios::out | ios::binary);
```

Каждый из файловых классов имеет метод `open()`; однако каждый поток имеет ряд установленных по умолчанию флагов, определяющих его назначение. Значения по умолчанию, перечисленные в табл. 10.2, имеют силу только в том случае, если метод вызывается без флагов.

Кроме того, можно открыть файл, указав его имя при вызове конструктора. Конструкторы этих классов имеют те же аргументы, что и метод `open()`.

Таблица 10.1. Флаги открытия файла

Флаг	Назначение
`ios::in`	Открыть файл в режиме чтения
`ios::out`	Открыть файл в режиме записи
`ios::ate`	Начать работу с конца файла
`ios::app`	Добавлять вывод в конец файла
`ios::trunc`	Перезаписывать файл, если существует
`ios::binary`	Производить доступ к файлу в бинарном режиме

Таблица 10.2. Значения по умолчанию для флагов файловых классов

Класс	Флаги
ofstream	ios::out | ios::trunc
ifstream	ios::in
fstream	ios::in | ios::out

Например, если используется класс ifstream, прототип конструктора выглядит следующим образом:

```
ifstream::ifstream(const char * filename, openmode mode);
```

Здесь параметры *filename* и *mode* имеют те же значения, что и для метода open().

Корректность открытия файла можно проверить вызовом метода is_open():

```
bool is_open()
```

is_open() возвращает значение true, если файл был успешно открыт, и false в противном случае.

Самые распространенные причины того, что не удается корректно открыть файл:

- Неправильное имя файла – файл с указанным именем не существует.
- Поврежденный файл – к примеру, расположенный в сбойном секторе жесткого диска.
- Другая программа имеет монопольный доступ к файлу: это означает, что все прочие программы временно не могут работать с файлом, пока программа с монопольными правами не вернет их системе.
- Файл находится под защитой Windows. Windows может автоматически ограничивать доступ к файлам определенных типов либо расположенных в определенных папках. В этом случае свяжитесь с системным администратором.

Закрываем файлы

Чтобы файл стал доступен другим программам, его следует закрыть. Кроме того, один объект может единовременно работать только с одним файлом. Закрытие файлы выполняется посредством метода close. Метод существует в каждом из трех классов и объявлен следующим образом:

```
void close();
```

При уничтожении объекта потока деструктор автоматически закрывает связанный с этим объектом файл.

Работаем с текстовыми файлами

Файловый ввод и вывод ничем не отличается от экранного, поскольку классы `cin` и `ifstream`, как можно видеть из схемы на рис. 10.1, являются потомками класса `istream`. Аналогичным образом `cout` и `ofstream` являются потомками класса `ostream`. В обоих классах используется перегруженный оператор вставки в поток, `<<`.

Следующий код поясняет сказанное:

```
//10.1 - Вывод текста - Дирк Хенкеманс - Premier Press
#include <fstream>
using namespace std;

int main( void )
{
     //флаги по умолчанию: ios::out | ios::trunc
     ofstream dragons("dragons.txt");

     if(dragons.is_open())
     {
          dragons << "Медный дракон" << endl
               << "Бронзовый дракон"<< endl
               << "Серебряный дракон" << endl
               << "Золотой дракон" <<endl;
     }

     dragons.close();

     return 0;
}
```

Вывод:

```
Медный дракон
Бронзовый дракон
Серебряный дракон
Золотой дракон
```

Обычно файл `"dragons.txt"` создается в папке проекта. Если его там нет, воспользуйтесь меню Windows: Пуск, Найти, Файлы и папки. Откроется окно, озаглавленное примерно так: «Найти: Все файлы». Наберите имя файла и выберите диск, на котором он хранится. Нажмите на кнопку «Найти», и Windows найдет файл. (Не забывайте подставлять свои имена файлов.)

Если выполнить программу дважды, можно заметить, что текстовый файл содержит одну и ту же информацию. Это происходит потому, что для класса `ofstream` по умолчанию установлен режим `ios::trunc`, при котором существующие файлы перезаписываются.

Несмотря на возможность чтения данных из файла при помощи перегруженного оператора сдвига (`>>`), такой способ не позволяет определить, что достигнут конец файла. Простейший способ извлечения данных из файла – построчное чтение посредством метода `istream::getline()`. Это

один из самых нужных методов, поскольку большая часть данных сохраняется построчно, так что, прочитав строку, ее можно разобрать на поля, из которых она состоит.

Метод `istream::getline()` существует в трех разновидностях:

```
istream& getline(char* pch, int nCount, char delim = '\n');
istream& getline(unsigned char* puch, int nCount, char delim = '\n');
istream& getline(signed char* psch, int nCount, char delim = '\n');
```

Параметры имеют следующие значения:

- *pch*, *puch*, *psch*. Указатель на массив символов. Помните, что метод `getline` перезаписывает хранимые в массиве данные.
- *nCount*. Максимальное число символов, которое может быть сохранено методом `getline`, включая завершающий пустой символ.
- *delim*. Символ-разделитель, ограничивающий чтение дных методом `getline()`.

И наконец, читать данные следует лишь до конца файла, но как определить, что конец файла достигнут? С помощью метода `ios::eof()`:

```
int eof();
```

Метод `eof()` возвращает ненулевое целое значение, если достигнут конец файла.

Учитывая все эти обстоятельства, мы можем заняться чтением данных из файла `"dragon.txt"`. Вот пример реализации кода чтения:

```
//10.2 - Чтение драконов - Дирк Хенкеманс
//Premier Press
#include <fstream>
#include <iostream>
using namespace std;

int main( void )
{
     //флаги по умолчанию: ios::in
     ifstream dragons("dragons.txt");

     char buffer[50];

     if(!dragons.is_open())
     {
          cout<<"Ошибка, невозможно открыть файл";
          //ненулевой код завершения (обычно) означает, что возникла ошибка
          exit(1);
     }

     while(!dragons.eof())
     {
          dragons.getline(buffer, 49);
          cout<< buffer <<endl;
     }
```

```
        dragons.close();

        return 0;
    }
```

Вывод:

```
Медный дракон
Бронзовый дракон
Серебряный дракон
Золотой дракон
```

Метод `getline()` удаляет разделители (в нашем случае – переносы строк), поэтому программа самостоятельно добавляет переносы после каждой прочитанной строки.

Вот вы и прочитали свой первый файл!

Проверка потоков

Как определить, что поток выполнил порученное ему задание? Интерфейс объектов потоков предоставляет ряд методов, позволяющих определять успешность завершения заданий. Некоторые из методов проверки потоков приведены в табл. 10.3.

Для сброса включенных методов проверки вызовите метод `ios::clear()` без параметров.

Таблица 10.3. Методы проверки

Метод	Описание
`bad()`	Возвращает `true`, если при попытке прочитать или записать данные в файл произошла серьезная ошибка. Такая ошибка может, например, возникнуть при попытке записи на устройство, на котором отсутствует свободное пространство.
`eof()`	Возвращает `true`, если при работе с потоком достигнут конец файла. Работает только для чтения из файлов.
`fail()`	Возвращает `true`, если возникла ошибка, связанная с форматом. Например если поток попытался прочитать целое число, а вместо этого обнаружил букву.
`good()`	Общий метод, возвращающий `false` в случае, когда один из методов – `bad`, `eof` и `fail` – возвращает `true`.

Работаем с бинарными потоками

Несмотря на удобства работы с текстом, этот формат далеко не всегда является эффективным. При посылке данных через модем на скорости соединения 28 800 бит/с имеет смысл посылать минимальные объемы данных. В игре вроде «StarCraft» от Blizzard можно заранее предпи-

сать передачу определенных типов данных. Это позволяет выйти за ограничение символьного типа данных.

Решением проблемы являются *бинарные потоки*, позволяющие посылать различные типы данных. Программе *необходимо* знать порядок и вид типов данных, чтобы произвести чтение данных. Текстовый и бинарный потоки показаны на рис. 10.2.

Для корректной интерпретации программе необходимо знать, с какой структурой связаны байты бинарного потока. Это означает, что для расшифровки данных требуется программный алгоритм (различный для различных типов данных), а в текстовом потоке каждый байт считается отдельным символом. Обратите внимание, на рис. 10.2 верхний поток данных связан с текстовым файлом, и каждый байт представляет отдельный символ. Байты данных второго потока составляют более сложные структуры: программа должна знать, какие именно, чтобы иметь возможность правильно их интерпретировать.

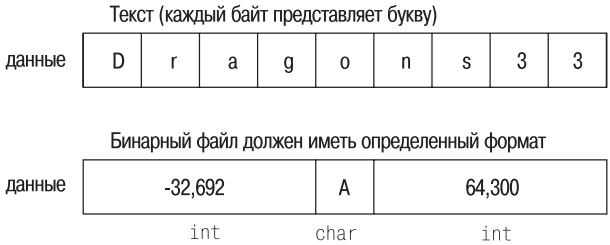


***Рис. 10.2.** Форматы различных файлов потоков и классов, которые они содержат*

Потоковые указатели get и put

Индексирование памяти существует для нашего с вами удобства, но кроме того – и это важно – для обеспечения произвольного доступа. Произвольный доступ позволяет обращаться к частям файлов и потоков в любом порядке. Например, файл можно читать от конца к началу почти так же легко, как и от начала к концу. Позиции чтения и записи определяются двумя указателями, которые известны в качестве *потоковых указателей*. Речь идет об указателях `get` и `put`. Указатель `get` определяет текущую позицию чтения, а указатель `put` – текущую позицию записи.

Каждый поток ввода-вывода имеет по меньшей мере один потоковый указатель, зависящий от метода потока. Что касается наиболее востребованных классов:

- `ifstream`. Подобно `istream`, этот класс обладает указателем `get`.
- `ofstream`. Подобно `ostream`, этот класс обладает указателем `put`.
- `fstream`. Являясь потомком классов `istream` и `ostream`, этот класс обладает обоими указателями.

Интерфейс потоковых указателей

Для применения потоковых указателей необходимо создать интерфейс, включающий следующие методы:

- `istream::tellg()`. Возвращает число байт, на которое указатель `get` отстоит от начала файла (не адрес, на который указывает `get`). Тип данных `streampos` является, по сути дела, псевдонимом целочисленного типа. Прототип метода:

```
streampos tellg();
```

- `ostream::tellp()`. Возвращает число байт, на которое указатель `put` отстоит от начала файла. Прототип метода:

```
streampos tellp();
```

- `istream::seekg()`. Устанавливает положение указателя `get` в потоке. Прототип метода:

```
seekg (pos_type position);
```

- `ostream::seekp()`. Устанавливает положение указателя `put` в потоке. Прототип метода:

```
seekp (pos_type position);
```

- `seekg()` и `seekp()`. Устанавливают направление смещения указателя и величину смещения. Прототипы методов:

```
seekg ( off_type offset, seekdir direction );
seekp ( off_type offset, seekdir direction );
```

Возможные значения для направления смещения (второй параметр) перечислены в табл. 10.4.

Таблица 10.4. Направления смещений

Направление	Описание
`ios::beg`	Смещение отсчитывается от начала потока
`ios::cur`	Смещение отсчитывается от текущего положения указателя
`ios::end`	Смещение отсчитывается от конца потока

В следующем примере описанные методы используются для определения размера файла:

```
//10.3 - Размер бинарного файла - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <fstream>
using namespace std;

int main( void )
{
     int n1, n2;
     ifstream dragons("dragons.txt", ios::in|ios::binary);
     n1 = dragons.tellg();
```

```
    dragons.seekg (0, ios::end);
    n2 = dragons.tellg();
    cout<<"Размер файла dragons.txt: " << n2 - n1 << endl;

    return 0;
}
```

Вывод:

```
Размер файла dragons.txt: 58
```

Бинарные потоки, чтение и запись

Файловые потоки знакомят нас с двумя новыми методами, предназначенными специально для последовательного ввода и вывода данных: `ostream::write()` и `istream::read()`. Прототипы этих методов выглядят следующим образом:

```
write (char * buffer, streamsize size );
read( char * buffer, streamsize size );
```

`write()` помещает указанное число байт (`size`) после указателя `put`, таким образом записывая их в файл.

`read()` читает указанное число байт от указателя `get` и сохраняет их в параметре-указателе на буфер.

Проиллюстрируем использование этих методов на примере следующей программы, которая копирует файл `"dragons.txt"` в файл с другим именем, `"dragons2.txt"`.

```
//10.4 - Копирование файлов - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <fstream>
using namespace std;

int main( void )
{
    char buffer;
    int index = 0;

    //имена файлов
    const char filename1[] = "dragons.txt";
    const char filename2[] = "dragons2.txt";

    //открываем файлы
    fstream file1(filename1, ios::in);
    fstream file2(filename2, ios::out);

    //указатель в начало файла
    file1.seekg(0, ios::beg);
    file2.seekp(0, ios::beg);

    //читаем первый символ
    file1.read(&buffer, 1);

    //записываем оставшиеся символы
    while(file1.good() && file2.good())
```

```
        {
            file2.write(&buffer, 1);
            index++;
            file1.seekp(index);
            file2.seekg(index);
            file1.read(&buffer, 1);
        }
        //закрываем файлы
        file1.close();
        file2.close();
        return 0;
    }
```

Полученный файл `"dragons2.txt"` по содержанию не отличается от `"dragons.txt"`.

Работа с манипуляторами

Манипуляторы используются для обработки данных в потоке. Они могут, например, изменять регистр символов или формат отображения чисел. Большинство манипуляторов объявлены в заголовочном файле `ios`. В этом разделе мы будем использовать в основном простые манипуляторы из файла `ios`.

И хотя рассказать обо всех манипуляторах мы не сможем, некоторые из наиболее популярных описаны в табл. 10.5.

Таблица 10.5. Манипуляторы из заголовочного файла <ios>

Флаг	Назначение
`dec()`	Отображает числа в десятичной системе счисления
`hex()`	Отображает числа в шестнадцатеричной системе счисления
`oct()`	Отображает числа в восьмеричной системе счисления
`fixed()`	Вставляет действительные значения
`scientific()`	Вставляет действительные значения в экспоненциальной записи
`internal()`	По необходимости выполняет отбивку поля, внедряя пробелы
`left()`	Выравнивает символы по левому краю, добавляя пробелы справа
`right()`	Выравнивает символы по правому краю, добавляя пробелы слева
`boolalpha()`	Символьное представление `true` и `false`
`noboolalph()`	Отменяет `boolalpha()`
`showbase()`	Отражает основание системы счисления, предваряя восьмеричные и шестнадцатеричные числа префиксами `0` и `0x` соответственно
`noshowbase()`	Отменяет `showbase()`

Флаг	Назначение
showpoint()	Отображает десятичную запятую, даже если отсутствует дробная часть
noshowpoint()	Отменяет showpoint()
showpos()	Предваряет неотрицательные числа символом +
noshowpos()	Отменяет showpos()
skipws()	Пропускает пробелы
noskipws()	Отменяет skipws()
unitbuf()	Сброс буфера после каждой операции вставки в поток
nounitbuf()	Отменяет unitbuf()
uppercase()	Вставляет прописные эквиваленты строчных букв
nouppercase()	Отменяет uppercase()

Описанные манипуляторы можно использовать в любом потоке – с помощью метода ios::setf(), которому в качестве параметра передается соответствующий флаг. Следующая программа ясно демонстрирует эту процедуру для объекта cout.

```
//10.5 - Манипуляторы - Дирк Хенкеманс - Premier Press
#include <iostream>
#include <ios>
using namespace std;

int main( void )
{
     cout << "Стандартные true и false \n"
          << true << " " << false << endl;

     cout.setf(ios::boolalpha);
     cout<<"\nдля ios::boolalpha:\n"
          <<true << " " << false << endl;
     cout <<"\n140 в шестнадцатеричной\n";
     cout.setf(ios::hex, ios::basefield);
     cout.setf(ios::showbase);
     cout << 140 << endl;

     float f[2] = {1.0f, 775.374f};

     cout<< "\nСтандартная численная формула\n"
          << f[0] << endl
          << f[1] << endl;

     cout.setf(ios::showpos);
     cout<< "\nи для showpos\n"
          << f[0] << endl
          << f[1] << endl;

     //предписывает использовать точность в 6 знакомест
     cout.setf(ios::fixed);
     cout<< "\nс точностью 6\n"
```

```
            << f[0] << endl
            << f[1] << endl;
        return 0;
}
```

Битовые поля

Стандартные типы данных являются встроенными в C++; но иногда бывает более эффективно разделить стандартный тип данных на несколько *битовых полей* и обрабатывать их значения по отдельности. Предположим, необходимо разделить беззнаковый символьный тип на битовые поля. Беззнаковый символ состоит из восьми бит. Это означает, что все битовые поля в сумме должны занимать 8 бит.

Битовые поля объявляются при помощи двоеточий, которые разделяют объявления и числа, определяющие количество бит для каждого поля. Битовые поля должны объявляться в пределах структуры следующим образом:

```
struct shortBits
{
     short member1 : 2;
     short member2 : 7;
     short member3 : 7;
};
```

Поле `member1` может хранить только три значения: `1 + 2`.

Поля `member2` и `member3` могут хранить `127` значений: `1 + 2 + 4 + 8 + 16 + 32 + 64`.

На рис. 10.3 представлено графическое отображение области памяти структуры `shortBits`.

***Рис. 10.3.** На этом рисунке область памяти типа данных short поделена на три поля*

Занимательный поразрядный сдвиг

Операторы << (сдвиг влево) и >> (сдвиг вправо) являются операторами поразрядного сдвига. При поразрядном сдвиге биты типа данных сдвигаются на один бит. Например, если сдвинуть значение типа `char` на два бита влево, все биты значения сдвигаются на два бита влево. Два крайних бита слева отсекаются, а справа добавляются два сброшенных бита (рис. 10.4).

Сдвиг битов на одну позицию вправо эквивалентен умножению на 2, а на одну позицию влево – делению на 2 с отбрасыванием остатка.

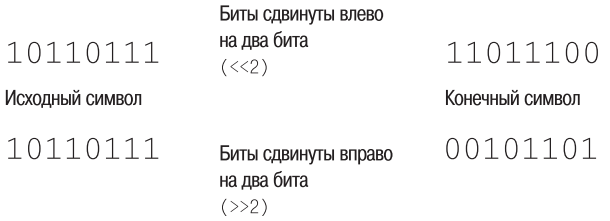


***Рис. 10.4.** Исходный символ (слева) сдвигается на два места; результирующий символ отображен справа*

Пишем программу шифрования

Агент, цель вашего задания, если вы его примете, – разработать высокоэффективную программу шифрования для секретных операций разведслужбы Великобритании. Программа должна работать быстро и эффективно, упрощая жизнь агентов разведслужбы на территории врага. Если вас поймают, мы, авторы, будем отрицать всякую причастность к вашему заданию.

Удачи, агент!

Алгоритм шифрования, который мы используем для этого задания, весьма прост. Программа должна менять местами первые четыре бита и последние четыре бита каждого байта. Эту операцию можно выполнить с помощью оператора поразрядного сдвига и деления на 128 с отбрасыванием дробной части.

Попытайтесь решить задачу самостоятельно, а если столкнетесь с затруднениями, сверьтесь с решением, изложенным ниже.

```
//10.6 - Программа шифрования - Дирк Хенкеманс
//Premier Press
//Программа шифрования текстовых данных
#include <iostream>
#include <fstream>
using namespace std;

class Encryption
{
     fstream file1;//исходный файл
     fstream file2;//конечный файл

     public:
     Encryption::Encryption(char* filename1, char* filename2)
     {
          file1.open(filename1, ios::in | ios::out | ios::binary);
          file2.open(filename2, ios::out|ios::binary);
     }
```

```
        //шифрует файл
        void Encrypt(void)
        {
             char currentByte;
             bool currentBit;

             int index = 0;

             //указатель в начало файла
             file1.seekg (0, ios::beg);
             file2.seekp (0, ios::beg);

             //читаем первое значение
             file1.read(&currentByte, 1);
             while(file1.good())
             {
                  //цикл на четыре бита
                  for(int c = 0; c < 4; c++)
                  {
                       //является ли первый бит единицей?
                       currentBit = (int)((unsigned char)currentByte / 128);
                       //сдвигаем байт
                       currentByte <<= 1;

                       //если первый бит был единицей, добавляем его в конец
                       if(currentBit)
                       {
                            currentByte += 1;
                       }
                  }

                  //записываем символ
                  file2.write(&currentByte, 1);

                  //приращение указателя
                  file1.seekg (++index);
                  file2.seekp (index);

                  //чтение следующего значения
                  file1.read(&currentByte, 1);
             }
        }

        //закрываем оба файла
        void close(void)
        {
             file1.close();
             file2.close();
        }
};

int main( void )
{
        cout<< "Вас приветствует программа шифрования службы  S.A.S.";
        Encryption delta("dragons.txt", "output1.txt");
        delta.Encrypt();
```

```
    delta.close();

    Encryption gamma("output1.txt", "output2.txt");
    gamma.Encrypt();
    delta.close();
    return 0;
}
```

Не пытайтесь обрабатывать этой программой шифрования большие или нужные файлы. Ни авторы, ни издательство Premier Press не несут ответственности за утрату или изменение важной информации, равно как и не гарантируют безопасности программы.

После первого прогона программы в целях шифрования файла полученный результат (при просмотре в текстовом редакторе вроде Notepad или DOS Edit) будет выглядеть довольно странно и, скорее всего, расположится в одну строку. Например, (кроме шуток) результат может выглядеть так:

```
@##%$%^&*7gfh@#$#4
```

При повторном прогоне программы файл должен быть расшифрован и примет свой изначальный вид.

Резюме

Глава началась с изучения иерархии и определений потоков. Затем последовали файловые потоки и текстовые файлы в файловом режиме. После этого знания читателей обогатились копированием файлов при помощи потоков, а также информацией об указателях `get` и `put`. Используя манипуляторы из библиотеки `<ios>`, мы работали с потоком `cout` и добились некоторых замечательных эффектов. И наконец, мы разработали программу шифрования.

Задания

1. Создайте программу, записывающую приведенные ниже строки текста в файл «Question1.txt».

   ```
   Программировать весело.
   Мне нравится программировать.
   ```

2. Каким образом можно определить, что достигнут конец файла? Каким образом можно определить, что при работе с файлом возникла ошибка? Каким образом можно определить, что достигнут конец файла или возникла ошибка?

3. Каков результат поразрядного сдвига влево (`<<3`) для символа A? Каков результат поразрядного сдвига вправо (`>>2`) для символа A?

4. Объясните, почему для расшифровки исходного файла требуется повторно выполнить программу шифрования.

11

Ошибки и обработка исключений

В этой главе речь пойдет о том, как сделать программы надежными и защищенными от сбоев. Вы узнаете, как обеспечивать и проверять истинность определенных условий и создавать ограничители, не позволяющие выходить за пределы массива. В двух словах, вы приобретете навыки создания очень стабильных программ, способных справиться с непредвиденными ситуациями. В этой главе:

- Применение утверждений
- Обработка исключений
- Перехват всех исключений

Доказательство утверждений

Отдельные действия в программировании могут быть опасными. Они способны искажать данные в памяти, приводить к зависаниям компьютера и служить источником многих других неприятностей. В обычных условиях автор программы знает, какие предпосылки должны быть истинными для успешного выполнения определенной операции. Он знает, какие условия должны выполняться. Если эти условия не выполняются, результатом может стать катастрофа. Обязанностью автора программы является проверка этих условий до выполнения операций. Если условия не выполняются, программу следует немедленно завершить или встретить стихийное бедствие лицом к лицу.

Если отбросить эмоции, то некоторые операции действительно могут служить источником проблем, но эти проблемы редко когда бывают серьезными. Однако чтобы программа могла успешно выполнить свою задачу, имеет смысл проверять истинность условий перед выполнением важных (или потенциально небезопасных) операций. Если условие не выполняется в процессе тестирования программы, программу можно завершить, получив при этом сообщение, отражающее смысл происходящего и координаты причины.

В C++ существует специальная макроподстановка, известная как *утверждение*, которая завершает программу и отображает сообщение, связанное с указанным условием. *Утверждение* – это макроподстановка, которая проверяет истинность условия и прерывает работу программы, если условие ложно. Стандартная версия этой макроподстановки определена в файлах заголовков <cassert> и <assert.h>. Прототип утверждения выглядит следующим образом:

```
void assert(int expression);
```

Выражение (*expression*) является целым числом, но интерпретируется в качестве логического выражения (например, 5 < 3). Таким образом, проверка условия может выглядеть так:

```
assert (y > 0);
```

Данное утверждение игнорирует случаи, когда y больше 0, но останавливает программу и отображает сообщение об ошибке, если y не больше 0. Чтобы увидеть это сообщение об ошибке, скомпилируйте и выполните следующую программу:

```
//11.1 - Ложное утверждение - Марк Ли - Premier Press
#include <cassert>

int main(void)
{
     assert(5>6); //всегда ложно
     return 0;
}
```

Данное утверждение ложно при любых обстоятельствах, поскольку число 5 не может быть больше 6. Что происходит в случае ложного утверждения? Прежде всего, отображается сообщение:

```
Assertion failed: 5>6, file C:\Projects\Temp\Test13\Test13.cpp, line 5
```

На моем компьютере (Windows 2000 Professional и Visual Studio 6.0) открывается окно с сообщением о том, что программа аварийно завершила работу (рис. 11.1).

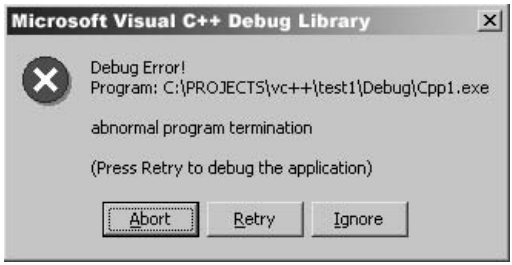


***Рис. 11.1.** Ложные утверждения приводят к появлению подобных окон сообщений. Сообщение содержит имя исполняемого файла и имя каталога*

Макроподстановка assert принудительно завершает программу вызовом функции abort(). Функция abort() объявлена в файлах <cstdlib> и <stdlib.h>. Она работает без аргументов и не возвращает значений – просто завершает работу программы. Именно эта функция стала причиной возникновения окна сообщений (рис. 11.1). Эта же функция выводит сообщение об аварийном завершении программы.

Если данный вариант вас не устраивает (скажем, необходимо продолжить выполнение программы в случае ложного утверждения), потребуется определить свою собственную макроподстановку для утверждений. Однако в целях отладки программ должно быть вполне достаточно стандартной макроподстановки assert.

Пример того, как может выглядеть пользовательское макроопределение ASSERT:

```
#define ASSERT(a) if (!a) {
   cerr << "\nУтверждение ложно. Условие: " << #a << " не выполнено."
        << "\nЛожное утверждение в строке " << __LINE__ << " файла:"
        << "\n\t\"" << __FILE__ << "\"\n";
```

cerr по функциональности идентичен cout, но используется для вывода информации об ошибках. Прототип этого оператора объявлен в файле <iostream>. Код #a преобразует ложное условие в строку. Конструкции __LINE__ и __FILE__ заменяются соответственно номером строки и именем файла.

Пример использования этой макроподстановки:

```
int array[] = {5,4,2,7,5};
for (int i = 0; i <5; i++)
{
     ASSERT(i <sizeof(array)/sizeof(int));
     array[i] = i;
}
```

В приведенном коде элементы массива инициализируются в теле оператора for. Однако прежде чем использовать i в качестве индекса, необходимо проверить утверждение, что i находится в пределах границ массива.

Утверждения могут использоваться в подобных случаях для предупреждения выхода за границы массива. Кроме того, утверждения могут использоваться в теле функций для проверки полученных параметров, а также для подтверждения корректности результатов работы сомнительного кода.

Помните, что утверждения используются для отладки кода, поэтому их следует удалять в момент создания окончательной версии для пользователей или других разработчиков. (Пользователям программы совершенно не обязательно знать, что в 675 строке файла встретилось ложное утверждение.) Чтобы разобраться с непредвиденными ошиб-

ками, возникающими в процессе работы программы, воспользуйтесь обработкой исключений.

Обработка исключений

При использовании кода, не являющегося полной программой (кода класса или набора функций), могут возникать непредвиденные ошибки. Нельзя полагаться на получаемые значения, как нельзя полагаться на компетентность пользователей кода. Идеальный фрагмент кода предотвратит отказ всей программы при получении некорректных параметров. Один из путей к такому идеалу связан с применением *обработки исключений*, которая позволяет управлять нестандартными ситуациями, возникающими в работе программы.

Обработка исключений действительно позволяет избежать непредсказуемых и исключительных событий. Исключительная ситуация возникает, когда фрагмент кода не может выполнить указанную задачу. Обработка исключений позволяет изменить способ решения задачи или попытаться решить проблему.

Исключение возникает, когда фрагмент кода *генерирует* исключение. Пример непредсказуемого события, или исключения, – закончилось свободное пространство на жестком диске компьютера. Но эта ситуация не обязательно приведет к сбою программы или невозможности завершить работу над задачей. Можно создать код, перехватывающий и обрабатывающий исключения определенного рода.

Для перехвата исключений служат блоки `try`. *Блок* `try` – это фрагмент кода, связанный с ключевым словом `try`. Данное ключевое слово отслеживает возникновение исключений. Блоки `try` имеют следующий синтаксис:

```
try
{
     фрагментКода
}
```

Применение такого блока сообщает компьютеру, что заключенный в нем фрагмент кода может сгенерировать исключение, а потому за ним необходимо наблюдать.

После создания блока `try` возникающие в нем исключения можно перехватывать при помощи оператора `catch`. Синтаксис оператора:

```
catch(имяИсключения)
{
     действия
}
```

Здесь *имяИсключения* определяет имя исключения. Когда фрагмент кода, заключенный в блок `try`, генерирует исключение, компьютер начина-

ет выполнение кода, заключенного в оператор `catch`. Оператор `catch` должен следовать непосредственно за оператором `try`, чтобы компьютер связал их. Между этими операторами код недопустим:

```
try
{
     фрагментКода
}
catch (имяИсключения)
{
     действия
}
```

За оператором `try` может следовать произвольное число операторов `catch`:

```
try
{
     фрагментКода
}
catch (имяИсключения1)
{
     действия1
}
catch (имяИсключения2)
{
     действия2
}
```

Когда генерируется определенное исключение, выполняется соответствующий оператор `catch`.

Исключения генерируются при помощи оператора `throw`. Его синтаксис:

```
throw исключение;
```

имяИсключения (в операторе `catch`) является именем типа, причем произвольного типа. Это может быть тип `int`, `char*` или `string`. По сути дела, этот параметр похож на аргумент функции, с той разницей, что ему не требуется имя переменной.

исключение является выражением. Применение оператора `throw` сравнимо с вызовом функции. Операторы `catch` можно считать перегруженными функциями. В зависимости от типа *исключения* выбирается наиболее подходящий оператор `catch` (если таковой существует).

Пример:

```
#include<iostream>

using namespace std;

int main (void)
{
```

```
    try
    {
        int x = 10;
        if (x == 10)
            throw x;
    }
    catch(int x)
    {
        cerr << "О нет! x равно " << x << "!!!!";
    }
    catch(float f)
    {
        cerr << "Как мы тут оказались?";
    }
    return 0;
}
```

В данном примере переменная x инициализируется значением 10. Программа проверяет x на равенство 10 и генерирует исключение, если условие истинно. Пример, очевидно, бесполезный, но позволяет проиллюстрировать сказанное.

Имена исключений в операторах catch – не просто имена типов. Они также являются объявлениями переменных, поскольку операторы catch работают аналогично функциям. Значение x (10) передается оператору catch, чтобы он мог должным образом обработать ошибку (информация о том, какая именно ошибка произошла, лишней для оператора перехвата не будет).

Кроме того, несмотря на наличие двух операторов catch, компьютер следует правилам, действующим для перегрузки функций, когда выбирает один из операторов.

Применение встроенных типов данных для работы с исключениями – идея, как правило, не очень хорошая. Если генерируется целое число, невозможно определить, какая произошла ошибка. Можно предполагать, что она связана с этим целочисленным значением, но наверняка сказать нельзя.

По счастью, существует стандартный способ передачи такой информации. Для каждого нового вида исключения определяется новый тип. Пример:

```
class FileError
{};
```

Заметили, что класс совершенно пустой? Дело в том, что ему не нужны компоненты. От него требуется только имя. Вот пример использования этого класса:

```
#include <iostream>
#include <istream>
#include <fstream>
```

```
using namespace std;

class FileError
{};

int main (void)
{

     try
     {
          char* filename = "hello.txt";
          fstream file;
          file.open(filename, ios::in) ;
          if (!file.is_open())
          throw FileError();
     }
     catch (FileError)
     {
          cerr << "Файловая ошибка.";
     }
     return 0;
}
```

Если код генерирует исключение, которое не перехватывается (нет подходящего оператора `catch`), компьютер самостоятельно предпринимает определенные действия. К сожалению, компьютеры довольно твердолобы. Они считают, что лучший способ обработать исключение – немедленно завершить программу. С этой целью компьютер вызывает функции `terminate()` и `unexpected()`, определенные в `<exception>`. Обе функции работают без аргументов и имеют тип возвращаемого значения `void`. Чтобы увидеть, что происходит при вызове этих функций, скомпилируйте и выполните следующую программу:

```
int main (void)
{
     throw 5; //произвольное значение
     return 0;
}
```

Можно заметить, что отображаемое сообщение об ошибке один в один повторяет сообщение функции `abort()`. Это происходит потому, что функция `terminate()` вызывает `abort()`. С помощью функции `set_terminate()` можно заставить `terminate()` вызывать любую другую функцию. Объявление

```
typedef void (*terminate_function)();
terminate_function set_terminate(terminate function term_func);
```

выглядит довольно сложно, но на деле достаточно передать имя функции в качестве параметра. Пример:

```
#include <exception>
#include <iostream>
```

```
using namespace std;
void error_handler();
void error_handler()
{
     cerr << "Возникла ошибка, которая не была перехвачена.";
}

int main (void)
{
     set_terminate(error_handler);
     throw 5;
     return 0;
}
```

Вывод:

```
Возникла ошибка, которая не была перехвачена.
```

Создание иерархии исключений

Если в крупной программе использовать исключения, точно описывающие ошибки, их число очень быстро вырастет до неразумных высот. Хвосты операторов `catch` после каждого блока `try` по меньшей мере мешают жить. Забудете всего один оператор – и столкнетесь с целой проблемой. По счастью, по мере набора опыта вы разработаете собственные методы решения этой проблемы.

Но есть и стандартный путь – использовать наследование (которое подробно описано в главе 8 «Наследование»). Иерархия исключений позволяет работать с обобщенными исключениями вместо конкретных либо с обобщенными в одних ситуациях и с конкретными в других. Такой метод является исключительно гибким и позволяет адаптировать программы к любым ситуациям.

Пример подобной иерархии:

```
class MathError {};
class DivideByZero : public MathError {};
class Overflow : public MathError {};
```

Эта иерархия позволяет обрабатывать конкретные исключения вроде `DivideByZero` (деление на нуль) либо математические ошибки в целом. Пример:

```
#include <iostream>

using namespace std;
int divide(int a, int b);

class MathError {};
class DivideByZero : public MathError {};
class Overflow : public MathError {};

int divide(int a, int b)
```

```
{
     if (b == 0)
     {
          throw DivideByZero();
          return false;
     }
     return a/b;
}

int main (void)
{
     try
     {
          if (!divide(5,0))
               throw MathError();
     }
     catch(DivideByZero)
     {
          cerr << "Ой! Попытка разделить на нуль!";
     }
     catch (MathError)
     {
          cerr << "Возникла какая-то математическая ошибка.";
     }
     return 0;
}
```

Вывод:

```
Ой! Попытка разделить на нуль!
```

Как видите, если попытаться разделить на нуль, генерируется исключение `DivideByZero`, а во всех остальных случаях более общее исключение `MathError`.

Этот пример иллюстрирует и другую важную концепцию. Если вызвать функцию внутри блока `try`, ошибки, сгенерированные этой функцией, также подлежат перехвату.

Перехват всех исключений

Можно перехватывать все исключения в программах, делая их практически неуязвимыми. С этой целью используется разновидность оператора `catch`, реализующая перехват всех исключений, которые еще не были обработаны. В качестве аргумента такого оператора `catch` выступает многоточие (`...`), а не тип исключения. При поиске оператора `catch`, который может обработать исключения, такой оператор будет использован в качестве последнего средства (если вы не забудете включить его в код). Синтаксис:

```
catch(...)
{
```

```
    обработкаИсключения
}
```

Приведем пример использования оператора `catch` по умолчанию:

```
#include <iostream>

using namespace std;

int main (void)
{
     try
     {
          throw 6;
     }
     catch(...)
     {
          cerr << "Возникло исключение, но оно перехвачено!";
     }
     return 0;
}
```

Вывод:

```
Возникло исключение, но оно перехвачено!
```

Эта версия оператора `catch` должна замыкать последовательность прочих операторов `catch`. В противном случае операторы `catch`, следующие за универсальным, не будут использоваться ни при каких обстоятельствах. Однако если универсальный оператор находится в конце списка, то он будет использован лишь в качестве крайнего средства.

Игра «Минное поле»

В качестве одного из членов элитного советского отряда XJ77 вы отправляетесь на разминирование смертоносного минного поля. Невероятно опасное задание. Только самые умные и умелые смогут выжить там. Скомпилируйте программу и проверьте, обладаете ли вы нужными качествами.

```
//11.2 - Минное поле - Марк Ли - Premier Press
#include <iostream>
#include <exception>
#include <string>
#include <vector>
#include <cstdlib>
#include <ctime>
#include "MenuUtility.h"

using namespace std;
using namespace menuNamespace;

class StepOnMine{};
```

```
class FailedDisarm{};

class MineField
{
     vector<bool> minefield;
     //точки, где игрок уже был
     vector<bool> beenThere;
     int location; //текущие координаты игрока
public:
     MineField()  //минное поле 4x4
     {
          srand(time(0));
          location = 0;
          for (int c = 0; c <16; c++)
          {
               minefield.push_back(false);
               beenThere.push_back(false);
          }
          for (int i = 0; i <10; i++) //разместить 10 мин случайным образом
               minefield[rand()%15+1] = true;
          beenThere[0] = true;
     }

     bool IsAMine(int location)
     {
          return minefield[location];
     }

     string draw()
     {
          string s;
          for (int i = 0; i <4; i++)
          {
               for (int c=0; c<4; c++)
               {
                    if (location == i*4+c)
                         s+='P';
                    else {
                         if (beenThere[i*4+c])
                              s+= "X";
                         else s+= " ";
                    }
                    s+="|";
               }
               s+= '\n';
          }
          return s;
     }

     bool moreMines()
     {
          for (int i=0; i<16; i++)
               if (beenThere[i]) return true;
```

```
            return false;
        }

        int Directions()
        {
            string options[4];
            options[0] = "Север";
            options[1] = "Восток";
            options[2] = "Юг";
            options[3] = "Запад";
            return menu(options, 4);
        }

        int& place() {return location;}

        void goThere(int place) {beenThere[place] = true;}
};

void Detonate()
{
    cout << "Вы подорвались на мине. Бабах!!!\n";
}

void disarm()
{
    int temp = rand()%2+1;
    if (temp-1)
        throw FailedDisarm();
}

int main (void)
{
    set_terminate(Detonate);
    MineField m; //создаем минное поле

    string input;
    cout << "Добро пожаловать на минное поле!!\n"
        << "В составе элитного отряда советских саперов XJ77 \n"
       <<" вы отправляетесь разминировать смертоносное поле, щедро усеянное "
        <<" реагирующими на тепловое излучение противопехотными минами.\n"
        << "Многие из вас погибнут.\n"
        << "Только лучшим из лучших суждено увидеть "
        << "следующий рассвет.\n Сможете ли вы пройти испытание?\n";
    cin >> input;
    if(input == "no" || input == "No")
        goto TOO_BAD;
    cout << "Вы находитесь в северо-западном углу поля.\n";
PLAY:
    try
    {
        int goTo;
        while(m.moreMines()){
            cout << endl << m.draw();
```

```
            cout << "Ваше положение отмечено буквой P.\n"
                 << "В каком направлении двигаться?"
                 <<endl;
            bool proper = false;
            do {
                 goTo = m.Directions();
                 if (goTo == 1 && m.place() >3)
                      proper = true;
                 if (goTo == 2 && (m.place()-3)%4 != 0)
                      proper = true;
                 if (goTo == 3 && m.place() <12)
                      proper = true;
                 if (goTo == 4 && m.place()%4 != 0)
                      proper = true;
                 if (!proper)
                      cout<<"\nВ ту сторону идти нельзя.\n";
            }while (!proper);

            if (goTo == 1)
                 m.place() -= 4;
            if (goTo == 2)
                 m.place()++;
            if (goTo == 3)
                 m.place() += 4;
            if (goTo == 4)
                 m.place()Ч;

            m.goThere(m.place());

            if (m.IsAMine(m.place()))
                 throw StepOnMine();
       }
  }
  catch(StepOnMine)
  {
       int input;
       do {
       cout << "\nВы нашли мину.\n"
            << "Ваши действия?\n"
            << "[1]Попытаться разминировать.\n"
            << "[2]Бежать.\n";
       cin >> input;
       }while(input <1 && input >2);
       if (input == 1)
       {
            try { disarm(); }
            catch(FailedDisarm) {terminate();}
            cout << "Мина обезврежена!!!\n";
            goto PLAY;
       }
       cout << "Вы подвели команду XJ77.\n";
       goto TOO_BAD;
```

```
    }

    return 0;
TOO_BAD: cout << "\nПопробуй на следующий год, парень.\n";
    return 0;
}
```

Резюме

Пусть и короткая, эта глава является очень важной. Даже самые опытные программисты слишком часто игнорируют концепции, описанные здесь, поскольку в действительности обработка исключений не повышает функциональности программ. Однако обработка исключений делает программы стабильными, и если вы хотите превратиться из хорошего программиста в исключительного, мы настоятельно советуем приобрести привычку использовать описанные методики.

Задания

1. Для чего нужно ключевое слово `try`?
2. Каково назначение иерархий исключений?
3. Каким образом можно создавать программы, которые не совершают ошибок?
4. Дайте определение исключения.
5. На какой стадии разработки программы следует использовать утверждения?

12

Программирование для Windows

Предшествующие главы были посвящены подробному изучению языка C++. Теперь на протяжении двух глав мы будем заниматься более современными технологиями программирования. В этой главе вы научитесь создавать программы для Windows с помощью прикладного интерфейса программирования Windows API. Пройдя такую подготовку, вы приступите к изучению главы 13 «DirectX», которая расскажет о применении библиотеки Microsoft DirectX для создания игр. В этой главе:

- Как создать программу Windows
- Применение `WinMain`
- Применение `WndProc`
- Создание окон
- Обработка сообщений

Знакомьтесь, Windows API

При разработке системы Windows компания Microsoft приняла решение создать библиотеку функций, которыми могли бы пользоваться разработчики для создания Windows-программ. Эта библиотека получила имя *Windows API* (Application Programming Interface, интерфейс прикладного программирования). Чтобы воспользоваться библиотекой Windows API, необходимо работать с одной из перечисленных систем: Windows 95/98/NT 4/2000. Полная документация по Windows API доступна по адресу *http://www.msdn.microsoft.com/library/default.asp* в разделе Platform SDK.

Одно из подмножеств этой библиотеки носит название *Windows GUI* (Graphic User Interface, графический интерфейс пользователя). Windows GUI позволяет разнообразить внешний вид программ, добавлять управляющие кнопки, списки прокрутки, пиктограммы и другие эле-

менты Windows, которые помогают пользователям взаимодействовать с программами.

Создание программы для Windows в CodeWarrior

Поздравляем, читатель; тебе предстоит выезд на экскурсию в целый новый мир программирования в стране Microsoft. Программирование для Windows – занятие более увлекательное, чем разработка устаревших консольных приложений, поскольку автор в большинстве случаев сразу видит результаты своей работы. Кроме того, около 98 процентов компьютеров работают под управлением Windows, что делает написанные программы переносимыми.

Действия по созданию программы для Windows в корне отличаются от тех, что связаны с консольными приложениями. В главе 1 «Путешествие начинается» мы советовали открыть диалоговое окно «New Project» (следующее после выбора имени проекта) и выбрать приложение типа «C++ Console App». Этот шаг сообщает компилятору CodeWarrior, что создаваемая программа не является приложением Windows. Однако теперь мы обратимся к другому варианту из предлагаемых в этом диалоговом окне, «Win32 C++ App».

Чтобы познакомить читателей с процессом, мы выполним необходимые действия шаг за шагом. Приведенная здесь последовательность действий позволяет создать программу, отображающую на экране сообщение `Hello from Win32`.

1. Запустите CodeWarrior и выберите пункт New из меню File. Диалоговое окно New откроется на вкладке Project.
2. В поле имени проекта (Project name) наберите строку `WinHelloWorld` (рис. 12.1).
3. В поле расположения проекта (Location) наберите имя каталога или нажмите на кнопку Set, чтобы получить обзор файловой системы и выбрать каталог.
4. В списке в левой части диалогового окна New выберите пункт «Win32 C/C++ Application Stationery».
5. Нажмите OK, чтобы перейти в диалоговое окно нового проекта (New Project) (рис. 12.2).
6. Выберите из списка параметров пункт «Win32 C++ App».
7. Нажмите OK. Откроется окно проекта. Создание каркаса приложения завершено.
8. В окне проекта содержится исходный текст программы для Windows (рис. 12.3).

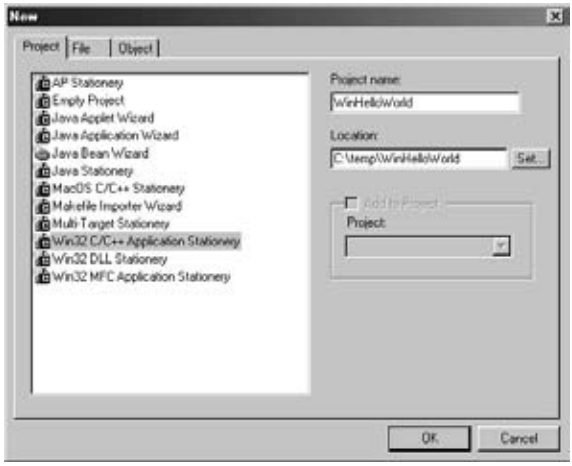


Рис. 12.1. *В диалоговом окне New можно задать тип проекта, его имя и расположение*

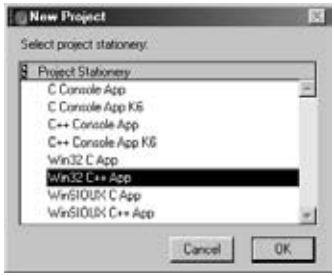


Рис. 12.2. *Диалоговое окно New Project*

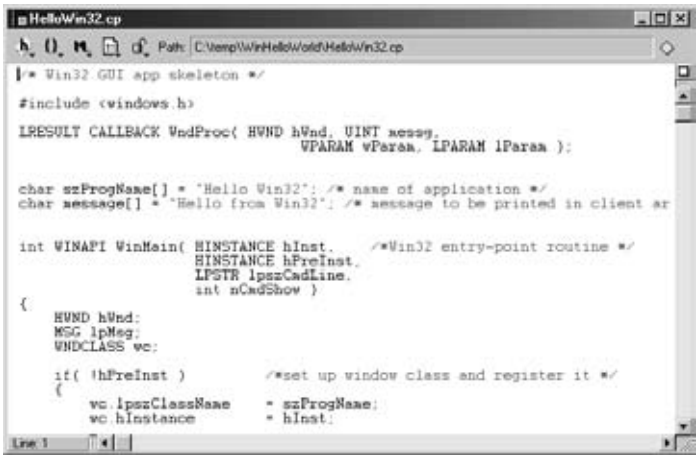


Рис. 12.3. *Это окно содержит исходный код, автоматически созданный компилятором CodeWarrior для приложения Windows*

9. Скомпилируйте и выполните программу как привычное консольное приложение. Откроется окно «CodeWarrior Win32 Stationery» с сообщением `Hello from Win32` (рис. 12.4). Создание первого Windows-приложения завершено.

Рис. 12.4. *Каркас кода, созданного компилятором, создает окно, которое можно перемещать, разворачивать, сворачивать и изменять в размерах*

И хотя многое сейчас осталось за кадром, уже видно, что Windows-приложения совсем не сложно создавать. В последующих разделах мы изучим код каркаса, сгенерированного компилятором CodeWarrior в предшествующем шаге.

Изучаем функции Windows

В процессе программирования для Windows приходится довольно быстро познакомиться с функциями `WinMain` и `WndProc`. Данный раздел посвящен именно этим функциям.

Выполнив шаги из предшествующего раздела, вы, вероятно, заметили, что CodeWarrior самостоятельно создает довольно большой объем кода. Ниже мы приведем его полностью:

```
/* Win32 GUI app skeleton */

#include <windows.h>

LRESULT CALLBACK WndProc( HWND hWnd, UINT messg, WPARAM wParam, LPARAM lParam );

char szProgName[] = "Hello Win32"; /* name of application */
/* message to be printed in client area */
char message[] = "Hello from Win32";


int WINAPI WinMain(/*Win32 entry-point routine */
                        HINSTANCE hInst,
                        HINSTANCE hPreInst,
                        LPSTR lpszCmdLine,
                        int nCmdShow )
```

```
{
     HWND hWnd;
     MSG lpMsg;
     WNDCLASS wc;

     if( !hPreInst ) /*set up window class and register it */
     {
          wc.lpszClassName = szProgName;
          wc.hInstance = hInst;
          wc.lpfnWndProc = WndProc;
          wc.hCursor = LoadCursor( NULL, IDC_ARROW );
          wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
          wc.lpszMenuName = NULL;
          wc.hbrBackground = (HBRUSH)
               GetStockObject( WHITE_BRUSH );
          wc.style = 0;
          wc.cbClsExtra = 0;
          wc.cbWndExtra = 0;

          if( !RegisterClass( &wc ) )
               return FALSE;
     }

hWnd = CreateWindow(/* now create the window */
          szProgName,
          "CodeWarrior Win32 stationery",
          WS_OVERLAPPEDWINDOW,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          CW_USEDEFAULT,
          (HWND)NULL,
          (HMENU)NULL,
          (HINSTANCE)hInst,
          (LPSTR)NULL
     );

     ShowWindow(hWnd, nCmdShow );
     UpdateWindow( hWnd );


     /* begin the message loop */
     while( GetMessage( &lpMsg, NULL, 0, 0 ) )
     {
          TranslateMessage( &lpMsg );
          DispatchMessage( &lpMsg );
     }
     return( lpMsg.wParam);
}

/*callback procedure */
LRESULT CALLBACK WndProc( HWND hWnd, UINT messg,
                                        WPARAM wParam, LPARAM lParam )
{
```

```
    HDC hdc; /* handle to the device context */
    PAINTSTRUCT pstruct; /*struct for the call to BeginPaint */

    switch(messg)
    {
        case WM_PAINT:
        /* prepare window for painting*/
        hdc = BeginPaint(hWnd, &pstruct );
        /*print hello at upper left corner */
        TextOut( hdc, 0, 0, message,
            ( sizeof(message) - 1 ) );
        /* stop painting */
        EndPaint(hWnd, &pstruct );
        break;

        case WM_DESTROY:
        PostQuitMessage( 0 );
        break;

        default:
        return( DefWindowProc( hWnd, messg,
            wParam, lParam ) );
    }

    return( 0L );
}
```

Учитывая уже приобретенные знания C++, некоторые фрагменты кода, вероятно, покажутся читателям знакомыми. Прежде всего, в коде присутствует единственная директива включения:

```
#include <windows.h>
```

Этот оператор включает в код стандартную библиотеку Windows, необходимую для создания программ Windows. Windows API содержит целый ряд библиотек, но этот заголовочный файл является основным. Запомните, что он должен включаться в любую программу для Windows.

Кроме того, выделяются два глобальных символьных массива, `szProgName` и `message`:

```
char szProgName[] = "Hello Win32"; /* name of application */
char message[] = "Hello from Win32";
/* message to be printed in client area */
```

Эти переменные не являются обязательными. Они созданы из соображений удобства. Как становится ясно из комментариев, `szProgName` хранит имя программы, а `message` – текст, который выводится на экран. Отметим, что если изменить эти значения, изменится отображаемый текст! Возможно, читателям это не покажется столь уж восхитительным, но, по крайней мере, понятно, что здесь есть выбор.

Следующее, что можно сразу понять из кода: в нем присутствует пара функций, WinMain и WndProc:

```
int WINAPI WinMain(
     HINSTANCE hInst,       /*Win32 entry-point routine */
     HINSTANCE hPreInst,
     LPSTR lpszCmdLine,
     int nCmdShow );
LRESULT CALLBACK WndProc( HWND hWnd, UINT messg,
     WPARAM wParam, LPARAM lParam );
```

Скорее всего, многие термины в этих объявлениях не знакомы читателям: скажем, WINAPI, HISTANCE, LPSTR, LRESULT, CALLBACK, HWND, UINT, WPARAM, а также LPARAM. WINAPI и CALLBACK – это специальные термины, которые пока что можно игнорировать (просто условимся считать, что их нет).

Все остальные слова представляют типы данных Windows различного назначения. Например, WinMain – это Windows-эквивалент функции main. Он служит той же цели, что и функция main в консольных программах – создает точку входа в программу. Все аргументы, передаваемые функции WinMain (от Windows к приложению) являются Windows-эквивалентами аргументов командной строки. Они являются источником всей информации, необходимой программе для выполнения стандартных функций Windows.

Функция WndProc обрабатывает специальные события, такие как нажатие кнопки. Аргументы WndProc передают информацию о типе события и его контексте.

Истории из жизни

В программировании для Windows особое значение имеют *события*. После запуска обычной Windows-программы она просто загружается и ожидает ввода пользователя, в отличие от, скажем, игр-стратегий реального времени, в которых компьютерные оппоненты собирают армию с целью уничтожить игрока вне зависимости от его действий. Первый вид программ создан при помощи *событийно-управляемого программирования*.

События в системе Windows происходят часто в конкретных ситуациях – например, когда наступает определенное время или пользователь нажимает клавишу <Enter>, или кнопку мыши. События возникают постоянно в процессе работы Windows.

Когда программисты проектируют программы для Windows, то создают функции, называемые *обработчиками событий* (более подробно эти функции описаны в разделе «Обработка событий»). Windows вызывает эти функции, когда возникает определенное событие. Например, когда пользователь нажимает на кнопку Exit в программе, последняя немедленно завершает работу.

Исследуем WinMain

Как было сказано выше, `WinMain` – это Windows-эквивалент функции `main`. Система Windows вызывает `WinMain` при запуске программы. Когда завершается выполнение `WinMain`, завершается и программа. В этом разделе мы подробно изучим функцию `WinMain`.

Windows передает приложению четыре аргумента: `HINSTANCE hInst`, `HINSTANCE hPreInst`, `LPSTR lpszCmdLine` и `int nCmdShow`.

`hInst` – это дескриптор приложения. *Дескриптор* во многом похож на указатель или ссылку и используется для отслеживания различных элементов системы Windows, таких как приложения, окна приложений и др. `Inst` является сокращением для *instance* (*экземпляр*). Если в одно время выполняются две копии приложения, каждая является отдельным экземпляром одного и того же приложения. `hInst` является дескриптором конкретного экземпляра приложения. Этот параметр может использоваться при вызове некоторых функций Windows API.

`hPreInst` – это дескриптор предыдущего экземпляра приложения. Параметр устарел и уже бесполезен, так что можете спокойно о нем забыть. Он существует только из соображений обратной совместимости. *Обратная совместимость* – это способность более новых версий программ взаимодействовать с более старыми. Она избавляет пользователей более старых программ от необходимости постоянно покупать новые версии.

`lpszCmdLine` – обычная строка, завершаемая пустым символом, которая хранит аргументы командной строки. Если воспользоваться меню Пуск и выбрать пункт Выполнить, после имени программы можно набрать любую строку. К примеру, если программа называется Hello.exe, можно набрать `Hello.exe Goodbye`. Программу можно спроектировать таким образом, что она будет выводить на печать любой аргумент, полученный в командной строке (в нашем случае – `Goodbye`). Отметим, что имя программы, Hello.exe, не является частью `lpszCmdLine`.

`nCmdShow` является целым числом, определяющим начальное состояние окна. Можно считать это значение предложением от системы Windows, которое, как и большинство других предложений, можно спокойно игнорировать. Аргумент `nCmdShow` имеет множество значений, которые мы не будем здесь перечислять; в табл. 12.1 описаны наиболее употребительные из них.

Выучить наизусть все возможные значения практически невозможно (табл. 12.1 содержит приблизительно 5% существующих значений). Но, скорее всего, вам придется пользоваться лишь первыми тремя значениями из табл. 12.1 в 99% своих программ.

Тем не менее, если возникнет интерес к прочим значениям, информацию о них можно найти в различных местах. Одним из таких источников является информационная сеть для разработчиков, Microsoft De-

veloper Network (MSDN), доступная по адресу *http://www.msdn.microsoft.com*. Более подробную информацию по программированию для Windows можно найти в библиотеке, в разделе *Platform SDK*.

Таблица 12.1. Возможные значения nCmdShow

Значение	Описание
SW_SHOWNORMAL	Активирует и отображает окно (окно занимает лишь часть экрана)
SW_SHOW	Активирует и отображает окно с текущими параметрами положения и размера
SW_HIDE	Скрывает окно
SW_MAXIMIZE	Увеличивает до максимума размер окна
SW_MINIMIZE	Сворачивает окно
SW_RESTORE	Активирует и отображает свернутое окно. Окно отображается с прежними параметрами положения и размера

Исследуем WndProc

WndProc – это функция, вызываемая системой Windows в момент, когда возникает событие, затрагивающее программу. Windows передает WndProc аргументы, которые поясняют, какое именно событие произошло. Автор программы может предоставить обработку событий системе, а может обработать их так, как считает нужным. Все события невозможно обработать самостоятельно – их слишком много.

WndProc получает четыре аргумента: HWND hWnd, UINT messg, WPARAM wParam и LPARAM lParam.

hWnd является дескриптором окна, к которому относится событие. Этот аргумент используется довольно редко, но если в программе существует несколько окон, обслуживаемых одной версией WndProc, hWnd позволяет определить, какое из окон является получателем сообщения. Тип данных HWND в программировании для Windows используется для работы с окнами. Дескрипторы используются аналогично классам. Например, чтобы обратиться к растровому изображению, следует воспользоваться типом данных HBITMAP.

messg содержит идентификатор сообщения. Существуют сотни различных сообщений, некоторые из которых описаны в табл. 12.2.

Ниже по тексту главы в разделе «Обработка событий» вы научитесь пользоваться некоторыми из идентификаторов сообщений. А пока достаточно знать, что они существуют, причем исчисляются сотнями.

wParam и lParam содержат дополнительную информацию о событии. Эти аргументы имеют различный смысл для различных событий и в зависимости от типа сообщения могут содержать или не содержать значения.

Таблица 12.2. Идентификаторы основных сообщений

Идентификатор	Действие
WM_ACTIVATE	Посылается, когда окно активируется или получает фокус ввода
WM_CLOSE	Посылается при закрытии окна
WM_CREATE	Посылается при создании окна
WM_DESTROY	Посылается перед уничтожением окна
WM_MOVE	Посылается при перемещении окна
WM_MOUSEMOVE	Посылается при перемещении указателя мыши
WM_KEYUP	Посылается, когда отпущена клавиша на клавиатуре
WM_KEYDOWN	Посылается при нажатии клавиши
WM_TIMER	Посылается, когда возникает событие таймера
WM_USER	Специальное сообщение, посылается при создании пользовательских событий
WM_PAINT	Посылается при необходимости обновить изображение окна
WM_QUIT	Посылается при завершении работы приложения Windows (а не отдельного окна)
WM_SIZE	Посылается при изменении размера окна

LRESULT является стандартным типом возвращаемого значения для многих функций Windows. Данный тип используется для хранения результатов выполнения функции.

Создание окон

Одним из важных умений в программировании для Windows является умение создать окно и отобразить его на экране. В этом разделе мы посвятим читателей в тонкости процесса и расскажем о возможных вариантах.

Создание окна сообщений

Прежде чем продолжить исследование кода, сгенерированного CodeWarrior, необходимо чуть больше узнать о создании окон.

В создании окон есть два варианта. Можно создавать полнофункциональные окна, обладающие большим потенциалом, либо простые окна, которые умеют только отображать сообщения. В этом разделе мы расскажем о простых окнах, известных в качестве *окон сообщений*, в качестве подготовки к изучению создания произвольных окон.

Для создания окна сообщений применяется функция MessageBox. Информация о создаваемом окне передается функции MessageBox посредством четырех аргументов. Объявление функции MessageBox выглядит так:

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

hWnd указывает дескриптор окна, которое будет управлять окном сообщений. Например, если в приложении возникла ошибка, ее можно отобразить в окне сообщений. В этом случае управляющим окном будет главное окно приложения. Окно, управляющее другим окном, называется *родительским окном*, а подчиненное окно – *порожденным*. Если в качестве значения этого аргумента передать NULL (0), родительским окном считается рабочий стол Windows.

lpText содержит строку, которую необходимо отобразить. Чтобы отобразить сообщение Привет, Win32, следует передать эту строку в качестве аргумента.

lpCaption содержит строку, которая отображается в заголовке окна. Например, если открыть CodeWarrior, в верхней части экрана можно видеть текст Metrowerks CodeWarrior. Эта строка является заголовком окна компилятора CodeWarrior. В заголовке главного окна приложения обычно содержится имя приложения (или нечто равноценное такому имени).

uType – целочисленное значение, аналогичное аргументу nCmdShow функции WinMain, которое определяет стиль окна сообщений. Можно указать, какие кнопки присутствуют в окне и какая пиктограмма отображается рядом с сообщением. Некоторые значения uType приведены в табл. 12.3, они определяют комбинации для кнопок. Возможные значения для пиктограмм приведены в табл. 12.4.

Таблица 12.3. Значения uType для кнопок

Значение	Описание
MB_OK	Значение по умолчанию. В окне сообщений присутствует кнопка OK
MB_OKCANCEL	В окне сообщений присутствуют кнопки OK и Cancel (Отмена)
MB_RETRYCANCEL	В окне сообщений присутствуют кнопки Retry (Повтор) и Cancel (Отмена)
MB_YESNO	В окне сообщений присутствуют кнопки Yes (Да) и No (Нет)
MB_YESNOCANCEL	В окне сообщений присутствуют кнопки Yes (Да), No (Нет) и Cancel (Отмена)

Кнопки и пиктограммы могут комбинироваться посредством оператора «или». Например, чтобы создать окно сообщений с кнопками OK и Cancel (Отмена)[1], а также пиктограммой знака «стоп», передайте значение MB_OKCANCEL | MB_ICONSTOP.

[1] Фактически работу по созданию кнопок выполняет не программа пользователя, а Windows. Поэтому надписи на кнопках могут различаться в зависимости от используемой версии и ее языка. Программа, создающая в английской версии Windows окно с кнопками OK и Cancel, в русской версии создаст окно с кнопками OK и Отмена. – *Примеч. ред.*

Таблица 12.4. Значения uType для пиктограмм

Значение	Описание
MB_ICONEXCLAMATION	Окно сообщений содержит пиктограмму с восклицательным знаком
MB_ICONINFORMATION	Окно сообщений содержит пиктограмму в виде буквы i в круге
MB_ICONQUESTION	Окно сообщений содержит пиктограмму с вопросительным знаком
MB_ICONSTOP	Окно сообщений содержит пиктограмму со знаком «стоп»

Пример создания окна сообщений:

```
//12.1 - Окно сообщений - Марк Ли
//Premier Press
#include <windows.h>
int WINAPI WinMain( /*Win32 entry-point routine */
                          HINSTANCE hInst,
                          HINSTANCE hPreInst,
                          LPSTR lpszCmdLine,
                          int nCmdShow )
{
     MessageBox(NULL, "Программировать для Windows легко!!",
          "Мое первое окно сообщений", MB_OKCANCEL | MB_ICONINFORMATION);
}
```

Вывод программы приведен на рис. 12.5. Обратите внимание, заголовок окна сообщений отображается также на панели задач. Это происходит потому, что родительским окном в данном случае является рабочий стол Windows.

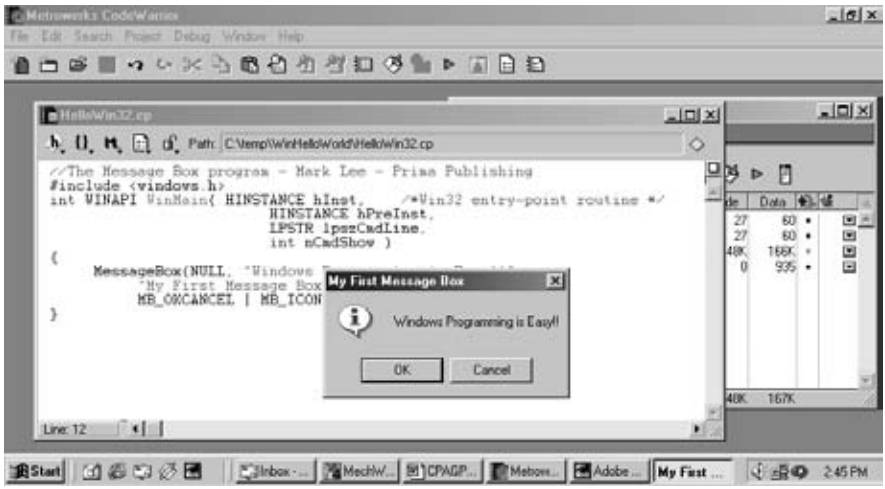


Рис. 12.5. *Код программы определяет вид и содержание окна сообщений*

Установка свойств окна

Прежде чем создать окно, необходимо задать его свойства. С этой целью используются структуры WNDCLASS и WNDCLASSEX. Эти структуры содержат ряд полей данных, в которых хранится соответствующая информация. После заполнения значений полей данных информация передается системе Windows в целях создания окна.

При создании приложения Win32 в компиляторе CodeWarrior используется стандартная структура WNDCLASS. Однако это структура уже устарела, и, скорее всего, ей на смену придет более современная структура WNDCLASSEX. В примерах этой книги мы будем использовать структуру WNDCLASSEX.

Определение WNDCLASSEX:

```
typedef struct _WNDCLASSEX
{
     UINT cbSize; //размер этой структуры
     UINT style; //стиль главного окна
     WNDPROC lpfnWndProc; //функция обработки событий
     int cbClsExtra; //дополнительная информация
     int cbWndExtra; //дополнительная информация окна
     //дескриптор экземпляра приложения
     HANDLE hInstance;
     HICON hIcon; //пиктограмма окна
     HCURSOR hCursor; //курсор для окна
     //способ заполнения фона окна
     HBRUSH hbrBackground;
     //имя меню (если оно существует)
     LPCTSTR lpszMenuName;
     //имя для класса
     LPCTSTR lpszClassName;
     //используемая пиктограмма (меньшего размера)
     HICON hIconSm;
} WNDCLASSEX;
```

Если взглянуть на код, сгенерированный компилятором CodeWarrior, можно обнаружить такую строку:

```
WNDCLASS wc;
```

Эта строка создает объект wc типа WNDCLASS. Чтобы воспользоваться структурой WNDCLASSEX, измените строку следующим образом:

```
WNDCLASSEX wc;
```

После создания объекта этой структуры следует заполнить поля подходящими значениями. Так выглядит заполнение wc в коде, сгенерированном компилятором:

```
wc.lpszClassName = szProgName;
wc.hInstance = hInst;
```

```
wc.lpfnWndProc = WndProc;
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
wc.lpszMenuName = NULL;
wc.hbrBackground = (HBRUSH)GetStockObject( WHITE_BRUSH );
wc.style= 0;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
```

Код может немного отличаться, если вы заполняете структуру самостоятельно. Первым полем структуры WNDCLASSEX является cbSize. Эта переменная хранит размер объекта. Зачем это нужно? При передаче объекта по указателю этим значением можно воспользоваться, чтобы определить точный объем памяти, занимаемый объектом. По большей части указание размера является мерой предосторожности, но это поле всегда следует заполнять. Вот так можно указать корректное значение cbSize:

```
wc.cbSize = sizeof(WNDCLASSEX);
```

Следующее поле – style. Оно определяет поведение окна. Поле заполняется константами, которые могут комбинироваться посредством оператора «или». Некоторые из допустимых значений приведены в табл. 12.5.

Таблица 12.5. Стили поведения окна

Значение	Действие
CS_HREDRAW	Обновляет изображение всего окна, если перемещение или подстройка размера изменяет ширину окна
CS_VREDRAW	Обновляет изображение всего окна, если перемещение или подстройка размера изменяет высоту окна
CS_OWNDC	Упрощает рисование в окне
CS_DBLCLKS	Посылает сообщение о двойном щелчке, произошедшем в области окна
CS_NOCLOSE	Отключает команду Close (Закрыть) системного меню окна

Этот обзор позволит читателям прочувствовать некоторые из возможностей программ Windows. Выбор значений зависит от вида приложения, которое вы разрабатываете. Для целей нашей книги можно оставить значение этого поля нулевым (как это сделано в коде, созданном компилятором):

```
wc.style = 0;
```

Следующее поле определяет функцию обработки событий, получаемых от системы Windows. В коде, сгенерированном CodeWarrior, это функция WndProc. Однако ей можно дать любое имя. Например:

```
wc.lpfnWndProc = WinProc;
```

Установка этого значения говорит Windows, какую из функций вызывать. Впоследствии Windows автоматически вызывает эту функцию при необходимости.

Еще два поля, `cbClsExtra` и `cbWndExtra`, предназначаются для обогащения функциональности окна, но их описание выходит за пределы этой книги. Достаточно будет установить нулевые значения:

```
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
```

Следующее поле, `hInstance`, является дескриптором экземпляра приложения. По счастью, Windows предоставляет значение этого параметра. Достаточно просто воспользоваться аргументом `hInst`, полученным функцией `WinMain`:

```
wc.hInstance = hInst;
```

Вызов функции Windows `LoadIcon` позволяет инициализировать следующее поле – `hIcon`. Можно использовать произвольную пиктограмму, а для наших целей вполне подойдет *системная пиктограмма* (то есть поставляемая в составе системы Windows). В табл. 12.6 перечислены некоторые из доступных системных пиктограмм.

Таблица 12.6. Системные пиктограммы

Значение	Описание
`IDI_APPLICATION`	Стандартная пиктограмма приложения (довольно непривлекательная)
`IDI_ASTERISK`	Пиктограмма звездочки
`IDI_EXCLAMATION`	Восклицательный знак (очень эмоциональный)
`IDI_HAND`	Пиктограмма с ладонью
`IDI_QUESTION`	Вопросительный знак
`IDI_WINLOGO`	Логотип системы Windows

В коде, сгенерированном компилятором, используется пиктограмма `IDI_APPLICATION`. Если вы предпочитаете не вмешиваться в процесс течения жизни, можно все так и оставить. Но некоторые из нас предпочитают ускорять события. Они предпочтут воспользоваться пиктограммой `IDI_EXCLAMATION`. Осторожно: сердечникам не рекомендуется. Значение устанавливается следующим образом:

```
wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
```

Поле `hCursor` инициализируется практически таким же образом, при помощи функции `LoadCursor`. `hCursor` указывает, как выглядит указатель мыши, когда находится в области окна. Опять же, можно создать свой курсор, но для целей главы можно обойтись и стандартными. Некоторые из системных курсоров перечислены в табл. 12.7.

Таблица 12.7. Системные курсоры

Значение	Описание
IDC_ARROW	Стандартный указатель
IDC_APPSTARTING	Стандартный указатель и маленькие песочные часы
IDC_CROSS	Перекрестие
IDC_IBEAM	Текстовый курсор
IDC_NO	Перечеркнутый круг
IDC_SIZEALL	Четырехлепестковая стрелка
IDC_SIZENESW	Двухлепестковая стрелка, указывающая на северо-восток и юго-запад
IDC_SIZENWSE	Двухлепестковая стрелка, указывающая на северо-запад и юго-восток
IDC_SIZENS	Двухлепестковая стрелка, указывающая на север и юг
IDC_SIZEWE	Двухлепестковая стрелка, указывающая на запад и восток
IDC_UPARROW	Стрелка вверх
IDC_WAIT	Песочные часы

Следующее поле, hbrBackground, определяет вид фона окна. Можно использовать специальные эффекты (вроде отвратительных полос), но большинство людей предпочитают сплошную заливку цветом. Способ закрашивания фона определяется при помощи кистей. Кисть (brush) в системе Windows – это способ хранения информации о том, как следует заполнять область цветом. При желании можно создавать собственные кисти, но большинство людей этого не делают. Некоторые из стандартных кистей перечислены в табл. 12.8.

Таблица 12.8. Системные кисти

Значение	Описание
BLACK_BRUSH	Сплошной черный
WHITE_BRUSH	Сплошной белый
GRAY_BRUSH	Сплошной серый
LTGRAY_BRUSH	Сплошной светло-серый
DKGRAY_BRUSH	Сплошной темно-серый
NULL_BRUSH	Пустая кисть

Дескриптор кисти запрашивается вызовом функции GetStockObject. Функция принимает один аргумент – имя кисти. Пример вызова:

```
wc.hbrBackground = GetStockObject(WHITE_BRUSH);
```

Чтобы организовать меню в окне (такое, например, как меню File в компиляторе CodeWarrior), воспользуйтесь полем lpszMenuName и определите состав меню. В противном случае следует установить значение NULL:

```
wc.lpszMenuName = NULL;
```

Следующее поле дает имя классу, который служит для создания окна. Каждое окно является объектом. При заполнении структуры WNDCLASSEX мы создаем новый класс. Затем на основе этого класса создается окно. Классу можно дать любое удобное для вас имя (поскольку в действительности это имя особого значения не имеет). Многие программисты используют простейшие имена вроде WINCLASS1. Пример:

```
wc.lpszClassName = "WINCLASS1";
```

И последнее поле. Оно хранит уменьшенную версию пиктограммы. Именно эта версия отображается в заголовке окна и на панели задач. Значение этому полю присваивается так же, как полю hIcon:

```
wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);
```

Вот, собственно, и все. Взгляните на окончательный вариант кода инициализации структуры WNDCLASSEX:

```
WNDCLASSEX wc;

wc.cbSize = sizeof(WNDCLASSEX);
wc.style = 0;
wc.lpfnWndProc = WinProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInst;
wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
wc.hbrBackground = GetStockObject(WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = "WINCLASS1";
wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);
```

Регистрация и создание окон

После создания класса окна (инициализации структуры WNDCLASSEX) следует зарегистрировать его в Windows посредством вызова функции RegisterClassEx. Функция имеет один аргумент – ссылку на объект WNDCLASSEX (wc).

Если окно создается на базе структуры WNDCLASS, класс следует регистрировать вызовом функции RegisterClass.

Вот так регистрируется класс окна:

```
RegisterClassEx(&wc);
```

Функция возвращает логическое значение, отражающее результат работы (true – функция завершилась успешно, и false в противном случае).

После регистрации класса окна можно переходить к созданию его экземпляра посредством вызова функции CreateWindow или CreateWindowEx. CreateWindowEx имеет больше параметров и является более новой (как WNDCLASSEX в сравнении с WNDCLASS).

Объявление функции CreateWindowEx:

```
HWND CreateWindowEx
(
     DWORD dwExStyle, // дополнительные стили
     LPCTSTR lpClassName, // имя класса
     LPCTSTR lpWindowName, // имя окна
     DWORD dwStyle, //стиль
     int x, // расстояние до окна от левой стороны экрана
     int y, // расстояние до окна от верхней стороны экрана
     int nWidth, // ширина окна
     int nHeight, // высота окна
     HWND hWndParent, // дескриптор родительского окна
     HMENU hMenu, // дескриптор меню
     HINSTANCE hInstance, // дескриптор экземпляра приложения
     LPVOID lpParam // параметры создания окна
);
```

Вот так выглядит создание окна в коде, сгенерированном компилятором CodeWarrior:

```
HWND hWnd;
hWnd = CreateWindow(/* now create the window */
                                    szProgName,
                                    "CodeWarrior Win32 stationery",
                                    WS_OVERLAPPEDWINDOW,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    CW_USEDEFAULT,
                                    (HWND)NULL,
                                    (HMENU)NULL,
                                    (HINSTANCE)hInst,
                                    (LPSTR)NULL          );
```

Функции CreateWindow и CreateWindowEx возвращают либо значение NULL, если создание окна прошло неудачно, либо дескриптор созданного окна в противном случае. Именно поэтому для хранения дескриптора окна необходимо создать переменную. Чтобы работать с окном впоследствии, следует иметь к нему доступ. Переменная создается так:

```
HWND hWnd;
```

Первый параметр функции, dwExStyle, хранит все дополнительные режимы для окна. Например, можно сделать окно всегда расположенным поверх всех окон. Подробную информацию о режимах можно

найти в документации по Windows API. Если дополнительные режимы не требуются, используйте значение NULL.

Второй параметр – имя класса, используемого для создания окна. В нашем случае – "WNDCLASS1".

Третий параметр – текстовый заголовок окна. Например, заголовком для окна компилятора CodeWarrior является текст Metrowerks CodeWarrior. В коде, сгенерированном компилятором, в качестве названия используется CodeWarrior Win32 stationery.

Четвертый параметр, dwStyle, определяет вид и поведение окна. Помните, что параметры стиля структуры WNDCLASSEX определяют вид и поведение каждого окна, созданного на базе класса, однако dwStyle определяет вид и поведение лишь одного окна. Зачем снова определять те же параметры? Дело в том, что при создании окна параметры становятся более конкретными. Некоторые из значений перечислены в табл. 12.9.

Таблица 12.9. Стили окон

Значение	Описание
WS_POPUP	Всплывающее окно
WS_OVERLAPPED	Перекрываемое окно с заголовком и границами
WS_OVERLAPPEDWINDOW	Перекрываемое окно со стилями WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_VISIBLE	Изначально видимое окно
WS_SYSMENU	Системное меню доступно в заголовке окна (по щелчку на маленькой пиктограмме). Должен использоваться стиль WS_CAPTION
WS_BORDER	Окно с тонкими границами
WS_CAPTION	Окно с заголовком (стиль WS_BORDER включается автоматически)
WS_MINIMIZE	Окно изначально свернуто
WS_MAXIMIZE	Окно изначально имеет максимальный размер

Параметры x и y определяют расстояние от левого верхнего угла экрана до левого верхнего угла окна. Чтобы оставить эти параметры на усмотрение Windows, можно воспользоваться значением CW_USEDEFAULT вместо конкретного числа.

Параметры nWidth и nHeight определяют ширину и высоту окна. Воспользуйтесь CW_USEDEFAULT, чтобы система задала их самостоятельно.

hWndParent определяет дескриптор родительского окна, если таковое существует. Значение NULL делает родительским окном рабочий стол Windows.

hMenu определяет дескриптор связанного с окном меню. Пока что используем NULL.

hInstance указывает экземпляр приложения. Воспользуйтесь аргументом hInst функции WinMain.

lpParam содержит дополнительные параметры. Пока что используем NULL.

Вот и все, что можно сказать о параметрах. Окончательный вызов функции выглядит так:

```
HWND hWnd;
hWnd = CreateWindowEx (
     NULL, //дополнительные стили
     "WINCLASS1", //имя класса
     "Простое окно", //заголовок окна
     WS_OVERLAPPEDWINDOW | WS_VISIBLE, //стили
     CW_USEDEFAULT, CW_USEDEFAULT, //начальное положение
     CW_USEDEFAULT, CW_USEDEFAULT, //начальный размер
     NULL, //родительским окном является рабочий стол
     NULL, //меню отсутствует
     hInst, //дескриптор экземпляра приложения
     NULL) //дополнительные параметры
```

Мы создали окно. Обратите внимание на использование стиля WS_VISIBLE. Он делает окно изначально видимым. В отсутствие этого стиля нам пришлось бы вызывать функцию ShowWindow, чтобы отобразить окно. Функция ShowWindow имеет два параметра: дескриптор окна и аргумент nCmdShow, полученный функцией WinMain. Вот так можно самостоятельно отобразить окно:

```
ShowWindow(hWnd, nCmdShow);
```

Возможно, вы помните, что в параметре nCmdShow Windows передает рекомендации относительно вида окна (мы рассказывали об этом в разделе «Исследуем WinMain» выше по тексту главы). Чтобы воспользоваться этими рекомендациями, обратитесь к функции ShowWindow. Предполагается, что разработчики так и должны делать, создавая окна своих приложений.

И наконец, следует вызвать функцию UpdateWindow, которая сразу же обновляет окно на экране (стандартное действие). Эта функция посылает событие WM_PAINT (о событиях вы узнаете в следующем разделе). Вызов UpdateWindow:

```
UpdateWindow();
```

Поздравляем! Вы создали свое первое настоящее окно, причем без помощи CodeWarrior.

Обработка сообщений

Следующим шагом после создания окна является обработка поступающих событий. Как вы уже знаете, для этого используется функция WndProc.

При создании приложения Windows компилятор CodeWarrior автоматически генерирует такой код:

```
/* begin the message loop */
while( GetMessage( &lpMsg, NULL, 0, 0 ) )
     {
          TranslateMessage( &lpMsg );
          DispatchMessage( &lpMsg );
     }
```

Этот код, несмотря на скромный вид, является сердцем всей программы. После создания окна программа входит в цикл `while` и находится в нем до завершения работы.

Функция, от которой зависит работа цикла, называется `GetMessage`. Она получает сообщения, подлежащие обработке, от системы Windows. Как только очередь сообщений исчерпана (программа завершена), `GetMessage` возвращает `0`, и происходит выход из цикла `while`.

Объявление функции `GetMessage`:

```
BOOL GetMessage (LPMSG lpMsg, HWND hWnd,
     UINT wMsgFilterMin, UINT wMsgFilterMax);
```

Функция заполняет структуру `lpMsg` информацией о полученном сообщении. Оставшиеся три параметра можно пока игнорировать. Они задействованы только в очень сложных программах.

В качестве первого аргумента следует передать ссылку на структуру `MSG` (спокойно, здесь нет ничего страшного). Структура `MSG`:

```
typedef struct tagMSG
{
     HWND hwnd; // окно, в котором создано сообщение
     UINT message; // идентификатор сообщения
     WPARAM wParam; // дополнительная информация
     LPARAM lParam; // дополнительная информация
     DWORD time; // время создания сообщения
     POINT pt; // координаты указателя мыши
} MSG;
```

До передачи функции `GetMessage` объект необходимо создать. На данном этапе код выглядит так:

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
}
```

У нас есть рабочий цикл обработки событий. Это очень важное обстоятельство. Теперь, когда мы получаем сообщения, нам, разумеется, захочется их обработать. Обработка выполняется при помощи функций `TranslateMessage` и `DispatchMessage`.

Аргументом каждой из них является ссылка на объект MSG. TranslateMessage производит предварительную обработку полученной информации. Пусть вас не заботят механизмы работы этой функции, просто запомните, что ее надо вызвать.

DispatchMessage посылает сообщение, подлежащее обработке, функции WndProc. Теперь код выглядит следующим образом:

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
     TranslateMessage(&msg);
     DispatchMessage(&msg);
}
```

Вкратце, это и есть обработка сообщений. Программировать для Windows не так уж и сложно, да? Достаточно просто привыкнуть к процессу.

Обработка событий

Когда сообщение (о событии) отправлено функции WndProc, его следует каким-то образом обработать. Стандартная политика обработки – создать оператор switch и заниматься различными событиями в различных вариантах (case).

Ранее в этой главе (в разделе «Исследуем WndProc») мы рассказывали, что аргумент messg функции WndProc содержит идентификатор сообщения (такой, например, как WM_PAINT). Именно на основе этого идентификатора должен происходить выбор вариантов в операторе switch. Вот так выглядит основа структуры:

```
switch (messg)
{
}
```

Теперь можно сравнивать messg с различными видами сообщений, чтобы определить, какое именно получено. Эта методика работает замечательно, но имеет один недостаток: различных видов сообщений невероятно много. Можно потратить годы на создание кода, который способен обработать все случаи, и конечный код будет иметь гигантский объем.

По счастью, можно делегировать обработку сообщений, которые вас не интересуют, системе Windows. Этой цели служит функция DefWindowProc. Ее вызов эквивалентен фразе: «Я не хочу обрабатывать это сообщение, займись этим самостоятельно. Вот вся необходимая информация». Аргументы DefWindowProc идентичны аргументам WndProc, так что их можно просто передавать напрямую. Вот так выглядит вызов функции DefWindowProc:

```
DefWindowProc( hWnd, messg, wParam, lParam );
```

И хотя большая часть сообщений, как правило, обрабатывается системой, некоторые из них мы хотели бы обрабатывать самостоятельно. Поэтому вызов функции DefWindowProc помещается в вариант по умолчанию оператора switch следующим образом:

```
switch (messg)
{
     default:
          DefWindowProc( hWnd, messg, wParam, lParam );
}
```

Возвращаемое значение DefWindowProc интерпретируется так же, как возвращаемое значение WndProc, так что это значение можно возвращать без предварительной обработки. Окончательная версия:

```
switch (messg)
{
     case WM_SOMETHING:
          //выполняем действия
     default:
          return (DefWindowProc(hWnd, messg, wParam, lParam));
}
return 0;
```

Теперь вы готовы к обработке различных сообщений.

Сообщение WM_CREATE передается при изначальном создании окна. Можно воспользоваться этим моментом и выполнить все необходимые задачи по инициализации (это официально принятая точка инициализации).

WM_DESTROY передается непосредственно перед уничтожением окна. Обычно оно означает, что приложение должно завершить работу (самостоятельно). С этой целью вызывается функция PostQuitMessage. Она посылает сообщение WM_QUIT (с которым мы можем связать дополнительный фрагмент кода) и завершает программу. Пример:

```
case WM_DESTROY:
     PostQuitMessage(0);
     break;
```

Как успеть вовремя

Часто необходимо выполнять действия в определенные моменты времени. Например, обновлять экран 26 раз в секунду. Нам повезло – в системе Windows существует удобный способ решения подобных задач.

Для этого потребуется создать *таймер*. Таймер посылает программе сообщение WM_TIMER через установленный интервал времени. Таймеров может быть несколько. Например, один с двухсекундной задержкой, а второй с трехсекундной.

Таймеры создаются посредством функции SetTimer. Функция имеет четыре аргумента. Вот ее объявление:

```
UINT SetTimer (HWND hWnd, UINT nIDevent, UINT nElapse, TIMERPROC lpTimerFunc);
```

hWnd – это дескриптор окна, с которым связан таймер. nIDevent является идентификатором таймера. Идентификатор таймера может быть произвольным целым числом. Каждый таймер имеет уникальный идентификатор. nElapse – это задержка таймера, измеряемая в миллисекундах (в одной секунде – 1000 миллисекунд). Последний параметр, lpTimerFunc, определяет имя функции, которая вызывается при срабатывании таймера. Установка параметра в значение NULL позволяет избежать вызова функции, вместо этого создается сообщение WM_TIMER.

Лучшая точка для вызова SetTimer – в коде для сообщения WM_CREATE.

После создания таймера его события можно обрабатывать в коде для сообщения WM_TIMER. Событие WM_TIMER возникает, когда срабатывает таймер. Аргумент wParam функции WndProc хранит идентификатор таймера, так что на различные таймеры можно реагировать различным образом.

И наконец, когда таймер больше не нужен (как правило, по завершении программы), его можно уничтожить вызовом функции KillTimer. Ее объявление:

```
BOOL KillTimer (HWND hWnd, UINT uIDEvent);
```

Первый аргумент хранит дескриптор окна, второй – идентификатор таймера. Вероятнее всего, вы будете вызывать эту функцию в коде обработки сообщения WM_DESTROY (непосредственно перед вызовом PostQuitMessage).

Пример применения таймера:

```
case WM_CREATE:
     SetTimer (hWnd, 1, 1000, NULL);
     break;
case WM_TIMER:
     switch (wParam)
     {
          case 1:
               //выполняем действия
               break;
     }
     break;
case WM_DESTROY:
     KillTimer (hWnd, 1);
     PostQuitMessage(0);
     break;
default:
     return (DefWindowProc(hWnd, messg, wParam, lParam));
```

Рисование в окне

Для рисования самых разнообразных вещей внутри окна можно воспользоваться сообщением WM_PAINT. Разумеется, никто не обязывает вас выполнять рисование именно по событию WM_PAINT. Рисовать можно в любой точке кода.

Событие WM_PAINT возникает при необходимости обновить окно – скажем, при изменении размеров окна, при восстановлении изображения, скрытого другим окном, или по любой другой причине.

В целях рисования можно пользоваться парой базовых функций, BeginPaint и GetDC. BeginPaint используется при рисовании по событию WM_PAINT, GetDC – для всех остальных случаев.

Обе функции создают *контекст устройства*. Контексты устройств – сложная тема. На данном этапе вам необходимо знать, что они позволяют рисовать. Без контекста рисование невозможно.

После завершения работы с контекстом устройства его следует освободить с помощью функции EndPaint (если использовалась BeginPaint) либо ReleaseDC (для GetDC).

BeginPaint принимает два параметра: дескриптор окна и ссылку на объект PAINTSTRUCT. Эта функция заполняет объект PAINTSTRUCT информацией о том, что именно необходимо перерисовать. Пусть это никого не смущает. При желании можно перерисовать окно целиком (хотя это и не очень эффективно). Вот так выглядит вызов BeginPaint:

```
HDC hDC;
PAINTSTRUCT ps;
hDC = BeginPaint(hWnd, &ps);
```

EndPaint имеет точно такие же аргументы. Пример вызова:

```
EndPaint(hWnd, &ps);
```

Параметром GetDC является дескриптор окна. Пример вызова:

```
HDC hDC;
hDC = GetDC(hWnd);
```

И наконец, ReleaseDC имеет два параметра: дескриптор окна и контекст устройства. Пример вызова:

```
ReleaseDC(hWnd, hDC);
```

А вот так может выглядеть код для обработки сообщения WM_PAINT:

```
case WM_PAINT:
     HDC hDC;
     PAINTSTRUCT ps;
     hDC = BeginPaint(hWnd, &ps);
          //рисование в окне через hDC
     EndPaint (hWnd, &ps);
     break;
```

А так выглядит рисование, выполняемое по другим событиям:

```
case WM_OTHER:
     HDC hDC;
     hDC = GetDC(hWnd);
          //рисование в окне через hDC
     ReleaseDC(hWnd, hDC);
     break;
```

Итак, с подготовкой покончено и пора поразвлечься. Займемся рисованием.

Проще всего нарисовать линию. Прежде всего, необходимо указать начальные координаты. Затем достаточно объяснить системе Windows, как рисовать линию. Пример:

```
MoveToEx(hDC,20,20,NULL);
LineTo(hDC, 20, 30);
```

Первым параметром обеих функций является контекст устройства, используемый для рисования. Следующие два параметра – координаты `x` (`x` начинается с `0` и увеличивается по мере движения к правой стороне экрана или контекста устройства) и `y` (`y` начинается с `0` и увеличивается по мере движения к нижней стороне экрана или контекста устройства). Последний аргумент функции `MoveToEx`, `NULL`, используется для хранения начальной точки (пока что оставим его в покое). Левый верхний угол окна имеет координаты (`0,0`).

При вызове функции `LineTo` происходит изменение текущих координат. При следующем вызове `LineTo` линия будет продолжена с того места, где закончилась предыдущая.

Теперь можно переходить к прямоугольникам. За рисование прямоугольников отвечает функция `Rectangle`. Ее объявление:

```
BOOL Rectangle (HDC hdc, int x1, int y1, int x2, int y2);
```

`x1` и `y1` определяют координаты левого верхнего угла, `x2` и `y2` – координаты правого нижнего угла. Пример:

```
Rectangle(hDC, 20, 20, 50, 50);
```

Эта функция рисует квадрат. Кроме того, при помощи функции `FillRect` можно нарисовать закрашенный прямоугольник. Аргументами функции `FillRect` являются не координаты прямоугольника, но указатель на структуру `RECT`. Структура `RECT` хранит четыре целочисленных значения: `top`, `left`, `right` и `bottom`. Второй параметр является дескриптором объекта `BRUSH` (который определяет стиль закрашивания прямоугольника). Пример:

```
FillRect(hDC, &rect, brush);
```

И наконец, можно нарисовать эллипс при помощи функции `Ellipse`. Необходимо передать функции координаты прямоугольника, в кото-

рый вписан эллипс. Система Windows автоматически определяет вид эллипса. Параметры функции Ellipse интерпретируются идентично параметрам функции Rectangle. Пример:

```
Ellipse (20, 20, 30, 30);
```

Этот вызов приводит к выводу окружности с центром в точке (25, 25) и радиусом 5.

Вот и все, что мы расскажем в этом ускоренном курсе рисования. Рисование в Windows не представляет никаких сложностей, достаточно просто привыкнуть к процессу.

Чтение клавиатурного ввода

С целью чтения клавиатурного ввода можно воспользоваться сообщением WM_KEYDOWN. Параметр wParam функции WndProc хранит виртуальный код, который связан нажатой клавишей. Табл. 12.10 содержит виртуальные коды клавиш стандартной клавиатуры.

Таблица 12.10. Виртуальные коды клавиш

Константа	Значение (hex)	Описание
VK_BACK	08	Backspace
VK_TAB	09	Tab
VK_RETURN	0D	Enter
VK_SHIFT	10	Shift
VK_CONTROL	11	Ctrl
VK_PAUSE	13	Pause
VK_ESCAPE	1B	Esc
VK_SPACE	20	Пробел
VK_PRIOR	21	PgUp
VK_NEXT	22	PdDn
VK_END	23	End
VK_HOME	24	Home
VK_LEFT	25	Курсор «стрелка влево»
VK_UP	26	Курсор «стрелка вверх»
VK_RIGHT	27	Курсор «стрелка вправо»
VK_INSERT	2D	Insert
VK_DELETE	2E	Delete
VK_HELP	2F	Справка
Нет кодов	30 – 39	0 – 9
Нет кодов	41 – 5A	A – Z
VK_F1 – VKF12	70 – 7B	F1 – F12

Прежде всего, следует преобразовать `wParam` в целое число:

```
int virtualCode = (int)wParam;
```

Затем в операторе `switch` по значению `virtualCode` можно определить ответные действия. Вот, собственно говоря, и все.

Рикошетирующий мяч

В этом разделе ваши познания в программировании для Windows подвергнутся суровым испытаниям! Готовы? Придется создавать окно, обрабатывать сообщения, применять таймер и рисовать на экране. Конкретная задача – создать мячик, который рикошетирует в пределах окна. Вид программы показан на рис. 12.6.

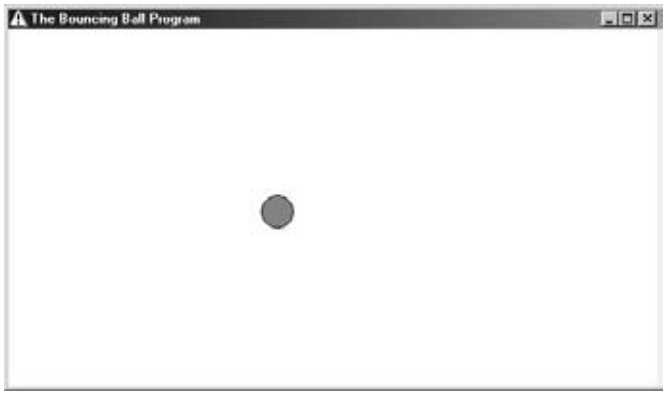


Рис. 12.6. *Готовая программа «Рикошетирующий мяч»*

Сможете ли вы одолеть эту задачу? Ниже приводится код одного из возможных решений.

```
//12.2 - Рикошетирующий мяч - Марк Ли -Premier Press
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
                        LPSTR lpszCmdLine, int nCmdShow)
{
     HWND          hWnd;
     MSG               msg;
     WNDCLASSEX     wc;

     //заполняем объект WNDCLASSEX подходящими значениями
     wc.cbSize = sizeof(WNDCLASSEX);
     wc.style = CS_HREDRAW | CS_VREDRAW;
     wc.lpfnWndProc = WndProc;
     wc.cbClsExtra = 0;
```

```
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;
    wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "BouncingBall";
    wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);

    //регистрируем новый класс
    RegisterClassEx(&wc);

    //создаем окно
    hWnd = CreateWindowEx(
        NULL,
        "BouncingBall",
        "Рикошетирующий мяч",
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInst,
        NULL
    );

    //цикл обработки событий - все сообщения
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    //стандартное возвращаемое значение
    return (msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
                                    LPARAM lParam)
{
    //статические переменные позволяют отслеживать
    //координаты мяча
    static int dX = 5, dY = 5; //хранит направление
    //хранит координаты
    static int x = 0, y = 0, oldX = 0, oldY = 0;
    //контекст устройства и кисть для рисования
    HDC hDC;
    HBRUSH brush;

    //выясняем, какое сообщение получено
    switch(nMsg)
    {
```

```
case WM_CREATE:
     //создаем таймер (0.02 секунды)
     SetTimer(hWnd, 1, 20, NULL);
     break;

case WM_TIMER: //когда таймер срабатывает (только один)
     //получаем контекст для рисования
     hDC = GetDC(hWnd);
     //белый цвет
     brush = (HBRUSH)SelectObject(hDC,
          GetStockObject(WHITE_BRUSH));

     //заполняем структуру RECT подходящими
     //значениями
     RECT temp;
     temp.left = oldX;
     temp.top = oldY;
     temp.right = oldX + 30;
     temp.bottom = oldY + 30;

     //затираем предыдущий эллипс
     FillRect(hDC, &temp, brush);
     //готовимся рисовать новый
     brush = (HBRUSH)SelectObject(hDC,
          GetStockObject(GRAY_BRUSH));
     //рисуем
     Ellipse(hDC, x, y, 30 + x, 30 + y);
     //обновляем значения
     oldX = x;
     oldY = y;
     //готовим координаты для следующего перемещения
     x += dX;
     y += dY;
     //получаем размер окна и сохраняем в rect
     RECT rect;
     GetClientRect(hWnd, &rect);
     //если окружность выходит за пределы
     //изменить направление
     if(x + 30 > rect.right || x < 0)
     {
          dX = -dX;
     }
     if(y + 30 > rect.bottom || y < 0)
     {
          dY = -dY;
     }
     //вернуть предыдущую кисть
     SelectObject(hDC, brush);
     //освободить контекст
     ReleaseDC(hWnd, hDC);
     break;

case WM_DESTROY:
```

```
                //уничтожаем таймер
                KillTimer(hWnd, 1);
                //конец программы
                PostQuitMessage(0);
                break;

            default:
                //все прочие сообщения обрабатывает Windows
                return(DefWindowProc(hWnd,
                    nMsg, wParam, lParam));
        }

        return 0;
    }
```

Резюме

Эта глава имеет особую важность, если вы хотите, чтобы программы соответствовали эпохе. Windows API существует уже более десяти лет и просуществует еще очень долго. Вы научились создавать программы для Windows, окна программ, реагировать на сообщения, создавать таймеры, рисовать, а также обрабатывать клавиатурный ввод.

Тем, кто заинтересовался и хочет узнать больше о Windows API, мы рекомендуем обратиться к следующим книгам: «Введение в программирование игр с Direct3D» (Beginning Direct3D Game Programming) Вольфганга Энгеля (Wolfgang Engel) и Эмира Гевы (Amir Geva), «Программирование изометрических игр с DirectX 7.0» (Isometric Game Programming with DirectX 7.0) Эрни Пазеры (Ernie Pazera), а также «Дао программирования игр с Direct3D» (The Zen of Direct3D Game Programming) Питера Уолша (Peter Walsh). Кроме того, рано или поздно вам придется повстречаться с гораздо более совершенной библиотекой от Microsoft – MFC (Microsoft Foundation Classes). Более подробная информация по MFC доступна в сети MSDN по адресу *http://msdn.microsoft.com/*. Обратитесь к разделу Libraries и далее по пунктам: Visual Tools and Languages, C/C++ and Visual C++, Visual C and C++ (General), Product Documentation, Visual C++ Programmer's Guide, Adding Program Functionality, Details, MFC Topics (General).

Задания

1. Создайте программу Windows, отображающую окружность, которая каждую секунду изменяет свои координаты.
2. Назовите четыре вида сообщений, которые может получать `WndProc`.
3. Каковы основные шаги в процессе создания окна?
4. Назовите по памяти пять виртуальных кодов клавиш.

13

DirectX

В этой главе читателям предстоит работать с C++ и *DirectX*, библиотекой функций программирования, созданной компанией Microsoft. В процессе вы поймете, насколько занимательно программирование с применением DirectX. Эта библиотека позволяет в процессе создания программ не задумываться о поддержке конкретных аппаратных устройств. В этой главе:

- Стандартные для индустрии игр методы разработки
- Хранение изображений в памяти
- Вывод изображений на экран
- Монтаж изображений

Составляющие DirectX

Набор инструментов разработки *DirectX* для C++ доступен на компакт-диске, прилагаемом к этой книге. Прежде чем начать программирование, следует установить программу на жесткий диск компьютера (инструкции установки приведены в приложении E «Содержимое компакт-диска»). DirectX SDK несколько отличается от той версии DirectX, что, скорее всего, установлена на вашем компьютере. Отдельные различия связаны со способом представления ошибок и способами настройки. Сообщения об ошибках примут иной вид, станут более подробными, появится возможность вызова отладки на месте. Настройки DirectX SDK можно изменять в панели управления.

Инструментарий DirectX включает семь составляющих: *DirectDraw*, *DirectSound*, *Direct3D*, *DirectInput*, *DirectPlay*, *DirectSetup*, *DirectMusic*. Разумеется, многие уже догадались, что каждая составляющая отвечает за отдельную область мультимедиа-разработки.

DirectDraw – рисование

DirectDraw предоставляет доступ к графическому аппаратному обеспечению. DirectDraw используется для хранения и вывода спрайтов, для отображения видимых составляющих видеопотоков, а также для решения всех прочих задач, связанных с выводом на экран. (*Спрайты* – это фрагменты конечного изображения; мы еще вернемся к ним во врезке «Истории из жизни».) Данные изображений могут храниться в памяти графической карты, что позволяет очень быстро с ними работать. Поскольку DirectX следует принципам объектно-ориентированного программирования, весь набор составляющих DirectDraw инкапсулирован в классе `DirectDraw`. Для взаимодействия с этим классом применяются объекты порожденных классов.

Истории из жизни

Статистика показывает, что игры, существовавшие до распространения DirectX, были несовместимы с половиной компьютеров. Библиотека DirectX решила эту проблему, предоставив разработчикам стандартный интерфейс, который позволяет программировать, не задумываясь об аппаратном обеспечении компьютера конечного пользователя. Первая версия DirectX получила название *Game Developers' Kit* (*GDK*), то есть инструментария разработчика игр. Но корпорация Microsoft быстро осознала потенциал библиотеки для разработки других мультимедиа-приложений и дала второй версии имя DirectX 2.0. DirectX по сей день сохраняет свою значимость и использовался практически в каждой из игр, которые можно найти в продаже, например «Starcraft» и «Diablo II» от Blizzard, «Baldur's Gate» от Bioware, «Age of Empires» и «Mechwarrior 4» от Microsoft, «X-Wing vs. Tie Fighter» от LucasArts.

DirectSound – звук

DirectSound предоставляет прямой доступ к звуковым аппаратным устройствам. DirectSound позволяет разработчикам игр и других мультимедиа-приложений использовать стандартные звуковые буферы, а также микшировать их для создания многоканальных звуковых эффектов. DirectSound также обеспечивает поддержку трехмерного звука. Трехмерный звук позволяет создавать пространственные звуковые эффекты, реализующие эффект присутствия.

Direct3D – новые границы мира

Трехмерные приложения – это очень здорово. Direct3D – это один из всего лишь двух существующих стандартов для создания трехмерной

графики. Вторым является *OpenGL*. Рис. 13.1 содержит снимок экрана из игры «MechWarrior 4 Vengeance», созданной компанией Microsoft с применением Direct3D. Трехмерные изображения состоят из треугольников, а для расчета отражения света используются математические модели.

Рис. 13.1. *Последний уровень игры «MechWarrior 4 Vengeance» – потрясающая трехмерная графика (все права на игру MechWarrior 4 принадлежат корпорации Microsoft)*

Истории из жизни

В играх все экраны являются результатом монтажа многих изображений. Изображения, составляющие экран, называются *спрайтами*. К примеру, в карточной игре, поставляемой в составе Windows («Пасьянс»), каждая из карт представлена отдельным спрайтом. Спрайты (карты) накладываются один на другой, и в результате получается конечное изображение, которое вы и видите, играя в эту игру.

Однако применение спрайтов не ограничено играми. На корпоративных веб-сайтах конечное изображение может состоять из ряда небольших самостоятельных изображений. Это становится особенно очевидно, если работать с модемом на скорости 28.8 (да-да, в прошлом приходилось звонить по этим штукам, чтобы получить доступ в Интернет).

DirectInput – обработка ввода

DirectInput предоставляет прямой доступ к некоторым из устройств ввода компьютера. В настоящее время поддерживаются практически все устройства ввода, включая мыши, клавиатуры, джойстики и устройства с обратной связью.

DirectMusic – развлекаем пользователя

DirectMusic предоставляет программам прямой доступ к звуковым устройствам, подобно DirectSound. Однако DirectMusic отличается от DirectSound, поскольку предназначается для работы с музыкальными файлами и более аккуратно обрабатывает все нюансы воспроизводимого звука.

DirectSetup – установка

DirectSetup – это простой способ добавлять файлы DirectX в программу установки. В сочетании с автоматическим запуском программ с компакт-дисков (AutoRun, стандартная возможность Windows) DirectSetup обеспечивает прозрачную установку нужных компонентов.

DirectPlay – возможности взаимодействия

DirectPlay предоставляет поддержку сетевого взаимодействия. Поддерживаются такие сетевые протоколы, как TCP/IP, IPX, протокол последовательного и модемного соединения. DirectPlay упрощает создание сетевых игр, самостоятельно обрабатывая низкоуровневую информацию (IP-адреса, маски подсетей и т. п.). DirectPlay позволяет создавать простейшие интерфейсы, с которыми легко и приятно работать пользователям, не знакомым с тонкостями сетевых технологий.

Подготовка к работе с DirectX

Код DirectX содержится в ряде заголовочных файлов и файле библиотеки. Чтобы начать работу с DirectX, необходимо сделать две вещи: включить файлы заголовков в программу и включить библиотечные файлы в проект.

Для DirectDraw требуется только один заголовочный файл, `ddraw.h`. Включите его стандартным способом:

```
#include <ddraw.h>
```

Затем, включите библиотечные файлы. Они расположены в подкаталоге каталога, в который установлен DirectX SDK. Я часто включаю весь набор библиотечных файлов, поскольку компилятор использует лишь те, что реально нужны; в этой главе мы используем только `ddraw.lib`.

Включить библиотечный файл можно посредством элемента Add Files выпадающего меню проекта (Project).

Итак, мы готовы отправиться в путешествие по великолепной стране графики!

Истории из жизни

В 1999 году Microsoft выпустила первые джойстики SideWinder с обратной связью. Это стало началом массового пришествия подобных устройств.

Обратная связь позволяет пользователю «чувствовать» события игры посредством джойстика, то есть служит дополнительным каналом получения игровых ощущений. Играя в симулятор воздушного боя времен второй мировой войны, вы можете видеть самолет противника и слышать звук своего пулемета. Обратная связь позволяет чувствовать отдачу пулемета с помощью джойстика. Это аналог существующей для Nintendo 64 системы тряски.

К сожалению, устройства с обратной связью пока находят применение только в играх и симуляторах, поскольку не могут быть полезны в большинстве приложений. Но в играх и симуляторах технология обратной связи – это нечто изумительное.

Архитектура DirectDraw

Архитектура DirectDraw построена из нескольких составляющих. Каждая составляющая представлена объектом. Не все объекты созданы корпорацией Microsoft, некоторая информация предоставляется разработчиками устройств. Единственное, что нам следует знать об этой информации (драйверы устройств предоставляют специальную информацию, которая позволяет DirectX работать с ними) – она существует и обрабатывается библиотекой DirectX.

Программное обеспечение, поставляемое вместе с графической картой, включает драйвер от производителя карты (скажем, весьма популярный драйвер Detonator, поставляемый Nvidia). Эти драйверы передают информацию о возможностях графической карты библиотеке DirectX.

На рис. 13.2 представлена архитектура DirectDraw в общих чертах.

Начнем с приложения Windows, которое входит в состав создаваемой программы. Программа взаимодействует с DirectDraw API (прикладным интерфейсом). Помните, что каждая из составляющих инкапсулирована. Таким образом, чтобы использовать их, необходимо применять существующие интерфейсы.

Когда DirectDraw получает определенный запрос, есть два варианта действий. Прежде всего, выполняется запрос к драйверам графической карты, позволяющий определить наличие аппаратной поддержки. Например, некоторые карты позволяют хранить в видеопамяти 32-битные палитры. Хранение палитры в видеопамяти существенно увеличивает скорость доступа к ней в сравнении со случаем оперативной памяти.

Такой прямой доступ к устройству носит название *HAL* (hardware abstraction layer, слой абстракции аппаратуры). Графическая карта определяет возможности HAL, а потому нельзя полагаться на специальные возможности некоторых устройств (таких как 32-битные текстуры). Зависимость от HAL может ограничить число компьютеров, на которых сможет работать программа. Ограничения подобных программ, использующих HAL, должны четко указываться на упаковке.

Второй вариант работы DirectDraw – *HEL* (hardware emulation layer, слой эмуляции аппаратуры). HEL подражает слою абстракции аппаратуры, искусственным образом перекладывая нагрузку на процессор и оперативную память. HEL работает медленнее, чем HAL, поскольку для эмуляции возможностей, не реализованных конкретным устройством, использует центральный процессор.

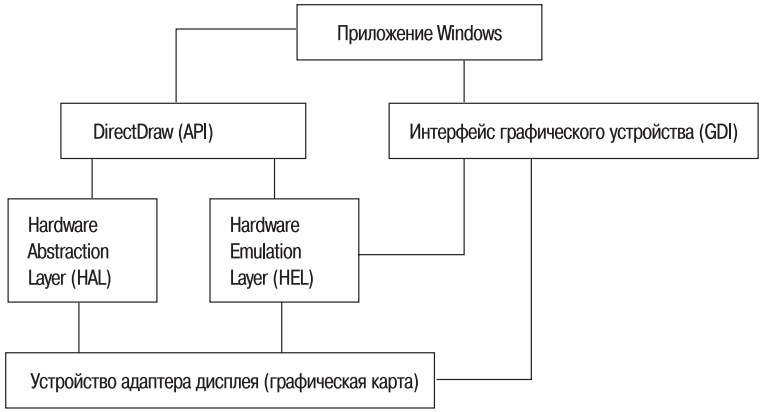


Рис. 13.2. *Архитектура DirectDraw имеет построение, позволяющее легко получать доступ к возможностям графической карты*

Интерфейсы и объекты DirectDraw

После создания приложения Win32 необходимо создать объект `DirectDraw`. Эта операция выполняется при помощи метода `DirectDrawCreate`. Вот его прототип:

```
HRESULT DirectDrawCreate(GUID FAR* lpGUID,
    LPDIRECTDRAW FAR* lplpDD, Iunknown FAR* pUnkOuter);
```

Метод DirectDrawCreate имеет три параметра:

- Указатель на GUID (globally unique identifier, глобально уникальный идентификатор), который указывает для объекта DirectDraw используемый видеодрайвер. При передаче значения NULL DirectDraw использует графическую карту по умолчанию.
- Второй параметр – адрес указателя объекта DirectDraw.
- Третий параметр должен иметь значение NULL. Он предназначен для расширения функциональности в будущих версиях библиотеки.

Функция возвращает объект HRESULT. Объектом HRESULT можно воспользоваться для определения результатов завершения операции. Если функция успешно выполнена, объект DirectDraw возвращает значение DD_OK.

Следующий фрагмент кода может использоваться для подготовки интерфейса DirectDraw:

```
LPDIRECTDRAW lpDD = NULL;
HRESULT hs;
hs= DirectDrawDrawCreate(NULL, &lpDD, NULL);
if(hs != DD_OK)
     return;
```

Объект DirectDrawSurface

Объект DirectDrawSurface представляет область памяти, используемую для хранения информации изображения. DirectDrawSurface содержит первичную плоскость, которая хранит изображение, выводимое на экран. Чтобы придать функциональность программе DirectX, следует создать объект плоскости. Объект DirectDrawSurface создается при помощи метода createSurface.

Объект DirectDrawPalette

Объект DirectDrawPallette хранит информацию о палитре для плоскостей DirectDraw. Каждый из цветов представлен тремя числами: зеленой, красной и синей составляющей. Для некоторых изображений независимое хранение палитры позволяет сократить конечный объем файла. Этот объект дает DirectDraw возможность использовать изображения, созданные с применением палитр.

Объект DirectDrawClipper

Объект DirectDrawClipper представляет прямоугольную область плоскости объекта DirectDrawSurface, которая может использоваться для рисования. Рисовать разрешено только внутри прямоугольной области, определяемой данным объектом. Для создания объекта DirectDrawClipper используется метод CreateClipper.

Объект DirectDrawVideoPort

Объект DirectDrawVideoPort позволяет DirectX направлять запросы от процессора или шины PCI к аппаратному видеопорту. Объект DirectDrawVideoPort создается при помощи метода QueryInterface и ссылки-идентификатора IID_IIDVideoPortContainer. (Об этом объекте мы не станем рассказывать из-за его сложности и ограничений объема книги.)

Способ взаимодействия программы с другими носит название *уровня сотрудничества* (*cooperative level*). Например, некоторые программы работают в полноэкранном режиме, и им требуется монопольный доступ к графическим устройствам. Другие работают в окнах наравне с другими программами Windows. А некоторые даже не позволяют использовать комбинацию клавиш <Ctrl>+<Alt>+<Delete> для перезагрузки компьютера.

Для установки уровня сотрудничества используется функция SetCooperativeLevel. Ее следует вызвать непосредственно после подготовки объекта DirectDraw. Функция имеет следующий прототип:

```
HRESULT SetCooperativeLevel (HWND hWnd, DWORD dwFlags);
```

Первый параметр – объект окна программы. (Созданием подобных объектов мы занимались в главе 12 «Программирование для Windows».)

Второй параметр является комбинацией флагов, определяющих параметры режима. Флаги описаны в табл. 13.1.

Возвращаемое значение имеет тип HRESULT и равно DD_OK, если операция выполнена успешно.

Чтобы создать полноэкранную игру, необходимо воспользоваться флагами DDSCL_FULLSCREEN и DDSCL_EXCLUSIVE.

Экранные режимы

При установке экранного режима вы передаете компьютеру характеристики экрана. Экранный режим складывается из четырех характеристик:

- Ширина
- Высота
- Глубина цвета
- Частота обновления

Ширина и высота определяют число точек по горизонтали и по вертикали экрана. Соотношение ширины и высоты должно равняться 4:3, величине для стандартных мониторов. Это соотношение называется *форматом*.

Таблица 13.1. Флаги уровней сотрудничества

Флаг	Назначение
DDSCL_ALLOWMODEX	Комбинация флагов DDSCL_FULLSCREEN и DDSCL_EXCLUSIVE. ModeX – это устаревший режим разработки игр, существующий только для обратной совместимости.
DDSCL_ALLOWREBOOT	Флаг указывает, что пользователям разрешено перезагружать компьютер с помощью комбинации клавиш <Ctrl>+<Alt>+<Delete>.
DDSCL_CREATEDEVICEWINDOW	Флаг включает поддержку работы с несколькими мониторами в DirectDraw (только для Windows 98, Windows 2000 и Windows ME/XP).
DDSCL_EXCLUSIVE	Флаг указывает, что объект DirectDraw имеет монопольный доступ к устройству. Должен использоваться совместно с флагом DDSCL_FULLSCREEN, поскольку все прочие приложения в этом режиме не имеют доступа к устройству.
DDSCL_FPUSETUP	Флаг указывает, что следует оптимизировать объект DirectDraw для работы с Direct3D.
DDSCL_FULLSCREEN	Флаг указывает, что объект DirectDraw может безопасно работать в многопоточном режиме. Многопоточный режим в настоящей книге не описан.
DDSCL_NORMAL	Флаг указывает, что объект DirectDraw не имеет особых привилегий, и должен взаимодействовать с прочими программами Windows. Данный флаг не может использоваться совместно с DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE или DDSCL_FULLSCREEN.
DDSCL_NOWINDOWCHANGES	Флаг запрещает объекту DirectDraw сворачивать и разворачивать окно приложения.
DDSCL_SETDEVICEWINDOW	Флаг указывает, что дескриптор окна, переданный в качестве первого аргумента, является дескриптором устройства окна. Флаг полезен при создании приложений DirectDraw, работающих с несколькими мониторами.
DDSCL_SETFOCUSWINDOW	Флаг указывает, что окно, дескриптор которого передается в качестве первого аргумента, имеет фокус ввода. Флаг полезен при создании приложений DirectDraw, работающих с несколькими мониторами.

Глубина цвета определяет число битов для хранения информации о цвете. Чем больше глубина цвета, тем медленнее работает программа, но тем больше цветов доступно для отображения.

Частота обновления указывает, сколько раз в секунду обновляется изображение на мониторе (в герцах). Частота обновления достаточно сильно зависит от конкретного устройства, поэтому в большинстве

случаев следует указывать нулевое значение, что является предписанием автоматического выбора частоты обновления устройством.

До появления DirectX установка экранных режимов была далеко не столь проста. Теперь она производится вызовом единственной функции:

```
HRESULT SetDisplayMode (DWORD dwWidth, DWORD dwHeight,
     DWORD dwBPP, DWORD      dwRefreshRate, DWORD dwFlags);
```

Первые два параметра определяют ширину и высоту. Третий отвечает за глубину цвета, четвертый за частоту обновления, пятый содержит флаги.

Чтобы установить экранный режим с разрешением 800 × 600 и глубиной цвета 16 бит (распространенные настройки для игр), воспользуйтесь функцией следующим образом:

```
hs = SetDisplayMode (800, 600, 16, 0, 0);
```

При успешном завершении функция возвращает значение `DD_OK`.

Первичные плоскости

Итак, объект `DirectDraw` готов для использования в приложении. Но это еще не все (разумеется, для читателей такая «новость» не станет сюрпризом). Теперь следует выделить объекту `DirectDraw` ресурсы для хранения изображений. Такими ресурсами являются первичные плоскости.

Первичные плоскости – это объекты плоскостей `DirectDraw`, под которые выделена видеопамять и которые отображаются прямо на экране. Каждая первичная плоскость обладает характеристиками экранного режима. Например, если экран работает в разрешении 640 × 480, первичные плоскости должны иметь размер 640 × 480.

Первичный и вторичный буфер

Графика должна выводиться на экран таким образом, чтобы отсутствовали нежелательные эффекты вроде мерцания и разрыва изображений. Решением этой задачи служат буферы, первичный и вторичный. Первичный и вторичный буферы составляют первичную плоскость. Пока выводится первичный буфер, программа создает следующий кадр во вторичном буфере. Такая система получила название *двойной буферизации*. По сути дела, двойная буферизация заключается в применении двух первичных плоскостей и переносе изображений из одной в другую. Схема двойной буферизации отражена на рис. 13.3.

Двойная буферизация работает потому, что объект `DirectDraw` хранит указатель на первичный буфер. Когда вторичный буфер готов к отоб-

ражению на экран, DirectDraw просто меняет значение указателя. Эта операция известна как *переброска* (*flipping*) *буферов*.

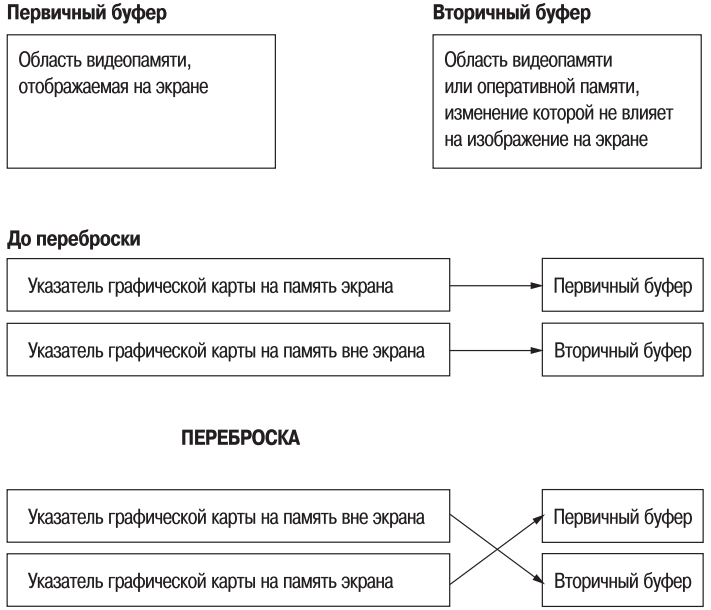


Рис. 13.3. *Концептуальная схема двойной буферизации*

Создание плоскостей

Первым шагом при создании плоскости является описание ее параметров. Для указания параметров служит объект DDSURFACEDESC2, спроектированный специально с этой целью. Объект DDSURFACEDESC2 хранит следующую информацию:

- Возможности плоскости
- Назначение плоскости
- Поведение плоскости

Структура DDSURFACEDESC2 для плоскости служит тем же целям, что и структура WNDCLASSEX для класса окна (более подробная информация о WNDCLASSEX содержится в главе 12). Объект заполняется подходящей информацией и передается функции. Для создания плоскости используется функция CreateSurface (вместо RegisterClassEx и CreateWindowEx для окон).

И наконец, когда первичная плоскость создана, необходимо извлечь связанный с ней вторичный буфер посредством вызова GetAttachedSurface. Когда все эти шаги выполнены, можно начинать рисование в DirectX.

Инициализация структуры

Первый шаг в создании плоскости – заполнение структуры DDSURFACEDESC2 подходящей информацией о плоскости. Взгляните на объявление этой структуры:

```
typedef struct _DDSURFACEDESC2 {
     DWORD     dwSize;
     DWORD     dwFlags;
     DWORD     dwHeight;
     DWORD     dwWidth;
     union
     {
          LONG     lPitch;
          DWORD     dwLinearSize;
     } DUMMYUNIONNAMEN(1);
     DWORD     dwBackBufferCount;
     union
     {
          DWORD     dwMipMapCount;
          DWORD     dwRefreshRate;
     } DUMMYUNIONNAMEN(2);
     DWORD     dwAlphaBitDepth;
     DWORD     dwReserved;
     LPVOID     lpSurface;
     union
     {
          DDCOLORKEY     ddckCKDestOverlay;
          DWORD     dwEmptyFaceColor;
     } DUMMYUNIONNAMEN(3);
     DDCOLORKEY     ddckCKDestBlt;
     DDCOLORKEY     ddckCKSrcOverlay;
     DDCOLORKEY     ddckCKSrcBlt;
     DDPIXELFORMAT     ddpfPixelFormat;
     DDSCAPS2     ddsCaps;
     DWORD     dwTextureStage;
} DDSURFACEDESC2, FAR* LPDDSURFACEDESC2;
```

Структура может показаться сложной, но на деле следует ознакомиться лишь с некоторыми ее полями, о которых мы сейчас и поговорим. Но прежде следует создать экземпляр этой структуры. Пример создания объекта DDSURFACEDESC2:

```
DDSURFACEDESC2 ddsd;
```

Первое важное поле – dwSize. Оно хранит размер структуры. Может показаться странной идея хранения размера объекта, но если не заполнить это поле, могут возникать ошибки. Инициализация поля dwSize весьма проста. Просто присвойте значение sizeof(ddsd). Например, так:

```
ddsd.dwSize = sizeof(ddsd);
```

Следующее поле, о котором стоит рассказать, – dwFlags. Оно определяет набор других полей, которые будут использоваться. DirectX будет работать только с информацией, описанной значением этого поля. Некоторые из существующих значений перечислены в табл. 13.2. Они могут комбинироваться при помощи оператора «или» (|).

Таблица 13.2. Возможные значения dwFlags

Значение	Описание
DDSD_ALL	Доступны все поля ввода
DDSD_ALPHABITDEPTH	Доступно поле dwAlphaBitDepth
DDSD_BACKBUFFERCOUNT	Доступно поле dwBackBufferCount
DDSD_CAPS	Доступно поле ddsCaps
DDSD_CKDESTBLT	Доступно поле ddckCKDestBlt
DDSD_CKDESTOVERLAY	Доступно поле ddckCKDestOverlay
DDSD_CKSRCBLT	Доступно поле ddckCKSrcBlt
DDSD_CKSROVERLAY	Доступно поле ddckCKSrcOverlay
DDSD_HEIGHT	Доступно поле dwHeight
DDSD_LINEARSIZE	Доступно поле dwLinearSize
DDSD_LPSURFACE	Доступно поле lpSurface
DDSD_MIPMAPCOUNT	Доступно поле dwMipMapCount
DDSD_PITCH	Доступно поле lPitch
DDSD_PIXELFORMAT	Доступно поле ddpfPixelFormat
DDSD_REFRESHRATE	Доступно поле dwRefreshRate
DDSD_TEXTURESTAGE	Доступно поле dwTextureStage
DDSD_WIDTH	Доступно поле dwWidth

На данном этапе нас с вами интересуют только два значения: DDSD_CAPS и DDSD_BACKBUFFERCOUNT. Инициализируем поле dwFlags следующим образом:

```
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
```

Поле ddsCaps является объектом типа DDSCAPS2 и хранит информацию о возможностях плоскости. Единственное поле ddsCaps, которое должно вас интересовать, это dwCaps. Поле dwCaps структуры DDSCAPS2 хранит флаги возможностей плоскости. Некоторые из возможных значений поля перечислены в табл. 13.3.

Чтобы создать первичную плоскость с закрепленным за ней вторичным буфером, воспользуйтесь значениями DDSCAPS_PRIMARYSURFACE, DDSCAPS_COMPLEX и DDSCAPS_FLIP. Вот так выглядит указание значения:

```
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDDCAPS_COMPLEX | DDSCAPS_FLIP;
```

Таблица 13.3. Возможности плоскостей

Значение	Описание
DDSCAPS_PRIMARYSURFACE	Плоскость является первичной
DDSCAPS_BACKBUFFER	Плоскость является вторичным буфером для первичной плоскости
DDSCAPS_FLIP	Плоскость может перебрасываться (более подробная информация о переброске – ниже по тексту главы в разделе «Рисуем на экране»)
DDSCAPS_COMPLEX	Плоскость входит в состав сложной цепи переброски, с ней связан целый ряд плоскостей

И наконец, необходимо инициализировать поле dwBackBufferCount. Оно хранит число вторичных буферов, закрепленных за плоскостью. В обычной цепи переброски только один вторичный буфер. Устанавливаем значение следующим образом:

```
ddsd.dwBackBufferCount = 1;
```

Вот и все. Структура DDSURFACEDESC2 инициализирована в достаточной степени, чтобы можно было создать рабочую плоскость. Взгляните на окончательный вариант кода:

```
DDSURFACEDESC2 ddsd;
ZeroMemory(&ddsd,sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_COMPLEX | DDSCAPS_FLIP;
ddsd.dwBackBufferCount = 1;
```

Функция ZeroMemory обнуляет все ячейки памяти, выделенной под объект ddsd, что является разумным способом инициализации объектов.

Создание плоскости

Следующий логический шаг – создание плоскости. С этой целью необходимо вызвать метод CreateSurface. Этот метод является компонентом объекта DIRECTDRAW7. Взгляните на объявление метода:

```
HRESULT CreateSurface (LPDDSURFACEDESC2 ddsd,
     LPDIRECTDRAWSURFACE7 FAR *lplpDDSurface, IUnknown FAR* pUnkOuter);
```

Первый параметр является указателем на объект DDSURFACEDESC2, второй – указателем на объект LPDIRECTDRAWSURFACE7 (ради этого значения и вызывается метод), а третий в настоящее время не используется и должен иметь значение NULL. Таким образом, следует создать объект LPDIRECTDRAWSURFACE7 и вызвать функцию. Вот как это может выглядеть:

```
LPDIRECTDRAWSURFACE7 lpPrimary;
g_pDD->CreateSurface(&ddsd, &lpPrimary, NULL);
```

Мы получили плоскость быстрого приготовления. CreateSurface возвращает DD_OK в случае успешного завершения.

Доступ ко вторичному буферу

На данном этапе у нас есть первичная плоскость и вторичный буфер. Последний шаг – получение указателя для доступа ко вторичному буферу. С этой целью применяется метод GetAttachedSurface, который является компонентом класса DIRECTDRAWSURFACE7. Прототип этого метода:

```
HRESULT GetAttachedSurface (LPDDSCAPS2 lpDDSCaps,
     LPDIRECTDRAWSURFACE7 FAR* lplpDDAttachedSurface);
```

Первый параметр является указателем на объект DDSCAPS2, описывающий возможности закрепленной плоскости. Второй параметр – указатель на закрепленную плоскость (ради этого значения и вызывается метод). Чтобы вызвать метод, следует инициализировать объект DDSCAPS2 подходящими значениями. Пример:

```
DDSCAPS2 ddsd;
LPDIRECTDRAWSURFACE7 lpBack;
ZeroMemory(&ddsc,sizeof(ddsc));
ddsc.dwCaps = DDSCAPS_BACKBUFFER;
g_pddsprimary->GetAttachedSurface(&ddsc,&lpBack);
```

Мы готовы к рисованию на плоскостях.

Рисуем на экране

Рисование на экране следует определенным правилам. Прежде всего, необходимо подготовить вторичный буфер. Следует инициализировать его в точности тем изображением, которое необходимо показать на экране. Затем выполняется переброска первичной плоскости и вторичного буфера. Изображение будет отображено на экране мгновенно.

Для подготовки вторичного буфера обычно используется метод Blt (читается «блит»). Метод Blt позволяет решать разнообразные задачи, включая копирование одной плоскости в другую, заполнение плоскости цветом, перенос изображения в определенную область плоскости. Взгляните на прототип метода Blt:

```
HRESULT Blt (LPRECT lpDestRect, LPDIRECTDRAWSURFACE7 lpDDSrcSurface,
     LPRECT lpSrcRect, DWORD dwFlags, LPDDBLTFX lpDDBltFX);
```

Первый параметр является указателем на структуру RECT, которая определяет область плоскости для обработки. Полная плоскость задается значением NULL.

lpDDSrcSurface – это объект LPDIRECTDRAWSURFACE7, который должен служить источником операции. Если нет необходимости копировать информацию с другой плоскости, этот параметр может иметь значение NULL.

Третий параметр определяет область плоскости-источника для работы. Чтобы использовать полную плоскость (или не использовать плоскость вовсе), передайте значение NULL.

Четвертый параметр определяет, какие поля lpDDBltFX используются, а также содержит иные параметры операции. Некоторые из значений этого поля перечислены в табл. 13.4.

Если нет необходимости использовать стили, поле может иметь значение NULL.

Таблица 13.4. Возможные стили выполнения Blt

Значение	Описание
DDBLT_COLORFILL	Использует поле dwFillColor структуры DDBLTFX в качестве RGB-значения цвета заливки прямоугольной области целевой плоскости.
DDBLT_DDFX	Использует поле dwDDFX структуры DDBLTFX для определения эффектов операции.
DDBLT_WAIT	Откладывает возврат значения DDERR_WASSTILLDRAWING, если механизм Blt занят выполнением других операций, и возвращает его после завершения операции либо при возникновении ошибки.

Пятый параметр является указателем на объект DDBLTFX, который содержит параметры операции. Структура DDBLTFX состоит из большого числа полей, а нас, по счастью, интересуют лишь два из них: dwSize и dwFillColor.

dwSize – это размер объекта. dwFillColor определяет цвет заливки для плоскости (если выполняется заливка). Каждое число соответствует уникальному цвету. Поэкспериментируйте с различными значениями и выясните, какие цвета можно получить. Черному цвету соответствует нулевое значение.

Пример вызова Blt для вторичного буфера:

```
DDBLTFX ddbltfx;
ZeroMemory(&ddbltfx,sizeof(ddbltfx));
ddbltfx.dwSize = sizeof(ddbltfx);
ddbltfx.dwFillColor = 2;
lpBackBuffer->Blt(NULL,NULL,NULL, DDBLT_COLORFILL, &ddbltfx);
```

После подготовки вторичного буфера необходимо выполнить переброску для первичной плоскости и вторичного буфера. Вторичный буфер при этом становится первичной плоскостью, а первичная плоскость – вторичным буфером. Переброска выполняется при помощи метода Flip. Его прототип:

```
HRESULT Flip (LPDIRECTDRAWSURFACE7 lpSurfaceOverride, DWORD dwFlags);
```

Первый параметр указывает плоскость из цепи переброски. Чтобы воспользоваться стандартным механизмом переброски, укажите значение NULL.

Второй параметр позволяет указывать специальные параметры переброски. Если в этом нет необходимости, передайте нулевое значение.

Вот пример выполнения этой операции для плоскостей:

```
lpPrimary->Flip(NULL,0);
```

Для отображения картинки на экране требуется лишь один вызов функции – проще некуда.

Растровые изображения

Существует возможность загружать файлы изображений в видеопамять. Это позволяет рисовать изображения заранее, сохранять их на диске, а затем отображать по мере необходимости. Одним из распространенных форматов графических файлов является растровый. Растровые файлы .bmp имеют относительно простой формат и могут быть прочитаны практически на всех компьютерах.

Существует два способа загрузки растрового изображения в плоскость. Первый способ – создать временную плоскость, скопировать изображение на нее, а затем скопировать эту плоскость в другую. Второй способ – загрузить изображение в структуру HBITMAP, а затем скопировать изображение в целевую плоскость.

В обоих случаях необходимо включить файл ddutil.h (файл доступен на компакт-диске книги). Файл ddutil.h облегчает работу с растровыми изображениями. В первом случае необходимо создать плоскость, вызвать функцию DDLoadBitmap для загрузки изображения, а затем выполнить «блит» для копирования одной плоскости в другую. DDLoadBitmap имеет четыре параметра: объект DirectDraw, имя файла изображения (включающее полный путь к нему), ширину и высоту изображения. Функция возвращает созданную плоскость с изображением. Пример вызова:

```
lpTempSurface = DDLoadBitmap(g_pDD, "myImage.bmp", 800,600);
```

Второй способ отобразить растровый файл на экране: вызвать функцию Windows LoadImage, а затем функцию DDCopyBitmap для копирования изображения в плоскость. LoadImage имеет шесть параметров. Первый параметр должен иметь значение NULL, второй является полным именем файла, третий должен иметь значение IMAGE_BITMAP, четвертый и пятый параметры определяют высоту и ширину, а шестой должен иметь значение LR_LOADFROMFILE | LR_CREATEDIBSECTION. LoadImage возвращает объект HBITMAP, представляющий изображение. Пример:

```
HBITMAP temp = (HBITMAP)LoadImage(NULL, "myImage.bmp",
```

```
IMAGE_BITMAP, 800,600, LR_LOADFROMFILE | LR_CREATEDIBSECTION);
```

Обратите внимание, что возвращаемое значение необходимо приводить к типу HBITMAP. Дело в том, что функция LoadImage работает со многими видами изображений, а не только с форматом bmp. Получив в свое распоряжение объект HBITMAP, мы можем вызвать DDCopyBitmap, чтобы скопировать изображение в плоскость. DDCopyBitmap имеет шесть параметров. Первый определяет целевую плоскость, второй является объектом HBITMAP, третий и четвертый определяют начальные координаты в изображении, с которых начинается копирование (обычно 0,0), а пятый и шестой определяют ширину и высоту растрового изображения (отсчет от начальных координат). Пример:

```
DDCopyBitmap(lpBack,temp,0,0,800,600);
```

Вот и все, что нужно для копирования изображения на экран. Немного практики – и все будет в порядке.

Пишем программу «Случайный цвет»

В этом разделе мы займемся созданием программы, которая отображает на экране случайный цвет каждые три секунды. Сможете ли вы угадать, какой цвет будет следующим? Эта программа испытает ваши познания в DirectX и иллюстрирует интеграцию программ Windows с библиотекой DirectX.

```
//13.1 - Программа "Случайный цвет" - Марк Ли - Premier Press
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <ddraw.h>

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
     LPARAM lParam);

LPDIRECTDRAW7 g_pdd; // объект DirectDraw
LPDIRECTDRAWSURFACE7 g_pddsprimary; //первичная плоскость
LPDIRECTDRAWSURFACE7 g_pddsback; //вторичный буфер
LPDIRECTDRAWSURFACE7 g_pddsone; //временная плоскость
DDSURFACEDESC2 ddsd; //используется для описания плоскостей
DDSCAPS2 ddsc; //хранит возможности плоскости
//используется для временного хранения результатов выполнения функции
HRESULT hRet;


int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
     LPSTR lpszCmdLine, int nCmdShow)
{
     HWND hWnd;
     MSG msg;
     WNDCLASSEX wc;
```

```
    //инициализируем структуру WNDCLASSEX
    //подходящими значениями
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInst;
    wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "RandomColor";
    wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);

    //регистрируем новый класс
    RegisterClassEx(&wc);

    //создаем окно
    hWnd = CreateWindowEx(
        NULL,
        "RandomColor",
        "Случайный цвет",
        WS_VISIBLE,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInst,
        NULL
    );

    //цикл обработки событий - диспетчер сообщений
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    //стандартное возвращаемое значение
    return (msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam,
    LPARAM lParam)
{

    //определяем, какое получено сообщение
    switch(nMsg)
    {
        case WM_CREATE:
```

```
        //создаем таймер (3 секунды)
        SetTimer(hWnd, 1, 3000, NULL);
        //создаем объект DirectDraw
        hRet = DirectDrawCreateEx(NULL,
             (void**)&g_pdd,IID_IDirectDraw7, NULL);
        if(hRet != DD_OK)
             MessageBox(hWnd,
                  "Невозможно выполнить DirectDrawCreateEx",
                  "Ошибка", NULL);
        //Установка уровня сотрудничества
        hRet = g_pdd->SetCooperativeLevel(hWnd,
             DDSCL_FULLSCREEN | DDSCL_EXCLUSIVE);
        if(hRet != DD_OK)
             MessageBox(hWnd,
                  "Невозможно выполнить SetCooperativeLevel",
                  "Ошибка", NULL);
        //Экранный режим: 800x600
        //16 битов на точку
        hRet = g_pdd->SetDisplayMode(800, 600, 16,0,0);
        if(hRet != DD_OK)
             MessageBox(hWnd, "Невозможно выполнить SetDisplayMode",
             "Ошибка", NULL);

        //подготовка параметров первичной плоскости
        ZeroMemory(&ddsd,sizeof(ddsd));
        ddsd.dwSize = sizeof(ddsd);
        ddsd.dwFlags =
             DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
        ddsd.dwBackBufferCount = 1;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
             DDSCAPS_FLIP | DDSCAPS_COMPLEX;

        //создаем плоскость
        hRet = g_pdd->CreateSurface(&ddsd,
             &g_pddsprimary, NULL);
        if (hRet != DD_OK)
             MessageBox(hWnd, "Невозможно выполнить CreateSurface",
                  "Ошибка", NULL);

        //подготовка параметров вторичного буфера
        ZeroMemory(&ddsc,sizeof(ddsc));
        ddsc.dwCaps = DDSCAPS_BACKBUFFER;

        //получаем указатель на вторичный буфер
        hRet = g_pddsprimary->GetAttachedSurface(&ddsc,
             &g_pddsback);
        srand(time(0));
        break;

   case WM_TIMER: //таймер сработал (только один)
        DDBLTFX ddbltfx;
        ZeroMemory(&ddbltfx,sizeof(ddbltfx));
        ddbltfx.dwSize = sizeof(ddbltfx);
```

```
                ddbltfx.dwFillColor = rand()%16;
                g_pddsback->Blt(NULL,NULL,NULL,
                     DDBLT_COLORFILL, &ddbltfx);
                g_pddsprimary->Flip(NULL,0);
                break;

           case WM_DESTROY:
                //уничтожаем таймер
                KillTimer(hWnd, 1);
                //завершаем программу
                PostQuitMessage(0);
                break;

           default:
                //остальные сообщения оставляем системе Windows
                return(DefWindowProc(hWnd, nMsg, wParam,
                     lParam));
      }

      return 0;
}
```

Резюме

В этой главе вы изучили основы работы с DirectX. Используя полученные знания, вы встали на верный путь к созданию игр высокого качества, соответствующих высочайшим стандартам производителей игр. Но это еще не все. Вы получили опыт использования библиотеки разработки, что невероятно важно, поскольку программист в своей деятельности имеет дело с большим числом библиотек. Внемли, благородный читатель, в главе 14 «Создаем пиратское приключение» все эти знания подвергнутся самым серьезным испытаниям.

Задания

1. Приведите два способа отображения растрового bmp-файла на экране.
2. Назовите семь составляющих библиотеки DirectX.
3. Перечислите шаги создания плоскостей DirectDraw.
4. Что представляет первичная плоскость?

14

Создаем пиратское приключение

Последняя глава! Любезный читатель, если ты читал книгу с начала, то проделал долгий путь. Здесь мы расскажем, как применить информацию из других глав для создания простого механизма компьютерной игры. Будет использоваться DirectX, изученный нами в главе 13 «DirectX»; система Microsoft Windows, о которой шла речь в главе 12 «Программирование для Windows»; а также большая часть информации по C++ из прочих глав книги. Именно вам, новым программистам, предстоит дорабатывать этот код в соответствии со своими желаниями в целях создания полноценной игры. Желаем удачи!

В этой главе:

- Глобальные переменные механизма игры
- Класс `Ship`, представляющий корабли
- Основная программа
- Перемещение корабля по экрану
- Отображение событий игры на экране
- Остановки в городах

Обзор игры

При разработке игр мы советуем в первую очередь создавать графику, поскольку графика способствует приданию формы игре. (Да и если подумать, что более увлекательно – программировать на протяжении шести часов, разглядывая одинокую надпись «Здесь будет вступительная заставка» или разглядывая замечательную заставку, созданную своими руками?) Имея в своем распоряжении элементы графики, можно их использовать для проверки работоспособности кода.

Для игры «Пиратское приключение» мы используем 11 экранов: заставку (которая отображается до начала игры), картину города, в который входит игрок, и девять экранов карт для игры.

Основа игрового механизма – карта, которой пользуется игрок, чтобы перемещаться по бассейну Карибского моря. Полная карта приведена на рис. 14.1. Карта поделена на шесть квадратов (сетка размером 3 × 3). Каждый элемент имеет размер компьютерного экрана (800 × 600 точек). Каждый из девяти фрагментов хранится в отдельном растровом графическом файле (.bmp). Файлы имеют имена от map1.bmp до map9.bmp и показаны соответственно на рис. с 14.2 по 14.10.

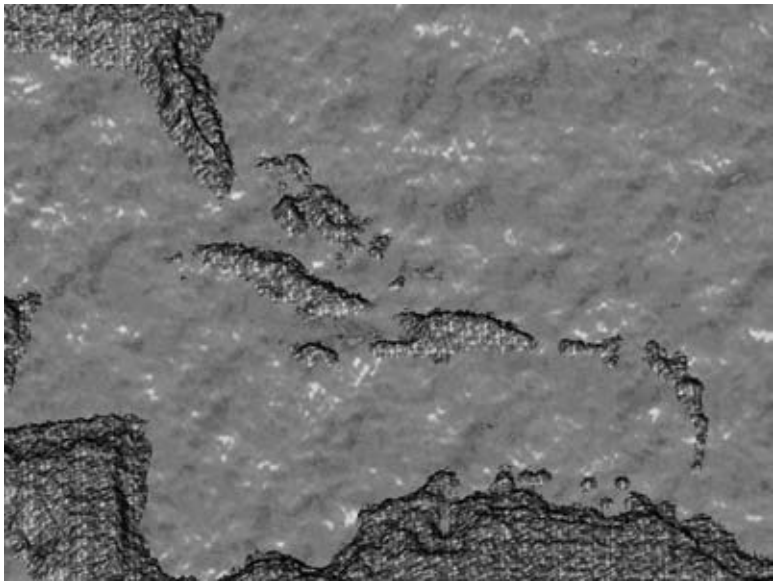


Рис. 14.1. *Полная карта игры. Каждый экран содержит одну девятую часть этого изображения (map.bmp)*

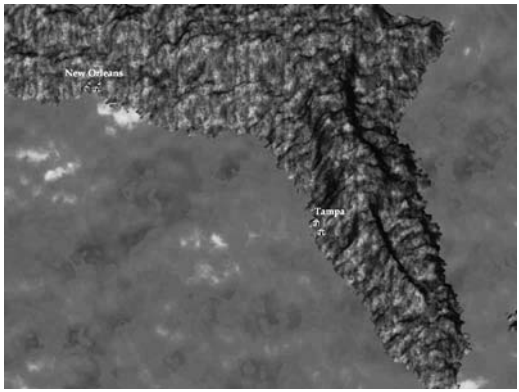


Рис. 14.2. *Северо-западный угол карты – южная часть США, включая Флориду (map1.bmp)*

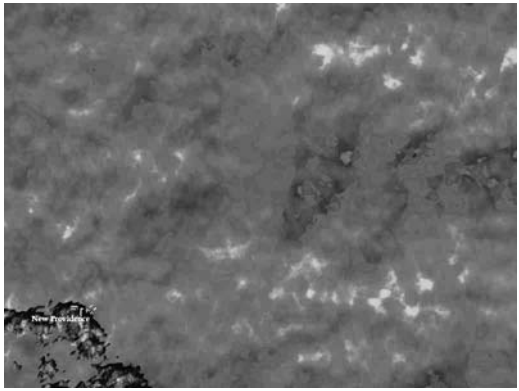


Рис. 14.3. *Преимущественно морской район, внизу фрагмента расположен Нью-Провиденс. Верхняя средняя часть карты (map2.bmp)*

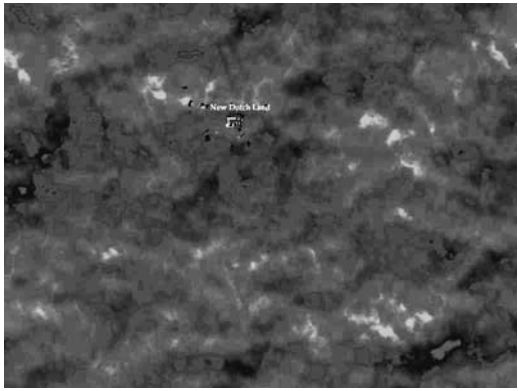


Рис. 14.4. *Нью-Дачленд (вымышленный город) изолирован в центре верхней правой части карты (map3.bmp)*

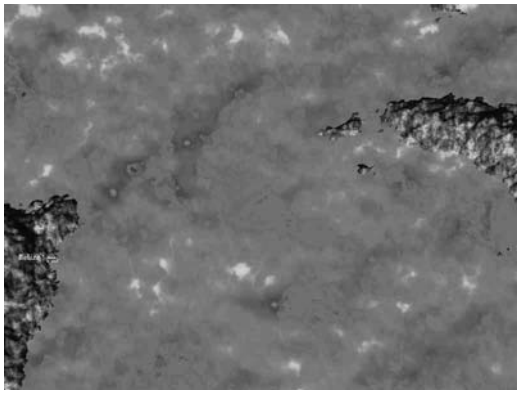


Рис. 14.5. *Под картой 1 расположен фрагмент карты с городом Белиз (map4.bmp)*

Игра должна выглядеть профессионально, а значит, «Пиратскому приключению» нужен экран-заставка. Как правило, он содержит название игры (в большинстве игр стартовый экран позволяет начать новую игру). Перед началом игры «Пиратское приключение» в течение пяти секунд отображается заставка (рис. 14.11).

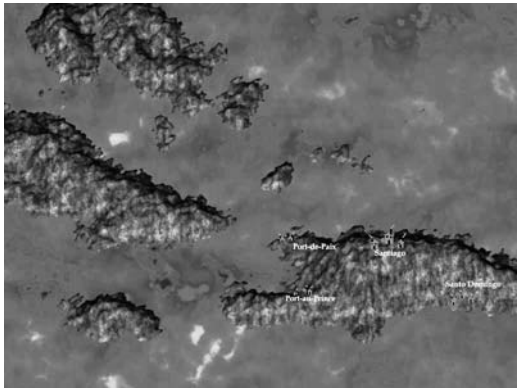


Рис. 14.6. *В самом центре карты – процветающий район с большим числом городов (map5.bmp)*

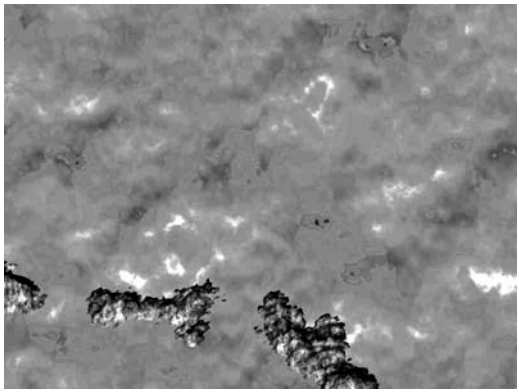


Рис. 14.7. *Непосредственно под картой 3 и рядом с картой 5 – пустынный район (map6.bmp)*

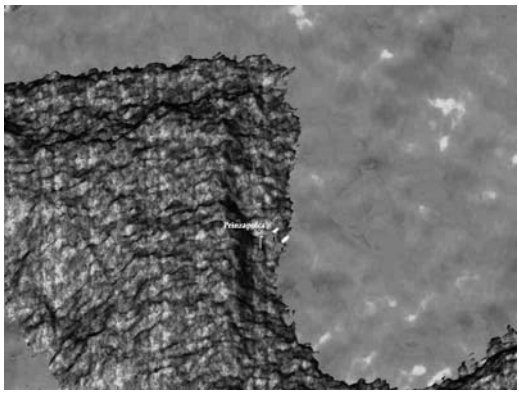


Рис. 14.8. *Нижняя левая часть карты – много земли и всего один город (map7.bmp)*

И наконец, экран города (рис. 14.12) отображается, когда игрок входит в город. Это вносит некоторое разнообразие в игру. Не стоит всю игру показывать игроку один и тот же экран.

Теперь, разобравшись с картинками, мы можем заняться графикой кораблей. В этой игре будет только один корабль (корабль игрока), но

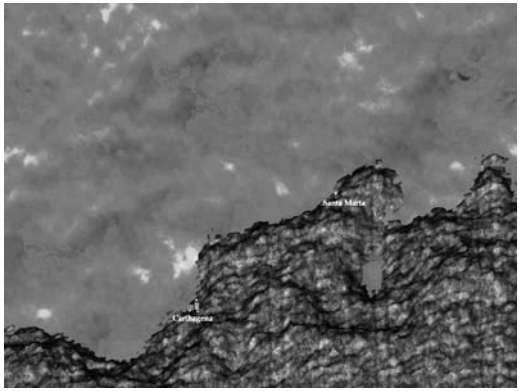


Рис. 14.9. *Нижняя средняя часть карты – города Санта-Мария и Картахена (map8.bmp)*

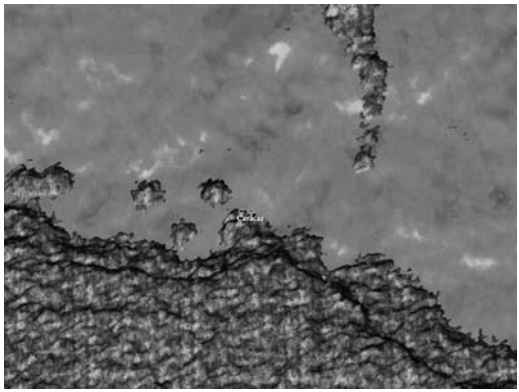


Рис. 14.10. *Нижний правый угол – один-одинешенек город Каракас (map9.bmp)*

нет ничего проще, чем повторно использовать существующую графику и создать на экране второй корабль. Поскольку корабль может двигаться только в четырех направлениях, понадобится лишь четыре изображения корабля, по одному на каждое направления. К счастью, эти изображения достаточно маленькие, и их можно сохранить в одном изображении. Когда понадобится одна из частей этого изображения, мы легко ее скопируем.

Каждое изображение корабля называется *спрайтом* (все спрайты показаны на рис. 14.13). Корабль будет отображаться поверх экрана карты, а значит, фон спрайтов должен быть прозрачным. С этой целью необходимо выбрать такой цвет фона, который в игре не используется для других целей. Затем достаточно сообщить DirectX, что этот цвет является прозрачным, установив цветовой ключ.

Механизм игры

С графикой разобрались, начинаем программировать. Исходные тексты механизма игры приведены в последующих разделах, начиная с «Глобальных структур» и заканчивая «Посещением городов».

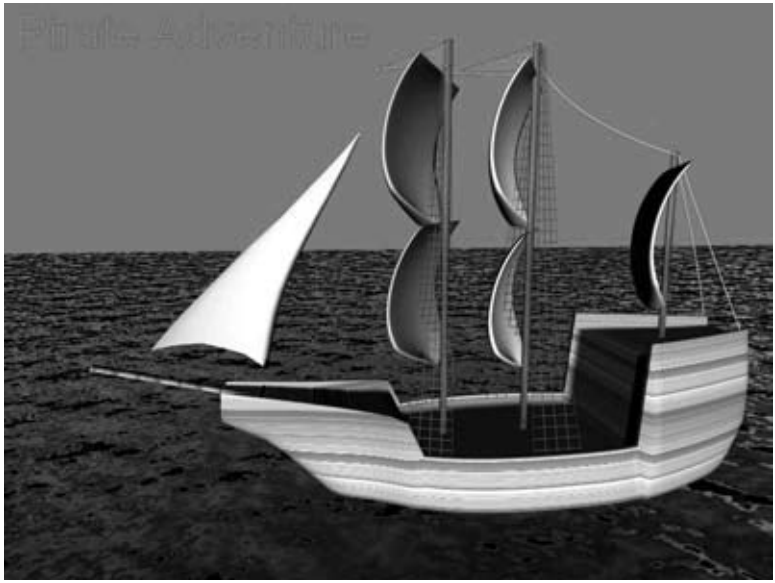


Рис. 14.11. *Заставка игры «Пиратское приключение» во всей своей красе (Screen.bmp)*

Рис. 14.12. *Вот так выглядит экран города (cityScreen.bmp)*

***Рис. 14.13.** Спрайты корабля будут использованы для индикации направления движения (sprites.bmp)*

Глобальные структуры

Первый файл – globals.h. Он содержит некоторые из объявлений глобальных переменных и функций (глобальные переменные описаны в главе 4 «Пишем функции»). Функции, объявленные в этом файле, являются функциями Windows. Реализации функций даны в следующем файле, globals.cpp.

```
//14.1 - globals.h - Марк Ли - Premier Press

#ifndef GLOBALS_H
#define GLOBALS_H //включать файл не более одного раза

#include "Ship.h"

#include <windows.h>

#include "ddutil.h"
#include "drawing.h"
#include "Movement.h"
#include "Towns.h"

//удобные константы
#define OpeningScreen "OpeningScreen.bmp"
#define CityScreen "city.bmp"
#define OpeningTimer 1
#define MainTimer 2
```

```
//глобальные переменные
//сохраняют направление движения корабля
BOOL moveUp, moveDown, moveLeft, moveRight,fire;
Ship* ship; //корабль игрока
int currentMapX; //текущая карта в сетке 3x3  (0,1 или 2)
int currentMapY; //вторая координата
HBITMAP entireMap[3][3];//все карты (9 секций)

//выполнить цикл событий из WinMain
int DoEventLoop();
//создать окно
void InitApp(HINSTANCE hInst,int nCmdShow);
//обработать сообщения
LRESULT CALLBACK WndProc( HWND hWnd, UINT messg,
     WPARAM wParam, LPARAM lParam );
#endif

//14.2 - globals.cpp - Марк Ли - Premier Press
#include "globals.h"

int DoEventLoop()
//цикл событий из WinMain
{
     MSG msg;

     //цикл событий - обработать все сообщения
     while(GetMessage(&msg, NULL, 0, 0))
     {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     //стандартное возвращаемое значение
     return (msg.wParam);
}

void InitApp(HINSTANCE hInst,int nCmdShow)
//создает окно
{
     HWND          hWnd;
     WNDCLASSEX     wc;

     //заполняем структуру WNDCLASSEX
     //подходящими значениями
     wc.cbSize = sizeof(WNDCLASSEX);
     wc.style = NULL;
     wc.lpfnWndProc = WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInst;
     wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
     wc.lpszMenuName = NULL;
```

```
    wc.lpszClassName = "PiratesAdventure";
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    //регистрируем новый класс
    RegisterClassEx(&wc);

    //создаем окно
    hWnd = CreateWindowEx(
        NULL,
        "PiratesAdventure",
        "Пиратское приключение",
        WS_POPUP,
        0,
        0,
        800,
        600,
        NULL,
        NULL,
        hInst,
        NULL
    );

    //загружаем карты
    LoadMaps();
}
```

Как видите, обработкой событий Windows занимается функция `DoEventLoop`. Точно такой же код мы приводили в главе 12.

`InitApp` создает и отображает окно. Окно полноэкранное, черное и не имеет строки заголовка. Для загрузки экранов карт вызывается функция `LoadMaps` из файла Drawing.h.

Директивы `#ifndef`, `#define` и `#endif` нужны, чтобы избежать повторного включения файла globals.h.

Создание класса Ship

Следующий файл – Ship.h. Это файл объявления класса `Ship` (классы описаны в главе 5 «Боевые качества ООП»). Реализация класса содержится в файле Ship.cpp.

```
//14.3 - Ship.h - Марк Ли - Premier Press
#ifndef SHIP_H
#define SHIP_H

#include "ddraw.h"
class Ship // класс Ship - хранит корабль игрока
{
    RECT position;//позиция на сетке 3x3 (максимальное значение (2399,1799))
    LPDIRECTDRAWSURFACE7 c_pddsone; //хранит растровый спрайт
    int speed; //скорость корабля
    int direction; //определяет, какой спрайт использовать
```

```
     //хранит расположение спрайтов
     //в растровом изображении (4 направления = 4 спрайта)
     RECT sprites[4];
public:
     Ship();
     void Move(int direction);//перемещает корабль
     BOOL isCollision(int, int);//проверка столкновения
     void moveBack(); //движение в обратном направлении
     RECT GetPosition() {return position;}
     int GetDirection() {return direction;}
     void Draw();//отображает экран игры

};
#endif
//14.4 - Ship.cpp - Марк Ли - Premier Press
#include "ddraw.h"
#include "Ship.h"
#include "globals.h"

Ship::Ship()
//создает объект корабля
{
     //загружаем спрайты
     c_pddsone = DDLoadBitmap(g_pdd, "sprites.bmp",800,600);
     //делаем фон спрайта прозрачным
     DDSetColorKey(c_pddsone,RGB(0,255,0));
     //вычисляем и сохраняем расположение спрайтов
     for (int i = 0; i<4;i++)
     {
          sprites[i].top = 0;
          sprites[i].left = i*32;
          sprites[i].bottom = 31;
          sprites[i].right = i*32 + 31;
     }
     //стартовое положение, направление и скорость
     position.top = 200;
     position.bottom = 231;
     position.left = 100;
     position.right = 131;
     direction = 3;
     speed = 4;
}

BOOL Ship::isCollision(int moveX, int moveY)
//проверка столкновения с сушей на основе
//предполагаемого перемещения корабля
{
     //определяем имя файла текущей карты
     char* temp;
     if (currentMapX == 0 && currentMapY == 0)
          temp = "map1.bmp";
     if (currentMapX == 1 && currentMapY == 0)
```

```
            temp = "map2.bmp";
        if (currentMapX == 2 && currentMapY == 0)
            temp = "map3.bmp";
        if (currentMapX == 0 && currentMapY == 1)
            temp = "map4.bmp";
        if (currentMapX == 1 && currentMapY == 1)
            temp = "map5.bmp";
        if (currentMapX == 2 && currentMapY == 1)
            temp = "map6.bmp";
        if (currentMapX == 0 && currentMapY == 2)
            temp = "map7.bmp";
        if (currentMapX == 1 && currentMapY == 2)
            temp = "map8.bmp";
        if (currentMapX == 2 && currentMapY == 2)
            temp = "map9.bmp";
        //используем landCollision для проверки
        return landCollision(position.left%800 + 16 +
            moveX, position.top%600 + 16 + moveY, temp);
}
void Ship::Move(int dir)
//перемещает корабль
{
    //определяем новое направление
    direction = dir;
    int moveX; //дистанция перемещения?
    int moveY;
    //вычисляем новое положение на основе направления
    switch (direction)
    {
        case 0:
            moveX = speed;
            moveY = 0;
            break;
        case 1:
            moveX = 0;
            moveY = speed;
            break;
        case 2:
            moveX = -speed;
            moveY = 0;
            break;
        case 3:
            moveX = 0;
            moveY = -speed;
            break;
    }

    //если игрок пытается уплыть за пределы сетки карты, не двигаться
    if ((position.bottom + moveY > 1799) ||
            (position.top + moveY < 0))
        moveY = 0;
```

```
     if ((position.right + moveX > 2399) ||
               (position.left + moveX < 0))
          moveX = 0;
     //если игрок уперся в сушу, не двигаться
     if (isCollision(moveX, moveY))
     {
          moveX = 0;
          moveY = 0;
     }

     //перемещаем корабль
     position.top += moveY;
     position.bottom += moveY;
     position.left += moveX;
     position.right += moveX;

     //изменилась ли карта?
     int tempX = position.left/800;
     int tempY = position.top /600;
     //изменяем текущую карту при необходимости
     if (tempX != currentMapX || tempY != currentMapY)
     {
          currentMapX = tempX;
          currentMapY = tempY;
     }
}

void Ship::Draw()
//draws the game screen
{
     //location - позиция, где следует разместить корабль на экране
     RECT location = makeRECT(position.left%800,
          position.top%600,
          position.right%800,
          position.bottom%600);
     //подготовка информации для отображения
     DDBLTFX ddbltfx;
     ZeroMemory(&ddbltfx,sizeof(ddbltfx));
     ddbltfx.dwSize = sizeof(ddbltfx);
     //копируем карту-фон на экран
     DDCopyBitmap(g_pddsback,
          entireMap[currentMapX][currentMapY],0,0,800,600);
     //поверх рисуем корабль, сделав
     //прозрачным фон спрайтов
     g_pddsback->Blt(&location,c_pddsone,&sprites[direction],
          DDBLT_WAIT | DDBLT_KEYSRC,&ddbltfx);
     //отображаем результат
     g_pddsprimary->Flip(NULL,0);
}

void Ship::moveBack()
//движение в противоположном направлении - для выхода из зоны города
{
```

```
    int temp;
    if (direction == 0)
        temp = 2;
    if (direction == 1)
        temp = 3;
    if (direction == 2)
        temp = 0;
    if (direction == 3)
        temp = 1;
    this->Move(temp);
}
```

Конструктор этого класса загружает спрайты в плоскость, вычисляет их координаты на плоскости, а затем инициализирует положение и направление движения для корабля.

Метод `isCollision` определяет текущую карту и координаты корабля (после следующего перемещения), а затем передает эти данные функции `landCollision` из модуля Movement.h.

Метод `Move` вычисляет дистанцию следующего перемещения корабля, проверяет возможные столкновения (если они могут произойти, значение дистанции обнуляется), перемещает корабль и определяет, произошел ли переход на другую карту.

Метод `Draw` вычисляет положение корабля на экране (хранимое положение определяет координаты для сетки карт в целом), копирует во вторичный буфер сначала карту, а затем спрайт, после чего отображает плоскость на экране.

Метод `moveBack` изменяет направление движения корабля, а затем перемещает его. Эта функция используется для выхода из района города. В противном случае снова и снова отображается экран города.

Основная программа

Следующий файл – Pirates.cpp. Это главный файл, включающий все остальные (подобные модули тестирования описаны в главе 5). В этом файле содержатся функции `WinMain` и `WndProc`.

```
//14.5 - Pirates.cpp - Марк Ли - Premier Press

#include "globals.h"
int WINAPI WinMain(/*Точка входа Win32 */
                        HINSTANCE hInst,
                        HINSTANCE hPreInst,
                        LPSTR lpszCmdLine,
                        int nCmdShow )
{
    //вызываем InitApp, инициализация
    InitApp(hInst, nCmdShow);
    //вызываем DoEventLoop, обработка цикла событий
```

```
     return DoEventLoop();
}
/*процедура обратного вызова */
LRESULT CALLBACK WndProc( HWND hWnd, UINT messg, WPARAM wParam, LPARAM
lParam)
//эта функция обрабатывает ввод пользователя
//и таймеры, используемые в игре
{
     static bool inCity = false; //Определяет, в городе ли игрок
     switch(messg) //Определяет, какое отправляется сообщение
     {
          case WM_CREATE: //создается окно
               //выполнить инициализацию DirectX
               InitializeDirectX(hWnd);
               CreateTowns();//создать все объекты Town

               //Заставка отображается в течение 5 секунд (5000 миллисекунд)
               //Затем начинается собственно игра
               //Установка таймера для заставки
               SetTimer(hWnd,OpeningTimer,5000,NULL);
               //Отображаем заставку
               DisplayScreen(OpeningScreen);

               break;//конец варианта WM_CREATE

          //это событие позволяет избавиться от курсора мыши
          case WM_SETCURSOR:
               SetCursor(NULL);//установка в NULL
               break;//конец варианта WM_SETCURSOR

          case WM_KEYDOWN: //если нажата клавиша
               //если игрок находится в городе
               //доступна только клавиша Enter
               if (inCity)
               {
                    //если нажата клавиша - завершаем показ экрана города
                    if(wParam == VK_RETURN)
                         inCity = false;
                    break;
               }
               //обрабатываем прочие варианты (игра вне города)
               switch(wParam)//какая клавиша нажата
               {
                    case VK_UP://стрелка вверх
                         //сообщаем всем, что следует
                         //двигаться вверх
                         moveUp = true;
                         break;
                    case VK_DOWN://стрелка вниз
                         moveDown = true;
                         break;
                    case VK_LEFT://стрелка влево
                         moveLeft = true;
```

```
                    break;
               case VK_RIGHT://стрелка вправо
                    moveRight = true;
                    break;
               case VK_SPACE://пробел
                    //залп бортовых орудий
                    //обратите внимание, стрельба
                    //еще не реализована
                    fire = true;
                    break;

               case VK_ESCAPE://клавиша Esc
                    //уничтожение главного таймера
                    KillTimer(hWnd,MainTimer);
                    PostQuitMessage(0);//конец программы
                    break;
          }
          break;//конец варианта WM_KEYDOWN

     case WM_KEYUP://клавиша отпущена
          switch(wParam)//определяем, какая
          {
               //если клавиша отпущена, прекращаем движение
               case VK_UP:
                    moveUp = false;
                    break;
               case VK_DOWN:
                    moveDown = false;
                    break;
               case VK_LEFT:
                    moveLeft = false;
                    break;
               case VK_RIGHT:
                    moveRight = false;
                    break;
               case VK_SPACE:
                    fire = false;
                    break;
          }
          break;//конец варианта WM_KEYUP

     case WM_TIMER://таймер сработал
          //если игрок в городе, не обращать внимания на таймер
          if (inCity)
               break;//прекратить обработку сообщения WM_TIMER

          switch(wParam)//выяснить, о каком таймере идет речь
          {
               //таймер заставки
               case OpeningTimer:
                    //запустить главный таймер
                    SetTimer(hWnd,MainTimer,15,NULL);
                    //уничтожить таймер заставки
```

```
                    KillTimer(hWnd,OpeningTimer);
                    //отобразить экран в первый раз
                    Draw();
                    break;//конец варианта OpeningTimer

               //таймер основной игры
               case MainTimer:
                    DoMove();//переместить корабль исходя из
                             //ввода пользователя
                    Redraw();//обновить изображение
                    //если игрок вошел в город
                    if(isInCity(&ship->GetPosition()))
                    {
                         //сообщить всем
                         inCity = true;
                         //отобразить экран города
                         DisplayScreen(CityScreen);
                         //поместить игрока за черту города
                         ship->moveBack();
                    }
                    break;//конец варианта MainTimer
          }
          break;//конец варианта WM_TIMER

     default: //остальные сообщения оставляем системе Windows
          return( DefWindowProc( hWnd,
               messg, wParam, lParam ) );
     }

     return(0);//возвращаемое значение по умолчанию
}
```

Обратите внимание, метод WM_SETCURSOR делает курсор мыши невидимым. Кроме того, в работе мы используем два таймера – один для экрана заставки, который отображается пять секунд, и второй – для главного цикла игры. Экран непрерывно обновляется со скоростью примерно 20 раз в секунду.

Перемещение корабля

Следующий файл – Movement.h. В нем объявлены алгоритмы движения корабля, а также функции обнаружения столкновений. Реализация всех функций содержится в файле Movement.cpp.

```
//14.6 - Movement.h - Марк Ли - Premier Press
#ifndef MOVEMENT_H
#define MOVEMENT_H

#include <fstream>
#include "globals.h"

using namespace std;
```

```
//создает  RECT из четырех целых
RECT makeRECT(int x1, int y1, int x2, int y2);
//возвращает true, если точка (x,y) находится внутри RECT
BOOL PointInRECT(int x, int y, RECT* rc);
//возвращает true, если два прямоугольника пересекаются
BOOL RECTinRECT(RECT* one, RECT* two);
//возвращает true, если прямоугольник касается города
BOOL isInCity(RECT* pos);
//возвращает true, если точка (x,y) принадлежит суше
bool landCollision(int x, int y, string filename);
//Перемещает корабль исходя из ввода пользователя
void DoMove();
#endif
//14.7 - Movement.cpp - Марк Ли - Premier Press
#include "Movement.h"

BOOL PointInRECT(int x, int y, RECT* rc)
//возвращает true, если точка находится внутри прямоугольника RECT
{
     //оператор if с длинным условием для проверки этого обстоятельства
     if (x < rc->right && x > rc->left &&
          y < rc->bottom && y > rc->top)
          return true;
     //возвращает false, если не внутри
     return false;
}

BOOL RECTinRECT(RECT* one, RECT* two)
//возвращает true, если у двух прямоугольников есть общая площадь
{
     //Принадлежит ли хотя бы один из четырех углов
     //одного прямоугольника второму?
     if (PointInRECT(one->left, one->top, two))
          return true;
     if (PointInRECT(one->left, one->bottom, two))
          return true;
     if (PointInRECT(one->right, one->top, two))
          return true;
     if (PointInRECT(one->right, one->bottom, two))
          return true;
     //возвращает false, если не пересекаются
     return false;
}

BOOL isInCity(RECT* pos)
//возвращает true, если pos касается какого-либа города
{
     for (int i = 0; i <13; i++)
          //перебираем города
          //и выполняем проверку RECTinRECT
          if (RECTinRECT(pos,&towns[i].position))
               return true;
```

```
     return false;
}

bool landCollision(int x, int y, string filename)
//возвращает true, если точка (x,y) принадлежит суше
//filename - имя растрового файла карты, из которого производится чтение
данных
{
     y = 599 - y;//переключаемся в компьютерную систему координат
     //если (x,y) находится за пределами экрана, столкновения нет
     if(x < 0 || y > 599)
          return false;
     if(y < 0 || x > 799)
          return false;
     //файловый ввод-вывод
     ifstream file(filename.c_str(), ios::in|ios::binary);
     //читаем данные с корректной позиции
     file.seekg(3 * y * 800 + 3 * x + 54);
     //три байта на каждую точку
     //3 - красный, 2 - зеленый, 1 - голубой
     char array[3];
     file.read(array, 3);
     //закончить файловый ввод-вывод
     file.close();
     //если зеленого меньше, чем голубого, мы находимся в море
     if((int)array[1] < (int)array[0])
          return false;
     //в противном случае - столкновение
     return true;

}

void DoMove()
//перемещает корабль в зависимости от того, какие нажаты клавиши
{
     //Определяем направление движения
     int dir = ship->GetDirection();
     if (moveUp == true)
          dir = 3;
     else if (moveDown == true)
          dir = 1;
     else if (moveLeft == true)
          dir = 2;
     else if (moveRight == true)
          dir = 0;
     else return;
     //перемещаем корабль
     ship->Move(dir);
}

RECT makeRECT(int x1, int y1, int x2, int y2)
//создает прямоугольник по четырем точкам
{
```

```
        //теория такова: один раз напишешь код, больше писать не придется
        RECT temp;
        temp.left = x1;
        temp.top = y1;
        temp.right = x2;
        temp.bottom = y2;
        return temp;
    }
```

PointInRECT возвращает true, если точка (x, y) расположена внутри прямоугольника RECT.

RECTinRECT четыре раза вызвает PointInRECT, чтобы выяснить, принадлежит ли какой-либо из углов первого прямоугольника площади второго.

isInCity тринадцать раз вызывает RECTinRECT, чтобы выяснить, граничит ли полученный прямоугольник (положение корабля) с чертой города.

landCollision открывает указанный файл и проверяет, является ли точка (x, y) более зеленой, чем голубой (то есть принадлежит ли суше).

doMove преобразует ввод пользователя в направление движения, а затем сообщает объекту Ship, что необходимо совершить перемещение.

makeRECT — это вспомогательная функция, которая создает структуру RECT на основе четырех целых чисел.

Отображение графики

Следующий файл — Drawing.h, который содержит все функции, связанные с выводом графики. Код DirectX практически полностью содержится в этом файле (библиотека «DirectX» описана в главе 13). Реализация этих функций хранится в файле Drawing.cpp.

```
//14.8 - Drawing.h - Марк Ли - Premier Press
#ifndef DRAWING_H
#define DRAWING_H

#include "ddraw.h"
#include "globals.h"

LPDIRECTDRAW7 g_pdd; //объект DirectDraw
LPDIRECTDRAWSURFACE7 g_pddsprimary; //основная плоскость
LPDIRECTDRAWSURFACE7 g_pddsback; //вторичный буфер
LPDIRECTDRAWSURFACE7 g_pddsone; //временная плоскость
DDSURFACEDESC2 ddsd; //хранит описания плоскостей
DDSCAPS2 ddsc; //хранит возможности плоскости
//переменная для временного хранения результата выполнения функции
HRESULT hRet;

//инициализирует все плоскости directX
void InitializeDirectX(HWND hWnd);
//начальное отображение экрана игры
void Draw();
```

```
//загружает карты в структуры HBITMAP
void LoadMaps();
//обновление экрана игры
void Redraw();
//выводит растровое изображение на экран
void DisplayScreen(char* bmp);
#endif
//14.9 - Drawing.cpp - Марк Ли - Premier Press
#include "drawing.h"

void DisplayScreen(char* bmp)
//выводит растровое изображение на экран
{
     //подготовка информации для отображения
     DDBLTFX ddbltfx;
     ZeroMemory(&ddbltfx,sizeof(ddbltfx));
     ddbltfx.dwSize = sizeof(ddbltfx);
     //создание плоскости на основе растрового изображения
     g_pddsone = DDLoadBitmap(g_pdd,bmp,800,600);
     //копировать плоскость во вторичный буфер
     g_pddsback->Blt(NULL,g_pddsone, NULL,NULL,&ddbltfx);
     //вывести подготовленный экран
     g_pddsprimary->Flip(NULL,0);
}

void Redraw()
//функция обновления изображения на экране
{
     ship->Draw();
}

void LoadMaps()
//создает структуры HBITMAP для всех файлов карт
{
     //Функция Win32, LoadImage
     entireMap[0][0] = (HBITMAP)LoadImage(NULL,
          "map1.bmp",IMAGE_BITMAP, 800,600,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION);
     entireMap[1][0] = (HBITMAP)LoadImage(NULL,
          "map2.bmp",IMAGE_BITMAP, 800,600,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION);
     entireMap[2][0] = (HBITMAP)LoadImage(NULL,
          "map3.bmp",IMAGE_BITMAP, 800,600,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION);
     entireMap[0][1] = (HBITMAP)LoadImage(NULL,
          "map4.bmp",IMAGE_BITMAP, 800,600,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION);
     entireMap[1][1] = (HBITMAP)LoadImage(NULL,
          "map5.bmp",IMAGE_BITMAP, 800,600,
          LR_LOADFROMFILE | LR_CREATEDIBSECTION);
     entireMap[2][1] = (HBITMAP)LoadImage(NULL,
          "map6.bmp",IMAGE_BITMAP, 800,600,
```

```
            LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        entireMap[0][2] = (HBITMAP)LoadImage(NULL,
            "map7.bmp",IMAGE_BITMAP, 800,600,
            LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        entireMap[1][2] = (HBITMAP)LoadImage(NULL,
            "map8.bmp",IMAGE_BITMAP, 800,600,
            LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        entireMap[2][2] = (HBITMAP)LoadImage(NULL,
            "map9.bmp",IMAGE_BITMAP, 800,600,
            LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        //первая карта - в левом верхнем углу
        currentMapX = 0;
        currentMapY = 0;
}

void InitializeDirectX(HWND hWnd)
//выполняет инициализацию DirectX
{
        //создание объекта DirectDraw
        hRet = DirectDrawCreateEx(NULL, (void**)&g_pdd,
            IID_IDirectDraw7, NULL);
        if(hRet != DD_OK)
            MessageBox(hWnd, "Невозможно выполнить DirectDrawCreateEx",
                "Ошибка", NULL);
        //Установка уровня сотрудничества
        hRet = g_pdd->SetCooperativeLevel(hWnd,
            DDSCL_FULLSCREEN | DDSCL_EXCLUSIVE);
        if(hRet != DD_OK)
            MessageBox(hWnd, "Невозможно выполнить SetCooperativeLevel",
                "Ошибка", NULL);;
        //Режим отображения: 800x600, 16 битов на точку
        hRet = g_pdd->SetDisplayMode(800, 600, 16,0,0);
        if(hRet != DD_OK)
            MessageBox(hWnd, "Невозможно выполнить SetDisplayMode",
                "Ошибка", NULL);;

        //подготовка информации основной плоскости
        ZeroMemory(&ddsd,sizeof(ddsd));
        ddsd.dwSize = sizeof(ddsd);
        ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
        ddsd.dwBackBufferCount = 1;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
            DDSCAPS_FLIP | DDSCAPS_COMPLEX;

        //создание плоскости
        hRet = g_pdd->CreateSurface(&ddsd,
            &g_pddsprimary, NULL);
        if (hRet != DD_OK)
            MessageBox(hWnd, "Невозможно выполнить CreateSurface",
                "Ошибка", NULL);

        //подготовка параметров вторичного буфера
        ZeroMemory(&ddsc,sizeof(ddsc));
```

```
        ddsc.dwCaps = DDSCAPS_BACKBUFFER;

        //получаем указатель на вторичный буфер
        hRet = g_pddsprimary->GetAttachedSurface(&ddsc,
             &g_pddsback);

    }

    void Draw()
    //функция начального отображения экрана игры
    {
        //создаем корабль
        ship = new Ship();
        //отображаем корабль поверх карты
        ship->Draw();
    }
```

Функция `DisplayScreen` создает плоскость, используя файл с указанным именем. Затем плоскость копируется во вторичный буфер и переносится на экран.

Функция `Redraw` вызывается при первом обновлении экрана игры. Она просто предписывает кораблю вывести свое изображение.

`LoadMaps` девять раз вызывает `LoadImage` для загрузки растровых файлов карт в структуры `HBITMAP`. Текущей картой становится секция в левом верхнем углу.

`InitializeDirectX` выполняет инициализацию DirectX. Функция создает первичную плоскость и вторичный буфер.

Функция `Draw` используется для начального отображения экрана игры. Она создает объект корабля и предписывает ему вывести свое изображение.

Посещение городов

Следующий файл – Towns.h; он содержит все функции для работы с городами. Кроме того, здесь же объявлена структура `Town`, которая хранит информацию для одного города (о структурах мы рассказывали в главе 9 «Шаблоны»). Реализация функций хранится в файле Towns.cpp.

```
//14.10 - Towns.h - Марк Ли - Premier Press
#ifndef TOWNS_H
#define TOWNS_H

#include <vector>
#include "Movement.h"

using namespace std;

//хранит информацию о городе
struct Town
{
```

```
    LPCTSTR name;
    RECT position;
};

vector<Town> towns;//все города (13 штук)

//создает все объекты Town
void CreateTowns();
#endif
//14.11 - Towns.cpp - Марк Ли - Premier Press
#include "Towns.h"

void CreateTowns()
//инициализация городов
{
    Town* temp = new Town;//создаем новый объект Town
    towns.push_back(*temp);//добавляем его к вектору
    towns[0].name = "New Orleans";//присваиваем имя
    //и задаем расположение
    towns[0].position = makeRECT(90,100,190,150);
    temp = new Town;
    towns.push_back(*temp);
    towns[1].name = "Tampa";
    towns[1].position = makeRECT(475,310,530,365);
    temp = new Town;
    towns.push_back(*temp);
    towns[2].name = "New Providence";
    towns[2].position = makeRECT(840,460,945,500);
    temp = new Town;
    towns.push_back(*temp);
    towns[3].name = "New Dutch Land";
    towns[3].position = makeRECT(1950,150,2020,200);
    temp = new Town;
    towns.push_back(*temp);
    towns[4].name = "Belize";
    towns[4].position = makeRECT(20,980,90,1010);
    temp = new Town;
    towns.push_back(*temp);
    towns[5].name = "Port-de-Paix";
    towns[5].position = makeRECT(1220,945,1320,990);
    temp = new Town;
    towns.push_back(*temp);
    towns[6].name = "Port-au-Prince";
    towns[6].position = makeRECT(1235,1030,1320,1070);
    temp = new Town;
    towns.push_back(*temp);
    towns[7].name = "Santiago";
    towns[7].position = makeRECT(1360,940,1435,1000);
    temp = new Town;
    towns.push_back(*temp);
    towns[8].name = "Santo Domingo";
    towns[8].position = makeRECT(1485,1030,1580,1090);
```

```
        temp = new Town;
        towns.push_back(*temp);
        towns[9].name = "Prinzapolca";
        towns[9].position = makeRECT(335,1530,460,1595);
        temp = new Town;
        towns.push_back(*temp);
        towns[10].name = "Santa Marta";
        towns[10].position = makeRECT(1285,1490,1370,1525);
        temp = new Town;
        towns.push_back(*temp);
        towns[11].name = "Carthagena";
        towns[11].position = makeRECT(1060,1660,1130,1705);
        temp = new Town;
        towns.push_back(*temp);
        towns[12].name = "Caracas";
        towns[12].position = makeRECT(1960,1510,2010,1545);
        //всего 13 городов
    }
```

В этом разделе присутствует лишь одна, но довольно объемная функция, `CreateTowns`, которая заполняет вектор городов объектами `Town`. Каждый город хранит собственное расположение.

Поздравляем, читатель!

Ты сделал это. Только лучшие из лучших способны пройти этот путь. Теперь ты по праву можешь называть себя программистом.

Тем из читателей, кого еще не оставила тяга к знаниям, мы советуем обратиться к другим источникам информации по таким темам, как программирование для Windows и DirectX. Замечательной книгой по программированию для Windows станет «Программирование в Windows» (Programming Windows) Чарльза Петзольда (Charles Petzold), Microsoft Press. Лучшим источником информации по DirectX являются файлы справки, поставляемые в составе DirectX SDK. Совершенствуя владение C++, загляните в книгу «Язык программирования C++»[1], написанную создателем языка Бьерном Страуструпом (Addison-Wesley). И конечно же, вы можете начать изучение других языков программирования, таких как Java и Visual Basic. Теперь вам принадлежит весь мир!

Конкурс

В игре «Пиратское приключение» вы едва коснулись того, что может быть сделано. Например, города не имеют никакой функциональнос-

[1] Б. Страуструп. «Язык программирования C++». – СПб: Невский диалект, 2001.

ти. Усовершествуйте игру и пришлите ее нам до 31 декабря 2001 года. Автор лучшей игры будет награжден полной учебной версией компилятора CodeWarrior 7!

Вот несколько пунктов, которые помогут вам начать работу:

- Добавьте облака
- Добавьте другие корабли, причем различных типов
- Наделите города функциональностью
- Создайте различные нации в Карибском бассейне
- Создайте вступительный ролик

Копии завершенных игр присылайте по адресу *marklee@power-surfr.com* либо на компакт-дисках почтой:

Dirk Henkemans
10714-126 St.
T5M 0N8
Edmonton Alberta Canada

Игра-победитель будет опубликована по адресу *http://plaza.power-surfr.com/FireStorm*.

A

Ответы к заданиям

В данном приложении содержатся ответы к заданиям, приведенным в конце каждой главы. Как мы неоднократно говорили, рекомендуется пытаться выполнить задания самостоятельно, прежде чем заглядывать в ответы. (Заметим, что в некоторых случаях приводимые здесь ответы являются лишь предложениями по решению задач.)

Глава 1

1. Напишите программу, которая выводит изображение домика, похожее на то, что представлено на рис. 1.7.

***Рис. 1.7.** Домик в стиле ASCII*

Ответ:

```
//выводит домик

#include <iostream>
using namespace std;

int main( void )
{
     cout << "    /\\   " << endl;
```

```
    cout << "  /   \\  " << endl;
    cout << " /      \\ " << endl;
    cout << " |      | " << endl;
    cout << " |[]  []| " << endl;
    cout << " |      | " << endl;
    cout << " ---- " << endl;

    return 0;
}
```

2. **Что выводит следующая программа?**

```
#include <iostream>
using namespace std;      //introduces namespace std
int x = 25;
string str2 = "Это проверка";

int main( void )
{
    cout<<"Проверка"<<1<<2<<"3";
cout<<25 %7<<endl<<str2.c_str();
return 0;
}
```

Ответ:

```
Проверка1234
Это проверка
```

Помните, что оператор `cout` **самостоятельно не выполняет перенос текста на следующую строку.**

3. **Напишите программу, которая запрашивает у пользователя его имя, приветствует его, затем запрашивает два числа и отображает их сумму.**

Ответ:

```
#include <iostream>
using namespace std;      //introduces namespace std
string name = "";
int integer1;
int integer2;

int main( void )
{
    //запрашиваем имя пользователя
    cout<< "Введите свое имя.";
    //пользователь вводит имя, которое
    //сохраняется в переменной name
    cin>> name;
    //программа приветствует пользователя и пропускает строку
    cout<< "Привет, " << name << endl;
    cout<< "Введите первое слагаемое: ";
    cin>> number1;  //пользователь вводит первое число
```

```
     cout<< "Введите второе слагаемое: ";
     cin>> number2;  //пользователь вводит второе число

\\выводим сумму двух чисел
     cout<<"сумма чисел: " << (number1 + number2);
     return 0;
}
```

4. Что произойдет, если сохранить 10.3 в качестве целого числа? А если 0.6? Можно ли сохранить число -101.8 в качестве целого?

   Ответ:

   C++ усечет (отбросит) дробную часть, то есть будут сохранены числа 10, 0 и -101.

5. Напишите код, умножающий исходное число на 2, если оно принадлежит интервалу от 1 до 100 (включительно) и делится нацело на 3; в противном случае умножает на три, если число принадлежит интервалу от 1 до 100, но не делится нацело на три; и наконец умножает число на остаток от его деления на 100, если число не принадлежит интервалу от 1 до 100. (Совет: используйте вложенные операторы `if`.)

   Ответ:

```
#include <iostream>
using namespace std;

int main( void )

{

     int number;

     cout << "Введите число" ;

     cin>> number;

     if (number >= 1 && number <= 100 && (number % 3 == 0))
     {
          number = number * 2;
     }
     else if (number >= 1 && number <= 100 && !(number % 3 == 0))
     {
          number = number * 3;
     }
     else if (!(number >= 1 && number <= 100))
     {
          number = number * number % 3;
     }

     return 0;
}
```

Глава 2

1. Выберите подходящий тип переменной для хранения следующей информации:

   Число книг на книжной полке

   Стоимость отдельной книги

   Число людей в мире

   Слово *Привет*

   Ответ:

   - Для хранения числа книг на полке лучше всего подойдет, вероятно, беззнаковое целое или короткое целое, поскольку книг не может быть отрицательное или дробное количество.
   - Для стоимости книги подойдет тип данных с плавающей точкой, поскольку наличие плавающей запятой позволяет хранить одновременно рубли и копейки.
   - Число людей в мире лучше всего хранить в переменной с плавающей точкой типа float, поскольку этот тип позволяет записывать невероятно большие числа в удобном экспоненциальном формате.
   - Слово *Привет* можно хранить в качестве строки либо в качестве массива символов.

2. Придумайте содержательные имена для переменных из первого задания.

   Ответ:

   Идентификаторы переменных могут быть различными, но они должны отражать назначение переменных. Это делает код понятным не только автору, но и другим программистам. Простые идентификаторы переменных из первого задания: `numOfBooks`, `bookCost`, `numOfPeople` и `helloString` соответственно.

3. Приведите две причины использования констант вместо литералов.

   Ответ:

   Применение константы позволяет изменить лишь одно значение – в точке ее объявления, тогда как в случае литералов приходится вносить изменения по всей программе. Кроме того, применение констант делает код более читаемым.

4. Напишите программу, которая вычисляет и отображает размеры всех основных типов.

   Ответ:

```
//выводит размеры основных типов переменных
```

```
#include <iostream>
using namespace std;

int main( void )
{
     cout << "Размеры основных типов"
          " переменных.";
     cout << "\nint - " << sizeof(int);
     cout << "\nshort - " << sizeof(short);
     cout << "\nfloat - " << sizeof(float);
     cout << "\ndouble - " << sizeof(double);
     cout << "\nbool - " << sizeof(bool);
     cout << "\nchar - " << sizeof(char);
     cout << "\n Наш эксперимент завершен :)";

     return 0;
}
```

5. Выясните, что происходит, когда символьный тип объявляется с модификатором `unsigned`. Соответствуют ли результаты ожиданиям? Сформулируйте причину соответствия или несоответствия.

   Ответ:

   Наличие или отсутствие знака у символьного типа не оказывает влияние на результаты для символов ASCII. Однако для значений больше 127 попытки использовать отрицательные значения могут иметь непредсказуемый результат, поскольку эта часть таблицы ASCII не стандартизирована.

Глава 3

1. Создайте условный оператор (`if`), который присваивал бы x значение x/y, если y не равно 0.

   Ответ:

```
if(y != 0)
{
      x = x/y;
}
```

2. Создайте цикл `while`, вычисляющий сумму положительных целых чисел от 1 до некоторого числа *n* (проверьте результат по формуле `n*(n+1)/2`).

   Ответ:

```
int i = 0;
int sum = 0;
while(i < n)

{
```

```
    i++;
    sum += i;
}
```

3. Создайте условный оператор, который присваивал бы x*y для четного x, в противном случае для нечетного x и y, не равного 0, присваивал бы x/y; наконец, если ни одно из предыдущих условий не вычисляется в true, выводил бы на экран сообщение, что значение y равно 0.

   Ответ:

```
if(x % 2 == 0)
{
     x = x * y;
}

else if((x % 2) == 1 && y != 0)
{
     x = x / y;
}

else
{
     cout << "y = 0" << endl;
}
```

Глава 4

1. Создайте функцию multiply, которая перемножает два числа и возвращает результат.

   Ответ:

```
long multiply(int x, int y)
{
    return  (x * y) ;
}
```

2. Измените функцию multiply, чтобы она запоминала, сколько раз ее вызвали.

   Ответ:

```
int multiply(int x, int y)
{
    static int staticMember;
    staticMember++;
    return  (x * y) ;
}
```

3. В чем разница между глобальной переменной и статической переменной? В каких ситуациях предпочтительно использовать тот или иной вариант и почему?

Ответ:

Как глобальная, так и статическая переменная существуют на протяжении всего цикла жизни программы. Разница заключается в области видимости переменных. Глобальная переменная доступна в любой точке программы, а статическая только в том методе, где объявлена. Таким образом, если доступ к переменной требуется только в пределах функции, более подходящим вариантом станет статическая переменная. Глобальные переменные используются в случаях, когда необходим доступ из нескольких функций.

4. Попробуйте переписать игру «Приключение в пещере», не используя функций. Упражнение покажет, насколько полезны функции.

Ответ:

Это возможно; однако если поместить весь код программы в одну функцию, получится так называемый код-«спагетти», который снижает прозрачность программы. Поверьте, функции действительно полезны!

5. Если вы честно выполнили все задания, купите себе прохладительный напиток.

Ответ:

Пять шагов к прохладительному напитку:

1. Заработайте $1.50.
2. Доберитесь до ближайшего круглосуточного магазина.
3. Помните, программисты не ходят пешком, поезжайте на машине.
4. Заплатите за напиток.
5. Прохлаждайтесь!

Глава 5

1. Создайте класс для представления персонажа ролевой игры. Класс должен хранить имя персонажа, его классовую принадлежность и расу.

Ответ:

```
class player
{
     string name;
     string charClass;
     string race;

     player(string lname, string lclass, string lrace);
};
//конструктор со списком инициализации
```

```
player::player(string lname, string lclass, string lrace)
     : name(lname), charClass(lclass), race(lrace);
{
}
```

2. Изложите три базовых принципа ООП.

   Ответ:

   Три базовых принципа:

   - **Абстракция данных**. Скрывает и защищает данные объекта. Доступ к данным осуществляется посредством методов.
   - **Инкапсуляция**. Каждая сущность выделяется в самостоятельный объект.
   - **Полиморфизм**. Каждый объект способен решать свои задачи в любой программе. Это основа переносимости и повторного использования кода.

3. В чем различие между классом и объектом?

   Ответ:

   Класс подобен шаблону в том смысле, что является указанием компьютеру, как создавать объект. Объявление класса эквивалентно обращению к компилятору: «Вот новый тип данных, и вот что он может делать». Класс похож на функцию, которая не была вызвана. Объект является экземпляром класса; по сути дела, класс – это основа для создания объекта. Объект занимает место в памяти, а класс – нет.

4. Имея возможность выбрать между общим (public), частным (private), глобальным и локальным объявлением без потери функциональности, что вы выберете?

   Ответ:

   Следует выбрать локальное объявление, поскольку оно имеет минимальную область видимости. Это принцип абстракции данных ООП в действии.

5. Какие атрибуты конструкторов и деструкторов отсутствуют в других функциях?

   Ответ:

   - Конструктор и деструктор не могут возвращать значения.
   - Деструктор не может иметь параметров.
   - Деструктор – единственная функция, имя которой начинается с тильды.
   - Имена конструктора и деструктора совпадают с именем объекта.

Глава 6

1. Какой размер имеет строка "Здравствуй, мир"? Какова длина массива s?

```
char s[] = "Здравствуй, мир";
```

Ответ:

Размер строки "Здравствуй, мир" – 16 байт. Каждая из букв занимает один байт, запятая и пробел – по одному байту, завершающий строку пустой символ также занимает один байт; итого – 16 байт. Но не стоит забывать, что индексы массива лежат в диапазоне от 0 до 15.

2. Приведите пять причин использования указателей.

Ответ:

Основания для применения указателей:

- Указатели позволяют обращаться к произвольным элементам массивов.
- Указатели позволяют передавать ссылку на данные вместо собственно данных.
- Указатели являются основой работы с динамической памятью.
- Указатели являются основой для доступа к бинарным файлам.
- Указатели являются основой итераторов.
- Указатели – это интересно!

3. Какие проблемы существуют в игре «Крестики-нолики», приведенной в конце главы? Как можно улучшить игру?

Ответ:

Интерфейс игры довольно неудобный. Чтобы ввести номер строки и колонки (0–2), необходимо знать, что нумерация начинается с нуля. Ситуацию можно исправить, пронумеровав строки и колонки.

Кроме того, можно создать компьютерного противника. Искусственный разум компьютерного оппонента должен проверять наличие возможности получить три X или O в ряд, а затем пытаться привести игру к такой ситуации.

4. Приведите три причины использования динамической памяти.

Ответ:

- Возможность создавать массивы произвольного размера.
- Динамической памяти больше, чем системной. Системная память выделяется программе операционной системой, а динамическую память программа «заимствует» из крупных областей свободной памяти.

- Программа может самостоятельно принимать решения по выделению и освобождению памяти.

Глава 7

1. Объясните, зачем можно использовать пространства имен.

   Ответ:

   Пространства имен позволяют разделять крупные области видимости на более мелкие. Это позволяет программистам избегать конфликтов имен.

2. Какие преимущества имеет безымянное пространство имен перед обычным?

   Ответ:

   Безымянное пространство имен автоматически получает уникальный идентификатор, тогда как именованное пространство имен может конфликтовать с другим, уже существующим.

3. Назовите два способа организации прямого доступа к пространствам имен. Чем они различаются?

   Ответ:

   Первый способ – использовать объявление `using`. Объявление `using` показывает, что автор намеревается работать с указанным компонентом области видимости, определенной данным пространством имен. В результате при доступе к указанному компоненту отпадает необходимость в явном уточнении области видимости.

   Второй способ – использовать директиву `using`. Директива `using` показывает, что автор намеревается работать со всеми компонентами указанного пространства имен. Директива `using` действует подобно объявлению `using` с той разницей, что применяется ко всем компонентам указанного пространства имен. Директива `using` избавляет от необходимости уточнять имена при обращении к компонентам пространства имен до конца программы.

4. Для приведенного ниже кода, каким образом можно вызвать функцию `breathFire()` из:

   a. глобального пространства имен

   b. пространства имен `dragon`

   c. из другого пространства имен

```
namespace dragon
{
    void breathFire() {cout<< "Дракон дышит огнем \n"; }
}
using dragon::breathFire();
```

Ответ:

a. `breathFire();`

b. `breathFire();`

c. `breathFire();`

Глава 8

1. Создайте иерархию классов оружия, в которой по меньшей мере четыре класса наследуются от базового. Какие поля данных должен содержать каждый из классов? Какие методы?

   Ответ:

   Иерархия может включать класс оружия в качестве базового и ряд конкретных порожденных классов. Примеры порожденных классов: `sword` (меч), `dagger` (кинжал), `bow` (лук). Каждый такой класс должен иметь ряд специальных методов, отражающих возможности оружия.

2. Приведите пример ситуации, когда может оказаться полезным множественное наследование. Легче ли решить задачу при помощи одиночного наследования?

   Ответ:

   Для любого класса, которому необходимы свойства двух базовых классов, следует использовать множественное наследование. Например, класс танка может происходить от класса оружия и класса транспортного средства.

3. Когда следует использовать наследование по моделям `protected` и `private`?

   Ответ:

   При наследовании класса ради реализации, а не дополнения интерфейса следует использовать наследование `private` или `protected`, чтобы реализация оставалась скрытой.

4. Спроектируйте и реализуйте абстрактный класс `Shape`. Что следует включить в класс? Что следует оставить для реализации в порожденных классах?

   Ответ:

   Все методы должны быть реализованы в порожденных классах. Абстрактный класс должен лишь определять интерфейс объекта. Класс `Shape` может выглядеть, например, так:

```
class Shape
{
```

```
    //выводит геометрическую фигуру на экран.
    virtual void Draw() = 0;
};
```

Глава 9

1. **Создайте вектор, содержащий набор векторов, каждый из которых хранит набор целых чисел.**

   Ответ:

```
typedef vector<int> Vi
vector<Vi> vvi;
```

2. **Создайте класс-шаблон** `store`**, который позволяет хранить массив элементов** `T` **(**`T` **является параметром шаблона).**

   Ответ:

```
template<class T> Store
{
    T array[5];
};
```

3. **Создайте итератор** `random_iterator`**, который использует другой итератор для произвольного доступа к элементам контейнера.**

   Ответ:

```
template<class Iter> class random_iterator :
public iterator<iterator_traits<Iter>::iterator_category,
iterator_traits<Iter>::value_type,
iterator_traits<Iter>::difference_type,
iterator_traits<Iter>::pointer,
iterator_traits<Iter>::reference> {

protected:
    //скрытый стандартный
    //итератор для служебных целей
    Iter current;
    int size;
public:
    //стандартное имя для типа итератора
    typedef Iter iterator_type;

    //конструктор по умолчанию
    random_iterator() : current() {srand(time(0));}
    //конструктор для обычного итератора
    random_iterator(Iter x, int y) : current(x), size(y) {}
    //конструктор для другого итератора произвольного доступа
    template<class U> random_iterator
        (const random_iterator<U>& x) : current(x.base()) {}

    //вернуть обычный итератор класса
```

```
    Iter base() const {return current;}

    reference operator* () const { Iter tmp = current;
        return *–tmp;} //разыменование
    pointer operator-> () const; //оператор компонентного доступа
    reference operator[] (difference_type n) const;

    random_iterator& operator++ () {return
        current[rand()%(size + 1)] } // (случайный)
    random_iterator& operator– () { return
        current[rand()%(size + 1)];} // (случайный)

    random_iterator operator+ (difference_type n) const;
    random_iterator operator+= (difference_type n);
    random_iterator operator- (difference_type n) const;
    random_iterator operator -= (difference_type n);
};
```

4. **Назовите три места (не считая этой книги), где можно быстро получить информацию о структурах стандартной библиотеки.**

   Ответ:

   - Исходные файлы
   - *http://www.cplusplus.com*
   - Файлы справки компилятора

Глава 10

1. **Создайте программу, записывающую приведенные ниже строки текста в файл «Question1.txt».**

```
Программировать весело.
Мне нравится программировать.
```

   Ответ:

```
#include <fstream.h>

int main () {

  //открываем файл
  ofstream file("Question1.txt");

  //проверяем, открыт ли файл
  if (file.is_open())
  {
    //записываем две строки в файл
    file << "Программировать весело.\n";
    file << "Мне нравится программировать.\n";

    //закрываем файл
    file.close();
  }
```

```
    return 0;
}
```

2. Каким образом можно определить, что достигнут конец файла? Каким образом можно определить, что при работе с файлом возникла ошибка? Каким образом можно определить, что достигнут конец файла или возникла ошибка?

   Ответ:

   - `eof()` используется для определения того, что достигнут конец файла.
   - `fail()` и `bad()` используются для определения того, что при работе с файловым потоком возникла ошибка.
   - `good()` используется для проверки и того и другого обстоятельства.

3. Каков результат поразрядного сдвига влево (`<<3`) для символа A? Каков результат поразрядного сдвига вправо (`>>2`) для символа A?

   Ответ:

   До сдвигов A имеет значение 01000001 (или 65). Сдвиг на три позиции влево превращает A в 00001000, сдвиг на две позиции вправо – в 00010000.

4. Объясните, почему для расшифровки исходного файла требуется повторно выполнить программу шифрования.

   Ответ:

   Первый проход шифрования просто меняет местами первые и последние четыре бита каждого байта. Обратная перестановка (повторный проход) позволяет вернуться к исходному файлу.

Глава 11

1. Для чего нужно ключевое слово `try`?

   Ответ:

   Используйте `try` для указания, что в блоке кода может быть сгенерировано исключение. Применение `try` позволяет осуществлять перехват исключений.

2. Каково назначение иерархий исключений?

   Ответ:

   Рост числа классов исключений может затруднить обработку исключений, но малое число обобщенных классов не всегда является оптимальным решением. Иерархия исключений позволяет выбирать степень детализации при перехвате и обработке исключений.

3. Каким образом можно создавать программы, которые не совершают ошибок?

   Ответ:

   Невозможно создать программу, которая никогда не дает сбоев. Однако внимательное проектирование, применение обработки исключений, а также качественная отладка определенно способствуют повышению живучести программ.

4. Дайте определение исключения.

   Ответ:

   Исключение – это нестандартная ситуация, с которой неспособен справиться блок кода.

5. На какой стадии разработки программы следует использовать утверждения?

   Ответ:

   Следует использовать утверждения в процессе разработки программы, но удалить их после отладки и перед опубликованием.

Глава 12

1. Создайте программу Windows, отображающую окружность, которая каждую секунду изменяет свои координаты.

   Ответ:

```
#include <windows.h>
#include <cstdlib>
#include <ctime>
using namespace std;

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam);

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
                       LPSTR lpszCmdLine, int nCmdShow)
{
     HWND          hWnd;
     MSG               msg;
     WNDCLASSEX     wc;

     //инициализация структуры WNDCLASSEX
     //подходящими значениями
     wc.cbSize = sizeof(WNDCLASSEX);
     wc.style = CS_HREDRAW | CS_VREDRAW;
     wc.lpfnWndProc = WndProc;
     wc.cbClsExtra = 0;
     wc.cbWndExtra = 0;
     wc.hInstance = hInst;
```

```
    wc.hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "RandomBall";
    wc.hIconSm = LoadIcon(NULL, IDI_EXCLAMATION);

    //регистрируем новый класс
    RegisterClassEx(&wc);

    //создаем окно
    hWnd = CreateWindowEx(
        NULL,
        "RandomBall",
        "Мигрирующая окружность",
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        50,
        50,
        600,
        600,
        NULL,
        NULL,
        hInst,
        NULL
    );

    //цикл обработки событий - диспетчер сообщений
    while(GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    //стандартное возвращаемое значение
    return (msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
{
    //статические переменные позволяют
    //отслеживать положение окружности
    static int oldX, oldY;

    //контекст устройства и кисть для рисования
    HDC hDC;
    HBRUSH brush;

    //определяем, какое получено сообщение
    switch(nMsg)
    {
        case WM_CREATE:
            //создаем таймер (1 секунда)
            SetTimer(hWnd, 1, 1000, NULL);
    srand(time(0));
```

```
                break;

            case WM_TIMER: //таймер сработал (только один)
                //получаем контекст устройства для рисования
                hDC = GetDC(hWnd);
                //чистый белый цвет
                brush = (HBRUSH)SelectObject(hDC,
                    GetStockObject(WHITE_BRUSH));

                //инициализируем объект RECT
                //подходящими значениями
                RECT temp;
                temp.left = rand()%600;
                temp.top = rand()%600;
                temp.right = temp.left + 30;
                temp.bottom = temp.top + 30;
                RECT temp1 = {oldX,oldY,oldX+30,oldY+30};

                //стираем предыдущий эллипс
                FillRect(hDC, &temp1, brush);
                //готовимся рисовать новый эллипс
                brush = (HBRUSH)SelectObject(hDC,
                    GetStockObject(GRAY_BRUSH));
                //рисуем
                Ellipse(hDC, temp.left, temp.top,
                    temp.right, temp.bottom);

                //предыдущая кисть
                SelectObject(hDC, brush);
                //освобождаем контекст устройства
                ReleaseDC(hWnd, hDC);
                break;
    case WM_DESTROY:
                //уничтожаем таймер
                KillTimer(hWnd, 1);
                //конец программы
                PostQuitMessage(0);
                break;

            default:
                //остальные события - на усмотрение системы Windows
                return(DefWindowProc(hWnd, nMsg,
                    wParam, lParam));
        }

        return 0;
    }
```

2. **Назовите четыре вида сообщения, которые может получать** `WndProc`**.**

   Ответ:

   `WM_PAINT`, `WM_CREATE`, `WM_DESTROY`, `WM_TIMER`.

3. Каковы основные шаги в процессе создания окна?

   Ответ:

4. Инициализация `WNDCLASSEX`.
5. Регистрация нового класса окон.
6. Создание окна.
7. Отображение окна.
8. Назовите по памяти пять виртуальных кодов клавиш.

   Ответ:

   `VK_SHIFT`, `VK_RETURN`, `VK_UP`, `VK_LEFT`, `VK_RIGHT`.

Глава 13

1. Приведите два способа отображения растрового bmp-файла на экране.

   Ответ:

   Можно создать временную плоскость при помощи `DDLoadBitmap`, а затем скопировать временную поверхность в основную либо создать дескриптор растрового изображения при помощи функции `LoadImage`, а затем скопировать изображение в плоскость посредством функции `DDCopyBitmap`.

2. Назовите семь составляющих библиотеки DirectX.

   Ответ:

   DirectDraw, DirectSound, Direct3D, DirectInput, DirectMusic, DirectSetup, DirectPlay.

3. Перечислите шаги создания плоскостей DirectDraw.

   Ответ:

   1. Инициализация объекта `DDSURFACEDESC2` информацией о поверхности.
   2. Вызов метода `CreateSurface` для создания поверхности.
   3. Вызов метода `GetAttachedSurface` для получения доступа к вторичному буферу.

4. Что представляет первичная плоскость?

   Ответ:

   Первичная плоскость представляет экран.

B

Восьмеричная, шестнадцатеричная, двоичная и десятичная системы счисления

Для большинства людей стандартной системой счисления является *десятичная система* (по основанию 10). Все числа этой системы основаны на степенях числа 10. Каждая цифра может иметь десять значений (1, 2, 3, 4, 5, 6, 7, 8, 9 или 0). Учитывая все размещения, две цифры десятичной системы позволяют охватить 100 различных значений (от 1 до 99 плюс 0), по такому же принципу работают все остальные системы.

В *двоичной системе счисления* (по основанию 2) две двоичных цифры дают четыре варианта (эквивалентные десятичные значения приведены в скобках): 00 (1), 01 (1), 10 (2), 11 (3).

Шестнадцатеричная система счисления (по основанию 16) находит применение в компьютерах (а значит, полезна для программистов). Каждая цифра может иметь 16 значений (0–9 и A–F). В десятичной системе счисления лишь десять уникальных цифр, поэтому для представления 16 уникальных значений приходится задействовать еще пять первых букв алфавита. A соответствует числу 10, B числу 11, C числу 12, D числу 13, E числу 14, а F числу 15. Шестнадцатеричные числа записываются с префиксом *0x*; так 0x3E1 является шестнадцатеричным числом.

Восьмеричная система (по основанию 8) также полезна в компьютерной технике. Каждая цифра может иметь восемь значений (0–7). Восьмеричные числа записываются с префиксом 0 – например 075.

Преобразование в десятичную систему

Пусть вас не пугают многочисленные системы счисления. Старые добрые числа на месте, просто они обозначены по-другому. Так как же

преобразовать число из определенной системы счисления в десятичную? Очень легко. Следует взять первую цифру (первую справа) и умножить ее на *n* (основание системы) в степени номера цифры, уменьшенной на единицу. То есть значение первой цифры умножается на `n^0`, значение второй на `n^1`, третьей – на `n^2`, и т. д. Затем полученные числа необходимо просуммировать.

Для примера рассмотрим преобразование шестнадцатеричного числа **0x3E1**. Помните, что префикс `0x` просто является указанием на систему счисления и сам в преобразовании не участвует. Кроме того, не забудьте, что **E** в шестнадцатеричной системе обозначает число **14**.

```
1 x 16^0 = 1
E x 16^1 = 224
+3 x 16^2 = 768
```

Теперь суммируем числа и окончательно получаем десятичный результат – **993**.

Преобразование из десятичной системы

Процесс преобразования из десятичной системы в систему с основанием *n* не так прост. Прежде всего, разделите число на основание целевой системы, возведенное в степень номера цифры. Номер первой колонки – 1, второй – 2, и т. д. Вычтите остаток из исходного числа. Чтобы найти значение цифры в этой колонке, разделите остаток на основание, возведенное в степень, на единицу меньшую номера текущей цифры. Звучит, возможно, весьма запутанно, но на практике выглядит существенно проще. Для примера выполним преобразование числа **993** обратно в шестнадцатеричную систему:

Колонка номер 1:

```
993 / 16^1 = 62 Остаток: 1
Шестнадцатеричная цифра = 1 / 16^0 = 1
Шестнадцатеричное число на данном этапе: 1
993 - 1 = 992
```

Колонка номер 2:

```
992 / 16^2 = 3 Остаток: 224
Шестнадцатеричная цифра = 224 / 16^1 = 14 (E)
Шестнадцатеричное число на данном этапе: E1
992 - 224 = 768
```

Колонка номер 3:

```
768 / 16^3 = 0 Остаток: 768
Шестнадцатеричная цифра = 768 / 16^2 = 3
Шестнадцатеричное число на данном этапе: 3E1
768 - 768 = 0
```

Вот и все! Шестнадцатеричное число – **0x3E1**.

C

Стандартная таблица символов ASCII

В мире компьютеров каждому символу поставлено в соответствие число. Числа позволяют компьютерам хранить символы в памяти (поскольку компьютеры умеют работать только с числами). Каждый символ может иметь 256 значений, но только 128 из них являются стандартными.

В табл. C.1 перечислены значения стандартных символов. Соответствующее каждому символу значение приводится в десятичной (по основанию 10), восьмеричной (по основанию 8), шестнадцатеричной (по основанию 16) и двоичной (по основанию 2) системах счисления. Информация о различных системах счисления содержится в приложении B «Восьмеричная, шестнадцатеричная, двоичная и десятичная системы счисления». Любое из значений может быть сохранено в переменной типа `char`. К примеру, присвоение значения `65` переменной типа `char` равноценно присвоению значения `'A'`.

Таблица C.1. Стандартная таблица символов ASCII

Десятичное	Восьмеричное	Шестнадцатеричное	Двоичное	Значение
000	000	000	00000000	NUL (пустой символ)
001	001	001	00000001	SOH (начало заголовка)
002	002	002	00000010	STX (начало текста)
003	003	003	00000011	ETX (конец текста)
004	004	004	00000100	EOT (конец передачи)
005	005	005	00000101	ENQ (запрос)
006	006	006	00000110	ACK (подтверждение)
007	007	007	00000111	BEL (сигнал)
008	010	008	00001000	BS (забой)

Десятичное	Восьмеричное	Шестнадцатеричное	Двоичное	Значение
009	011	009	00001001	HT (горизонтальная табуляция)
010	012	00A	00001010	LF (перевод строки)
011	013	00B	00001011	VT (вертикальная табуляция)
012	014	00C	00001100	FF (перевод страницы)
013	015	00D	00001101	CR (возврат каретки)
014	016	00E	00001110	SO (последовательный ввод) (сдвиг)
015	017	00F	00001111	SI (последовательный вывод) (сдвиг)
016	020	010	00010000	DLE (управляющая последовательность для канала передачи данных)
017	021	011	00010001	DC1 (XON) (управление устройством 1)
018	022	012	00010010	DC2 (управление устройством 2)
019	023	013	00010011	DC3 (XOFF) (управление устройством 3)
020	024	014	00010100	DC4 (управление устройством 4)
021	025	015	00010101	NAK (отрицательное подтверждение)
022	026	016	00010110	SYN (синхронизация)
023	027	017	00010111	ETB (конец блока передачи)
024	030	018	00011000	CAN (отмена)
025	031	019	00011001	EM (конец носителя)
026	032	01A	00011010	SUB (подстановка)
027	033	01B	00011011	ESC (выход)
028	034	01C	00011100	FS (разделитель файлов)
029	035	01D	00011101	GS (разделитель групп)
030	036	01E	00011110	RS (запрос отправки) (разделитель записей)
031	037	01F	00011111	US (разделитель блоков)
032	040	020	00100000	SP (пробел)

Десятичное	Восьмеричное	Шестнадцатеричное	Двоичное	Значение
033	041	021	00100001	!
034	042	022	00100010	"
035	043	023	00100011	#
036	044	024	00100100	$
037	045	025	00100101	%
038	046	026	00100110	&
039	047	027	00100111	'
040	050	028	00101000	(
041	051	029	00101001	)
042	052	02A	00101010	*
043	053	02B	00101011	+
044	054	02C	00101100	,
045	055	02D	00101101	-
046	056	02E	00101110	.
047	057	02F	00101111	/
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	<
061	075	03D	00111101	=
062	076	03E	00111110	>
063	077	03F	00111111	?
064	100	040	01000000	@
065	101	041	01000001	A

Десятичное	Восьмеричное	Шестнадцатеричное	Двоичное	Значение
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01010111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[
092	134	05C	01011100	\
093	135	05D	01011101	]
094	136	05E	01011110	^
095	137	05F	01011111	_
096	140	060	01100000	`
097	141	061	01100001	a
098	142	062	01100010	b

Десятичное	Восьмеричное	Шестнадцатеричное	Двоичное	Значение
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	156	06E	01101110	n
111	157	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	z
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{
124	174	07C	01111100	|
125	175	07D	01111101	}
126	176	07E	01111110	~
127	177	07F	01111111	DEL (удаление)

D

Ключевые слова C++

В языке C++ на удивление мало ключевых слов. Но и тех 70 с хвостиком, которые есть, более чем достаточно для решения практически любой задачи, которая может возникнуть при создании программ. В табл. D.1 приведен перечень всех ключевых слов языка C++. Пользуйтесь таблицей как справочником либо как средством для расширения своего словаря C++.

Таблица D.1. Ключевые слова C++

Ключевое слово	Описание
`and`	Синоним оператора `&&`
`and_eq`	Синоним оператора `&=`
`asm`	Используется для вставки в программы кода на языке ассемблера
`auto`	Используется для хранения переменных с автоматическим выделением памяти
`bitand`	Синоним оператора `&`
`bitor`	Синоним оператора `|`
`bool`	Логический тип данных
`break`	Прерывает работу оператора `case`
`case`	Оператор управления выполнением
`catch`	Используется для регистрации ошибок
`char`	Символьный тип данных
`class`	Абстрактный тип данных
`compl`	Синоним оператора `~`
`const`	Используется для объявления констант
`const_cast`	Используется для преобразования констант в неконстанты

Ключевое слово	Описание
continue	Осуществляет переход к следующей итерации цикла
default	Вариант по умолчанию в операторе switch
delete	Используется для освобождения памяти, динамически выделенной под объекты
do	Применяется в циклах do while
double	Вариант float с повышенной точностью
dynamic_cast	Определяет принадлежность объекта к определенному типу
else	Используется совместно с оператором if
enum	Объявляет перечислимый тип данных
explicit	Запрещает неявный вызов конструктора
export	Делает переменную доступной из другого файла
extern	Используется для импортирования функций и классов из других файлов
false	Представляет бит со значением 0
float	Тип переменной для хранения действительных чисел
for	Оператор цикла for
friend	Используется для предоставления определенным классам и функциям специальных полномочий доступа к классам
goto	Оператор управления выполнением
if	Используется для создания конструкции условного оператора if
inline	Заменяет вызов функции ее кодом, работает аналогично макроподстановкам
int	Целочисленный тип данных
long	Синоним int
mutable	Модификатор, разрешающий изменение поля const-объекта класса
namespace	Используется для разграничения области видимости
new	Создает новый объект с динамическим выделением памяти
not	Синоним оператора !
not_eq	Синоним оператора !=
operator	Используется для перегрузки операторов
or	Синоним оператора ||
or_eq	Синоним оператора |=
private	Указывает, что следующие поля являются скрытыми в классе

Ключевое слово	Описание
protected	Указывает, что следующие поля класса защищены от доступа извне
public	Указывает, что следующие поля класса доступны извне
register	Модификатор переменной, предписывающий компилятору оптимизировать ее для частого обращения
reinterpret_cast	Используется для явного приведения неродственных типов
return	Возвращает значение функции
short	Целочисленный тип данных, занимающих вдвое меньше памяти, чем int
signed	Указывает, что целочисленная переменная может принимать как положительные, так и отрицательные значения
sizeof	Используется для определения размера переменной в памяти
static	Используется для объявления статических полей класса
static_cast	Используется для явного приведения родственных типов
struct	Используется для создания структур (структуры схожи с объектами)
switch	Используется для создания конструкции условного оператора switch
template	Используется для объявления шаблонов
this	Указывает на адрес текущего объекта в памяти
throw	Используется для генерации исключения
true	Представляет бит со значением 1
try	Используется для контроля ошибок
typedef	Определяет псевдоним типа
typeid	Используется аналогично sizeof для определения типа выражения
typename	Используется для доступа к полям параметра-шаблона
union	Разновидность структуры, которая способна хранить только одно поле единовременно
unsigned	Указывает, что целочисленная переменная может принимать только положительные значения
using	Используется для включения указанного пространства имен в глобальное
virtual	Используется для объявления виртуальных функций
void	Указывает, что функция ничего не возвращает

Ключевое слово	Описание
volatile	Модификатор, указывающий компилятору, что значение объекта может изменяться извне программы C++
wchar_t	Символьный тип, позволяющий хранить больше чем 256 стандартных символов
while	Используется для создания конструкций while и do while
xor	Синоним оператора ^
xor_eq	Синоним оператора ^=

E

Содержимое компакт-диска

Компакт-диск, прилагаемый к этой книге, содержит полный дистрибутив набора средств для разработчика Microsoft DirectX 8. Кроме того, на диске содержатся файлы примеров.

Работа с компакт-диском в системах Windows 95/98/2000/NT

В целях упрощения работы с диском и экономии дискового пространства для его просмотра не требуется установка. Это означает, что на жесткий диск будут скопированы только те файлы, которые вы сами захотите скопировать или установить. HTML-страницы с графикой могут быть прочитаны в любой операционной системе, однако исполняемые программы будут работать только в системе Windows.

При включенном режиме автозапуска компакт-диска его HTML-интерфейс автоматически открывается в стандартном броузере.

Если этот режим выключен, выполните следующие шаги, чтобы получить доступ к диску:

1. Поместите диск в устройство чтения и закройте лоток.
2. Перейдите в папку Мой компьютер (My Computer) или запустите Проводник (Windows Explorer), а затем дважды щелкните по пиктограмме устройства чтения компакт-дисков.
3. Найдите и откройте файл start_here.html (это работает для большинства веб-броузеров).

В первом окне, которое вы увидите, содержится лицензионное соглашение издательства Premier Press. Прочтите его и, если вы согласны с условиями, нажмите кнопку «I Agree», чтобы перейти к пользовательскому интерфейсу. Если не согласны с условиями соглашения, нажмите кнопку «I Disagree». В этом случае компакт-диск не будет доступен.

Пользовательский интерфейс Premier Press

Первый экран пользовательского интерфейса Premier Press содержит кнопки навигации и информационную область. Кнопки навигации расположены в левой части окна броузера. Нажатием кнопок можно переходить в нужные разделы диска. После загрузки запрошенной страницы ее содержимое отображается в правой части окна.

Например, чтобы получить доступ к исходным текстам, нажмите на кнопку Source Code. Открывшаяся страница содержит ссылки на все исходные файлы, размещенные на диске. Файлы каждой главы сжаты для удобства распространения. Их можно распаковать с помощью программы работы с zip-архивами. Установить программу WinZip можно со страницы Programs. Распакованные файлы доступны в каталоге /SourceCode компакт-диска. Каждой главе отведен дополнительный подкаталог.

Изменение размера и завершение работы

Чтобы изменить размер окна, расположите указатель мыши над любой его границей или углом, нажмите кнопку и, не отпуская ее, измените положение угла или границы. Получив приемлемые размеры, отпустите кнопку мыши.

Чтобы завершить работу с пользовательским интерфейсом, выберите пункт Exit из меню File.

Глоссарий

ANSI (American National Standards Institute, Американский Национальный Институт Стандартов) – организация, разрабатывающая стандарты компьютерной индустрии.

ASCII, код (American Symbolic Code for Information Interchange, Американский стандартный код для обмена информацией) – код, определяющий набор из 256 уникальных символов для представления компьютерных данных.

blit (binary linear transfer, бинарный линейный перенос, «блит») – действие по переносу блока памяти (обычно в программировании графики DirectX), не требующее затрат процессорного времени.

boolean (логический) – один из основных типов данных; представляет два возможных значения: `true` (истинное) или `false` (ложное).

bug («баг») – компьютерная ошибка.

DirectDraw – составляющая библиотеки DirectX, отвечающая за двумерную графику.

DirectX – библиотека Microsoft; предназначена в основном для работы с графикой и реализации взаимодействия пользователей с играми. DirectX позволяет программистам обращаться напрямую к аппаратным средствам компьютеров конечных пользователей, не задумываясь при этом о совместимости.

namespace – ключевое слово C++, применяемое для деления областей видимости на более мелкие.

try-block – раздел кода, в котором допускается перехват генерируемых исключений.

Win32 – 32-битное приложение Windows, в котором задействованы библиотеки Windows, а не 16-битной системы DOS.

Windows, сообщения – сообщения, получаемые Win32-программой от системы Windows. Каждое сообщение отражает произошедшее событие (например, нажатие пользователем кнопки мыши).

абстрактный класс (abstract class) – класс с чистой виртуальной функцией. Объекты абстрактных классов создавать запрещено.

автоматическая память – память, в которой по умолчанию хранятся все локальные переменные.

адрес – уникальное число, отражающее расположение ячейки памяти.

алгоритм – логика решения определенной задачи.

аргумент – данные, передаваемые вызываемой функции вызывающей.

базовый класс (также суперкласс или родительский класс) – класс, порождающий другие классы.

байт – единица хранения информации, состоящая из 8 битов; может представлять одно из 256 возможных значений.

беззнаковая переменная – переменная целочисленного типа, которая может хранить только положительные значения.

библиотеки – наборы скомпилированных функций, которые можно применять в своих программах.

бинарный файл – файл, в котором данные хранятся в сжатом формате.

бит – наименьшая единица хранения информации; может иметь значения 0 и 1.

блок – раздел кода C++, заключенный в фигурные скобки.

булева логика – изучение отношений логических типов данных.

варианта, оператор – оператор, в котором происходит выбор исполняемого блока кода на основе выражения в ключевом слове `switch`.

ввод/вывод (I/O) – термин описывает способ общения компьютера с внешним миром. Устройства ввода – мышь, клавиатура и т. д. Устройства вывода – монитор, принтер, колонки и т. д.

вектор – сложная структура данных, реализуемая стандартной библиотекой. Содержится в библиотеке `<vector>`.

виртуальные функции – полиморфные функции, позволяющие ссылаться на нужный класс при вызове.

вложенность – наличие управляющего оператора (например, `if`) в теле другого управляющего оператора.

возвращаемое значение – данные, возвращаемые в вызывающую функцию вызываемой. Часто используется для передачи численных результатов вычислений, выполняемых функциями.

вставка – помещение байтов в поток ввода-вывода.

вставки, методы – методы, выполняющие вставку в поток.

выделение памяти – объявление и резервирование определенного участка памяти.

вызов – действие, приводящее к выполнению функции в определенной точке программы.

вызывающая процедура – функция, вызывающая другую функцию.

главная функция – функция, которая выполняется первой при старте программы. Наиболее распространенный прототип: `int main(void)`.

глобальная переменная – переменная, доступная в любой точке исходного файла.

двоичная система – система счисления, основанная на цифрах 0 и 1.

декремента, оператор – оператор `--`, уменьшающий значение переменной на единицу.

деструктор – функция класса, автоматически выполняемая при

уничтожении объекта. Имя деструктора представляет собой имя класса, предваренное символом ~.

динамическая память (память свободного хранения, «куча») – область памяти, в которой программист может сохранять новые данные. Память выделяется и освобождается при помощи операторов `new` и `delete`.

дисковая память – память, позволяющая организовывать полупостоянное хранение данных – жесткие и гибкие диски, приводы Zip и CD-ROM и т. д.

доступа, модификаторы (access specifiers) – указывают область видимости компонентов класса.

заголовочные файлы – файлы, включаемые в программы C++. Часто представляют файлы стандартной библиотеки C++.

затенение – объявление локальной переменной, идентификатор которой совпадает с идентификатором глобальной переменной. При обращении к такой переменной используется локальная переменная.

идентификаторы – имена, назначаемые переменным с целью последующего использования в программе.

извлечение – перемещение байтов из потока.

извлечения, метод – метод, выполняющий извлечение.

имеет тип – отношение между двумя элементами, один из которых представляет категорию, а второй – объект из этой категории.

именующие выражения (lvalues) – выражения, которым могут присваиваться значения и которые могут находиться в левой части оператора присваивания.

индекс – число, используемое для обращения к конкретному элементу массива.

инициализации, список; массивы – список значений, разделенных запятыми; используется для инициализации массива.

инициализации, список; объекты – список значений, инициализирующих поля данных объекта; используется в конструкторе.

инкапсуляция – принцип ООП, согласно которому каждый объект отвечает за решение одной и только одной задачи.

инкремента, оператор – оператор ++, увеличивающий значение переменной на единицу.

Интегрированная среда разработки (Integrated Development Environment, IDE) – графический интерфейс, объединяющий компилятор, средство файловой навигации, средства настройки и редактор исходных текстов.

исключений, обработка – обработка внештатных ситуаций, возникающих в процессе работы программы.

исходный текст – текстовое представление программы, написанной на языке программирования, таком как C++.

квадратное уравнение – любое уравнение, которое можно представить в виде `ax^(2) + bx + c`

килобайт – единица хранения информации, состоит из 1024 байт;

одна из основных единиц измерения объема памяти.

класс – структура, определяющая свойства объекта, включая поля данных и компоненты-функции.

класс-обертка – класс, расширяющий функциональность примитивного типа данных или объекта.

классов, иерархия – древовидная структура, отражающая отношения наследования для классов.

класс-потомок – см. *порожденный класс*.

класс-шаблон – класс, предназначенный для работы с произвольными типами данных; для достижения подобной функциональности используются шаблоны.

кода, фрагмент – раздел программы.

команда – инструкция языка программирования.

комментарий – информативный текст, включаемый в исходный текст для удобства автора либо постороннего читателя программы. Комментарии игнорируются компилятором.

компилятор – преобразует исходный текст в исполняемый формат.

компиляция – преобразование компилятором исходного текста в исполняемый формат.

компоновка (linking) – проверка работоспособности кода с учетом всех файлов, включенных в программу (не только файлов программы, но и файлов внешних библиотек, таких как DirectX).

консольное приложение – приложение, работающее в текстовом DOS-окне без использования библиотек Windows.

конструктор – функция, имя которой совпадает с именем класса. Конструктор выполняется каждый раз при создании объекта-потомка. Используется для выделения памяти и присвоения начальных значений свойствам объекта.

куча (heap) – см. *динамическая память*.

литералы – способ представления конкретных значений в программе. Литеральным представлением для чисел является арабская система нумерации (1, 2, 3 и т. д.); строки представляются последовательностями букв и цифр, заключенными в двойные кавычки.

локальная переменная – переменная, объявленная в теле функции; доступна только в пределах функции.

макроопределения – директивы, позволяющие присвоить имя выражению или иному элементу программы.

манипуляторы – обрабатывают данные потока определенным образом. К примеру, манипулятор может использоваться для изменения регистра всех символов или преобразования десятичных чисел в шестнадцатеричные.

массив, многомерный – массив массивов.

массив – список элементов данных, к которым можно обращаться по индексу.

математический оператор – оператор, выполняющий математи-

ческую операцию (например, сложение или вычитание).

машинный язык – скомпилированная, исполняемая версия программы.

множественное наследование – порождение класса на базе нескольких предков.

наследование – порождение одного класса от другого.

наследования, цепь – линия наследования классов.

наследовать – создавать классы на базе других классов.

нулевой указатель (не инициализированный указатель) – указатель, который имеет нулевое значение; считается, что он не указывает на данные.

область видимости – диапазон, в котором переменная доступна окружающему коду программы.

объект – конкретный экземпляр класса.

объектно-ориентированное программирование (ООП) – методология программирования, манипулирующая данными с помощью объектов, которые содержат как информацию, так и код для работы с ней.

объявление – передача компьютеру информации об имени и атрибутах переменной, функции или класса.

одиночное наследование – порожденный класс наследует свойства только одного базового класса.

операнды – аргументы оператора.

оперативная память (RAM) – память временного (до выключения компьютера) хранения данных.

оператор – символ или двойной символ (`<=`), а в некоторых случаях целое слово, например `sizeof()`, который предписывает компьютеру выполнить определенное действие.

операторов, перегрузка – создание новых применений существующих операторов.

операция – фраза языка программирования, содержащая оператор и его операнды.

освобождение памяти – передача компьютеру сообщения, что определенная область памяти больше не нужна для работы программы.

основные типы – типы данных, встроенные в язык C++.

остатка, оператор (%) – вычисляет остаток от деления x на y (`x % y`).

отладка – процесс исправления ошибок в программе.

память – устройство, используемое для хранения данных и обращения к ним. Чаще всего применяются жесткие диски, дискеты, а также оперативная память (RAM).

параметр – см. *аргумент*.

перебора, управляющие операторы – управляющие операторы, позволяющие многократно выполнять код по условию.

перегрузка – создание нескольких версий функции с одним именем. Какую версию вызывать, компьютер определяет исходя из аргументов.

переменная – именованная область памяти, в которой хранятся данные программы.

переменной, время жизни – время, в течение которого сохраняется резервирование памяти под хранение переменной.

переменной, область видимости – диапазон кода программы, в котором доступна конкретная переменная.

переопределение – реализация в порожденном классе метода, имя которого совпадает с именем унаследованного от базового класса. При работе с объектом порожденного класса предпочтение отдается унаследованной версии.

перехода, операторы – ключевые слова, позволяющие выполнять переход из одной точки программы в другую, пропуская фрагменты кода.

плоскостей, переброска – смена значения графического указателя с одного объекта плоскости DirectDraw на другой.

плоскость DirectDraw – область памяти, используемая для хранения изображений.

подкласс – см. *порожденный класс*.

подстрока – строка, которая является частью другой строки.

полиморфизм – принцип ООП, согласно которому каждый объект может использоваться в более чем одной программе.

порожденный класс (также подкласс, производный класс или класс-потомок) – класс, созданный на основе базового класса и наследующий все его свойства.

порядок действий – расстановка приоритетов операторов. К примеру, оператор + имеет более низкий приоритет, чем оператор `*`. Следовательно, он выполняется позже, чем оператор `*`, если только не предписано иное, к примеру, при помощи скобок, которые имеют более высокий приоритет, чем умножение.

порядок старшинства – см. *порядок действий*.

порядок – правило роста потребностей алгоритма.

потоковый объект – выступает в роли как источника, так и адресата данных в операциях ввода-вывода. Строковый объект работает с упорядоченной линейной последовательностью байтов.

программа – последовательность инструкций, выполняемых компьютером.

программирование – процесс набора исходного кода, который впоследствии компилируется и выполняется.

производная от – другое название отношения *имеет тип*.

прототип (или прототип функции) – представление функции; позволяет компилятору подготовиться к работе с функцией. Прототип включает тип возвращаемого значения, имя функции и тип аргументов.

пустой символ – завершающий символ строки; представляется специальным символом `\0`.

разработки, цикл – основные этапы создания и отладки программы.

реализация – определение функции.

рекурсия – прием программирования. Рекурсия происходит, когда функция вызывает саму себя.

родительский класс – см. *базовый класс*.

свободное хранение (free store) – см. *динамическая память*.

символ (char) – переменная, которая может хранить один из 256 различных символов набора ASCII.

скалярная переменная – переменная, способная хранить лишь один атом информации, в отличие от массива.

слияние или **конкатенация (concatenation)** – объединение двух или более строк.

сложность – мера эффективности алгоритма.

случайные числа – числа, последовательность которых не имеет выраженной зависимости. Случайные числа генерируются по сложным математическим формулам.

специальный символ (метасимвол) – символ, который не отображается на экране, но используется для представления других символов либо определения форматирования. В число таких символов входят табуляция, символ новой строки и другие.

статическая память – область памяти, в которой компилятор размещает статические и глобальные переменные.

статические переменные – переменные, которые сохраняют значения на протяжении всей программы, но не обязательно имеют глобальную область видимости.

строка (string) – объектный тип, позволяющий хранить последовательность символов в символьном массиве, завершаемым *пустым символом*.

строка в стиле C – массив символов, завершаемый нулевым символом.

строки, разрыв – специальный символ, представляющий начало новой строки.

суперкласс – см. *базовый класс*.

текстовый файл – файл, данные в котором представлены символами.

тестовый модуль – функция, применяемая для проверки работоспособности объекта.

тип – представление данных, таких как целые и действительные числа.

типа, альтернативное имя (псевдоним) – другое имя типа; часто определяется при помощи ключевого слова `typedef`.

точка конкретизации – момент создания класса-шаблона или функции-шаблона для конкретного набора параметров шаблона.

указатель – переменная, в которой хранится адрес значения в памяти.

управляющие символы (escape characters) – специальный набор символов, используемых для представления неотображаемых в литеральном формате символов (например, символ новой строки).

управляющий оператор – оператор, изменяющий ход программы в зависимости от результатов проверки определенного условия.

условный оператор – см. *управляющий оператор*.

утверждение (assertion) – оператор (обычно макроопределение), который проверяет истинность определенного условия.

функции, объявление – представляет функцию программе, определяет тип возвращаемого значения, имя и аргументы функции; также называется *прототипом*.

функции, определение – реализация кода функции.

функция – раздел кода, выполняющий при вызове определенную задачу.

функция-шаблон – функция, предназначенная для работы с произвольными типами данных; для реализации используются шаблоны.

целое – тип переменной, которая может хранить только целые числа (отрицательные, положительные либо нуль).

целые числа, циклический возврат – когда целое число достигает значения, на единицу больше максимального, происходит циклический возврат к минимальному значению целочисленной переменной.

частное – результат деления двух чисел.

чистая виртуальная функция – виртуальная функция, которая не реализована и существует только с целью создания абстрактного класса.

шаблон – метод обобщения функциональности класса или функции для работы с произвольными типами данных.

шаблона, параметр – параметр, аналогичный параметру функции, который определяет тип данных.

шестнадцатеричная система – система счисления по основанию 16, в которой для представления 16 уникальных цифр используются цифры от 0 до 9 и буквы от A до F.

экспоненциальная запись – краткий формат записи чисел огромной величины.

элемент – атомарная составляющая массива.

Алфавитный указатель

E

F

G

H

I

L

M

X

А

Б

В

K

Л

М

Н

О

П

Р

С

Т

У

Ф

X

Ц

Ч

Ш

Э