МАЙКЛ ДОУСОН

ИЗУЧАЕМ C++

ЧЕРЕЗ ПРОГРАММИРОВАНИЕ ИГР

MICHAEL DAWSON

BEGINNING C++

THROUGH GAME PROGRAMMING

CENGAGE Learning

МАЙКЛ ДОУСОН

ИЗУЧАЕМ C++

ЧЕРЕЗ ПРОГРАММИРОВАНИЕ ИГР

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Киев · Екатеринбург · Самара · Минск

2016

М. Доусон

Изучаем C++ через программирование игр

Перевели с английского Е. Зазноба, О. Сивченко

Заведующий редакцией	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*С. Дегтярев*
Литературный редактор	*Н. Рощина*
Художник	*В. Шимкевич*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*Г. Блинов*

ББК 32.973.2-018.1
УДК 004.43

Доусон М.

Д71 Изучаем C++ через программирование игр. — СПб.: Питер, 2016. — 352 с.: ил.
ISBN 978-5-496-01629-2

Если вы хотите научиться программировать первоклассные игры, вам просто необходимо изучить язык C++. Эта книга поможет вам освоить разработку игр с самых азов, независимо от того, есть ли у вас опыт программирования. Гораздо интересней учиться, когда обучение превращается в игру.

Каждая глава книги описывает самостоятельный игровой проект. В заключительной главе вам предстоит написать сложную игру, которая объединяет все приемы программирования, рассмотренные в предыдущих главах.

Книга, которую вы держите в руках, идеально подойдет для начинающего программиста, планирующего не только как следует освоить непростой язык C++, но и поупражняться в программировании игр.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1305109919 англ.

ISBN 978-5-496-01629-2

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 05.10.15. Формат 70×100/16. Усл. п. л. 28,380. Доп. тираж. Заказ 5567.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография» Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

Краткое содержание

Оглавление

Благодарности

Любая книга, которую вы когда-либо читали, начинается с наглой лжи. Я решил открыть вам этот маленький нелицеприятный секрет издательской индустрии: на самом деле любую книгу создает далеко не один тот человек, имя которого красуется на обложке. Для того чтобы книга вышла в свет, требуются общие усилия целой команды профессионалов. Автор сам по себе не в состоянии издать книгу — по крайней мере, такой автор, как я. Поэтому я хотел бы поблагодарить всех коллег, без участия которых этот том никогда не появился бы на книжных полках.

Спасибо Дэну Фостеру, выполнившему двойную работу: он потрудился и как выпускающий редактор, и как редактор проекта. Дэну удалось улучшить книгу, которая до него прошла уже не одну вычитку.

Спасибо Джошуа Смиту, моему техническому рецензенту, благодаря которому все программы работают в точности так, как описано в тексте.

Спасибо Келли Тэлбот, моему корректору, хорошо причесавшей текст — до буквы.

Также я хотел бы поблагодарить Эми Смит, моего старшего управляющего редактора, за то, как она вдохновляла меня творить.

Наконец, хотел бы поблагодарить авторов всех тех компьютерных игр, в которые я резался в детстве. Они вдохновили меня поработать над моими собственными играми — сначала совсем маленькими, а затем сказать свое слово в игровой индустрии. Надеюсь, некоторые читатели этой книги смогут повторить мой путь.

Об авторе

Майкл Доусон — автор, пишущий о компьютерных играх, а также преподаватель, обучающий студентов искусству и науке создания компьютерных игр. Майкл разрабатывал и читал курсы по программированию игр на факультете UCLA Extension в Калифорнийском университете Лос-Анджелеса (студенты этого факультета получают второе высшее образование). Также Майкл читал лекции в Академии цифровых и медиатехнологий (DMA) и в Кинематографической школе Лос-Анджелеса. Книги Доусона входят в обязательную программу многих университетов и колледжей в США.

Майкл начал карьеру в игровой индустрии как продюсер и дизайнер. Параллельно с этим он стал разрабатывать приключенческую игру, где пользователь управляет главным персонажем по имени Майк Доусон. По сюжету игры цифровой двойник Доусона должен предотвратить нашествие инопланетян прежде, чем у него из головы вылупится имплантированная личинка пришельцев.

В реальности Майкл известен как автор нескольких книг: Beginning C++ Through Game Programming, Python Programming for the Absolute Beginner, C++ Projects: Programming with Text-Based Games и Guide to Programming with Python. Доусон получил степень бакалавра кибернетики в университете Южной Калифорнии. Подробнее почитать об авторе и его книгах можно на персональном сайте автора (www.programgames.com), здесь же вы найдете ответы на многие вопросы по книгам Майкла.

Введение

Ультрасовременные компьютерные игры не уступают голливудским блокбастерам по визуальным эффектам, саундтреку и градусу напряжения. Но игры — совершенно особый вид развлечений; хорошая игра может заставить геймера часами просиживать у монитора. Основная изюминка компьютерных игр, которая делает их столь увлекательными, — это интерактивность. Компьютерная игра — это не фильм, где вы наблюдаете за приключениями героя, побеждающего несмотря ни на что, — в игре вы сами перевоплощаетесь в героя.

Такая интерактивность достигается прежде всего с помощью программирования. Именно программа оживляет монстра, эскадрилью штурмовиков или целую армию, заставляет эти виртуальные сущности по-разному реагировать на действия пользователя в зависимости от ситуации. Благодаря программе игровой сюжет может принимать новые обороты. На самом деле именно на уровне программной логики игра может отвечать на действия пользователя так, что удивляет даже собственных создателей.

Хотя в настоящее время существует огромное количество языков программирования, важнейшим из них в игровой индустрии является язык C++. Если в ближайшем универмаге вы зайдете в отдел компьютерных игр и наугад возьмете диск с полки, вполне вероятно, что вся игра от первой до последней строчки будет написана на C++. Мораль: если хотите профессионально заниматься программированием игр, выучите C++.

Цель данной книги — познакомить вас с языком C++ в контексте программирования игр. Хотя невозможно представить себе такую книгу, которая сделала бы вас мастером в двух столь объемных темах, как программирование игр и язык C++, это издание пригодится вам в качестве вводного курса.

Для кого эта книга

Эта книга для всех, кто желает научиться программированию игр. Она рассчитана на начинающих, для чтения не требуется никакого опыта программирования. Если вы хорошо умеете пользоваться собственным компьютером, можете начинать одиссею в мир программирования игр прямо сейчас. Но сам факт, что эта книга рассчитана на начинающих, еще не означает, что освоить C++ для программирования игр будет проще простого. Придется внимательно читать, работать, экспериментировать. Дочитав эту книгу, вы приобретете солидные базовые знания по важнейшему языку для профессионального программирования игр.

Как построена книга

Я начинаю рассказ о программировании игр на C++ с азов, предполагая, что читатель не имеет никакого опыта ни в программировании игр, ни в C++. В каждой новой главе рассматриваются все более сложные темы, каждая последующая глава основывается на материале из предыдущих.

В каждой главе рассматриваются определенная тема или несколько взаимосвязанных тем. Я перехожу от одной концепции к другой, сопровождая текст маленькими игровыми программами, иллюстрирующими каждую идею. В конце каждой главы объединяю наиболее важные из рассмотренных в ней концепций, предлагая вашему вниманию готовую игру. Заключительная глава завершается самым нетривиальным проектом, где мы задействуем все основные феномены, изученные в книге.

Дочитав эту книгу, вы не просто освоите язык C++ и получите представление о программировании игр. Вы также научитесь организовывать свой труд, подразделять проблемы на небольшие удобоваримые подзадачи, оттачивать код. На страницах книги вам придется постоянно сталкиваться с испытаниями, но ни одно из них не будет непреодолимым. В целом вас ждет забавное и интересное обучение. По ходу дела вы напишете несколько классных компьютерных игр и постигнете ремесло игрового программирования.

Глава 1. Типы, переменные, стандартный ввод-вывод. Игра «Утраченный клад». Здесь вы познакомитесь с основами C++ — важнейшего языка программирования, который применяется в игровой индустрии. Вы научитесь отображать вывод в окне консоли, выполнять арифметические вычисления, работать с переменными и получать пользовательский ввод, набранный на клавиатуре.

Глава 2. Истина, ветвление и игровой цикл. Игра «Угадай число». Многие интересные игры основаны на программах, которые выполняют, пропускают или повторно выполняют определенные блоки кода в зависимости от тех или иных условий. Вы научитесь генерировать случайные числа, что позволит добавить в ваши игры элемент непредсказуемости. Кроме того, вы познакомитесь с фундаментальным феноменом, называемым «игровой цикл». Именно этот цикл позволяет организовать игру и выстроить в ней череду событий.

Глава 3. Циклы for, строки и массивы. Игра «Словомеска». Вы узнаете об игровых последовательностях и научитесь работать со строками. Строка — это последовательность символов, незаменимая при игре в слова. Кроме того, вы познакомитесь с программными объектами — сущностями, позволяющими представлять в вашей игре существ или предметы. Примеры сущностей — инопланетный космический корабль, горшочек с лечебным зельем или сам персонаж.

Глава 4. Библиотека стандартных шаблонов. Игра «Виселица». Здесь вы познакомитесь с мощной библиотекой — инструментарием для разработчиков игр (и не только игр). Эта библиотека позволяет объединять элементы в коллекции и хранить их в таком виде, например, как снаряжение в рюкзаке у персонажа. Кроме того, здесь будут рассмотрены приемы, позволяющие создавать более крупные игровые программы.

Глава 5. Функции. Игра «Безумные библиотекари». В этой главе вы научитесь разбивать игровые программы на компактные блоки, с которыми удобно работать. Для этого вы познакомитесь с функциями — элементарными логическими единицами, применяемыми в программах.

Глава 6. Ссылки. Игра «Крестики-нолики». Здесь мы поговорим о том, как одновременно использовать определенную информацию в разных частях вашей программы, причем делать это четко и эффективно. Кроме того, здесь вы познакомитесь с искусственным интеллектом и узнаете, как сделать компьютерного соперника немного изобретательнее.

Глава 7. Указатели. Игра «Крестики-нолики 2.0». В этой главе вы познакомитесь с некоторыми наиболее низкоуровневыми и мощными возможностями языка C++. В частности, мы поговорим о том, как обращаться непосредственно к компьютерной памяти и манипулировать ею.

Глава 8. Классы. Игра «Тамагочи». Здесь вы научитесь создавать собственные объекты и описывать, как они должны взаимодействовать друг с другом. Для этого применяется парадигма объектно-ориентированного программирования. Вы создадите собственного любимца-тамагочи, о котором будете заботиться.

Глава 9. Более сложные классы и работа с динамической памятью. Игровое лобби. В этой главе вы научитесь еще плотнее взаимодействовать с компьютером, в частности занимать и высвобождать память, если этого требует игровой процесс. Также вы узнаете о подводных камнях, связанных с таким динамическим использованием памяти, и научитесь их обходить.

Глава 10. Наследование и полиморфизм. Игра Blackjack. В этой главе мы поговорим о том, как создавать объекты с помощью других объектов. Затем обобщим весь изученный материал в большой заключительной игре. Вы увидите, как проектируется и реализуется довольно объемная игра, а именно виртуальный симулятор классической азартной карточной игры Blackjack (без потертого сукна на ломберном столе как-нибудь обойдемся).

Условные обозначения, используемые в книге

На страницах книги я пользуюсь некоторыми условными обозначениями. Например, я даю курсивом *новый термин* и объясняю, что он означает. Кроме того, я делаю в тексте следующие особые пометки.

СОВЕТ

Классные идеи, которые помогли мне лучше освоить программирование игр.

ОСТОРОЖНО!

Это скользкие места, в которых легко допустить ошибку.

ПРИЕМ

Здесь я рассказываю о секретах и уловках, которые облегчают жизнь игровому программисту.

НА ПРАКТИКЕ

Правдивые факты из мира игрового программирования.

Исходный код к книге

Весь исходный код к примерам из этой книги доступен в Интернете по адресу www.cengageptr.com/downloads.

Несколько слов о компиляторах

Возможно, рассказывая здесь о компиляторах, я немного опережаю события. Но эта проблема очень важна, поскольку именно *компилятор* преобразует написанный вами исходный код в машинно-исполняемую программу, которую способен прочитать компьютер. Если вы работаете на компьютере с Windows, рекомендую использовать среду разработки Microsoft Visual Studio Express 2013 для Windows ПК, поскольку в ней вы бесплатно получаете современный компилятор C++. Установив эту программу, почитайте приложение 1 к этой книге «Создание первой программы на языке C++». Там рассказано, как скомпилировать код C++ в Microsoft Visual Studio Express 2013 для Windows ПК. Если вы пользуетесь другими компилятором или средой разработки, почитайте документацию по вашему инструменту.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты sivchenko@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Типы, переменные, стандартный ввод-вывод. Игра «Утраченный клад»

Программирование игр — взыскательная отрасль. В этой сфере и разработчик, и машина должны работать на пределе возможностей. Но игра здесь действительно стоит свеч. Итак, в этой главе вы познакомитесь с основами C++ — важнейшего языка для программирования первоклассных игр. В частности, вы научитесь:

- отображать вывод в окне консоли;
- выполнять арифметические вычисления;
- использовать переменные для хранения данных, их извлечения и манипуляций с ними;
- получать пользовательский ввод;
- работать с константами и перечислениями;
- работать со строками.

Введение в C++

На языке C++ постоянно работают миллионы программистов по всему миру. Это один из популярнейших языков для написания компьютерных программ и важнейший язык, на котором создаются крупнобюджетные компьютерные игры.

Язык C++ изобретен Бьерном Страуструпом и непосредственно основан на языке C. Фактически почти весь язык C сохранился в C++ в виде своеобразного подмножества. Однако C++ предлагает улучшенные способы решения многих задач и включает инновационные возможности, которых не было в языке C.

Использование языка C++ при программировании игр

Существует ряд причин, по которым разработчики игр активно пользуются языком C++. Рассмотрим некоторые из них.

- **Он быстр.** Грамотно написанные программы на C++ могут работать просто молниеносно. Одной из основных проектных характеристик языка C++ была высокая производительность. Если же в вашей программе требуется добиться просто запредельной производительности, то C++ позволяет работать и с *ассемблером*. Ассемблер — это самый низкоуровневый человекочитаемый язык программирования, взаимодействующий непосредственно с аппаратным обеспечением компьютера.
- **Он гибок.** C++ — это мультипарадигмальный язык, поддерживающий различные стили программирования, в том числе *объектно-ориентированное программирование*. Но, в отличие от многих современных языков программирования, C++ не имеет жесткой привязки к какой-либо парадигме программирования.
- **Он хорошо поддерживается.** Поскольку язык C++ уже очень давно используется в игровой индустрии, по нему доступно множество ресурсов. Это и графические API, и возможности 2D и 3D, и игровая физика, и звуковые движки. Программист, работающий с C++, может использовать готовый код, значительно ускоряя разработку новых игр.

Создание исполняемого файла

Исполняемым называется файл, используемый для запуска программы — не только игры, но и любого приложения. Исполняемый файл в несколько этапов создается из *исходного кода*, написанного на C++. Исходный код — это набор инструкций, написанных на C++. Этот процесс проиллюстрирован на рис. 1.1.

1. Сначала программист открывает *редактор* и пишет в нем исходный код на языке C++. Получается файл, обычно имеющий расширение `.cpp`. Такой инструмент напоминает обычный текстовый редактор. В этой программе программист может писать, редактировать и сохранять исходный код.
2. Сохранив файл с исходным кодом, программист запускает *компилятор* C++. Это программа, считывающая исходный код и преобразующая его в *объектный файл*. Обычно объектные файлы имеют расширение `.obj`.
3. Далее компоновщик, также именуемый *редактором связей*, связывает объектный файл с нужными внешними файлами и таким образом создает *исполняемый файл*, обычно имеющий расширение `.exe`. Теперь пользователь (геймер) может запустить программу, просто открыв исполняемый файл.

СОВЕТ

Здесь я описал процесс в упрощенном виде. Чтобы создать сложное приложение на C++, требуется собрать в одну программу множество файлов исходного кода, написанных программистом (нередко целой командой программистов).

Рис. 1.1. Создание исполняемого файла на основе исходного кода C++

Чтобы автоматизировать этот процесс, программисты обычно используют универсальный инструмент, называемый IDE *(интегрированная среда разработки)*. Как правило, в составе IDE есть редактор, компилятор, компоновщик, а также другие инструменты. Существует популярная (при этом свободно распространяемая) IDE для операционной системы Windows. Она называется Visual Studio Express 2013 для Windows ПК. Подробнее почитать об этой IDE, а также скачать ее можно по адресу www.visualstudio.com/downloads/download-visual-studio-vs.

Исправление ошибок

Рассказывая о создании исполняемого файла из исходного кода C++, я не упомянул об одной детали — об ошибках. Человеку свойственно ошибаться, а программисту — особенно. Даже самые лучшие программисты пишут такой код, в котором и после первой, и после пятой проверки обнаруживаются какие-нибудь ошибки. Программисту приходится исправлять ошибки и все переделывать. Далее перечислены наиболее распространенные типы ошибок, встречающиеся в C++.

- **Ошибки компиляции.** Они возникают на этапе компиляции кода. Поэтому объектный файл не получается. Это могут быть *синтаксические ошибки* — значит, компилятор чего-то «не понял». Зачастую они возникают из-за банальных

опечаток. Компилятор может выдавать предупреждения. Хотя обычно такие предупреждения погоды не делают, то, о чем в них говорится, следует расценивать как ошибки — исправлять и перекомпилировать.

- **Ошибка компоновки.** Такие ошибки возникают в процессе компоновки (связывания) и могут означать, что программа не может найти какие-то данные, ссылки на которые в ней имеются. Обычно для устранения таких ошибок достаточно исправить проблемную ссылку и повторить процесс компиляции/компоновки.
- **Ошибки времени исполнения.** Они возникают при запуске исполняемого файла. Если программа выполнит недопустимую операцию, она может внезапно завершиться. Но существуют и более тонкие ошибки времени исполнения — так называемые *логические ошибки*, в результате которых программа просто начинает действовать непредусмотренным образом. Если вам когда-нибудь доводилось играть в игру, персонаж которой вдруг начинает ходить в воздухе (но по сценарию этого делать не может), то вам встретилась типичная логическая программная ошибка.

НА ПРАКТИКЕ

Как и все разработчики софта, игровые компании напряженно работают, стремясь создавать продукты, начисто лишенные таких ошибок (багов). Последняя линия защиты от багов — это сотрудники отделов обеспечения качества, в просторечии называемые тестировщиками. Тестировщики-геймеры зарабатывают на жизнь, играя в игры, но эта работа далеко не такая классная, как может показаться на первый взгляд. Тестировщик должен проходить одни и те же этапы игры снова и снова, иногда сотни раз подряд, пробуя все возможные и невозможные варианты и тщательно описывая все замеченные аномалии. При такой монотонной работе зарплаты тестировщиков отнюдь не высоки. Однако получить такую работу — замечательный способ закрепиться в штате игровой компании.

Понятие о стандарте ISO

Стандарт ISO (International Organization for Standardization — *Международная организация по стандартизации*) для языка C++ — это определение C++, в точности описывающее, как должен работать этот язык. Кроме того, в данном стандарте ISO описана группа файлов, называемая *стандартной библиотекой*. Стандартная библиотека содержит файлы-кирпичики, позволяющие программисту решать распространенные задачи, например получать ввод и отображать вывод. Стандартная библиотека серьезно облегчает жизнь программисту и предоставляет ему фундаментальный код, избавляя от необходимости постоянно изобретать велосипед. Стандартная библиотека применялась при написании всех программ из этой книги.

СОВЕТ

Стандарт ISO часто называют ANSI (American National Standards Institute — Американский национальный институт стандартов) или ANSI/ISO. Эти аббревиатуры соответствуют разным организациям и комитетам, рассматривавшим и формировавшим этот стандарт. Простейшее обозначение кода C++, соответствующего всем требованиям стандарта ISO, — стандартный C++.

Разрабатывая программы для этой книги, я пользовался интегрированной средой Microsoft Visual Studio Express 2013 для Windows ПК. Компилятор, входящий в состав этой интегрированной среды разработки, безупречно соответствует стандарту ISO, поэтому вы без проблем сможете компилировать, компоновать и запускать любые программы, работая с любым другим современным компилятором. Правда, если вы работаете в Windows, рекомендую пользоваться именно Microsoft Visual Studio Express 2013 для Windows ПК.

СОВЕТ

Пошаговые инструкции о том, как написать, сохранить, скомпилировать и запустить программу Game Over с помощью Microsoft Visual Studio Express 2013 для Windows ПК, вы найдете в приложении 1 к этой книге, которое называется «Создание первой программы на языке C++». Если вы пользуетесь другой средой разработки или другим компилятором, почитайте документацию по этим инструментам.

Пишем первую программу на C++

Итак, довольно теории. Переходим к самому интересному: сейчас вы напишете свою первую программу на языке C++. Хотя эта программа и очень проста, она обладает всеми характерными чертами типичной программы. Кроме того, в ней будет продемонстрировано, как выводить текст в окне консоли.

Знакомство с программой Game Over

Как правило, начиная осваивать новый язык, разработчик пишет на нем программу «Hello World». Она отображает на экране слова «Hello World», которые можно перевести как «Привет, мир». Программа Game Over является игровой вариацией этой классической задачи и выводит на экран слова «Game Over!» («Игра окончена!»). На рис. 1.2 эта программа показана в действии.

Рис. 1.2. Ваша первая программа выводит на экран два слова, способные огорчить любого геймера (используется с разрешения компании Microsoft)

Вы можете скачать исходный код этой программы на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 1 (Chapter 1), имя файла — game_over.cpp.

СОВЕТ

Чтобы быстро найти этот код на ресурсе www.cengageptr.com/downloads, можете сразу поискать эту книгу. Например, ее удобно искать по идентификационному номеру (ISBN). ISBN этой книги — 9781305109919.

```
// GameOver
// Первая программа на C++
#include <iostream>
int main()
{
    std::cout << "Game Over!" <<std::endl;
    return 0;
}
```

Комментирование кода

Первые две строки в данной программе — это комментарии:

```
// GameOver
// Первая программа на C++
```

Строки комментариев компилятор игнорирует, поскольку предназначены они для людей, а не для машины. Комментарии помогают другим программистам, которые будут читать ваш код, понять ход ваших мыслей. Но комментарии могут пригодиться и вам самим. Они напоминают, как вы решили ту или иную задачу, тогда как из кода это может быть неочевидно.

Чтобы создать комментарий, достаточно начать строку с двух прямых слешей (//). Весь остальной текст до конца физической строки машина сочтет частью комментария. Таким образом, на одной и той же строке сначала может идти код C++, а затем — комментарий, но после кода до начала комментария должны стоять два прямых слеша.

СОВЕТ

Существует еще одна разновидность комментариев. Они называются «комментарии в стиле C» и могут занимать несколько строк. Такой комментарий необходимо начинать с символов /* и заканчивать символами */. Весь текст между двумя этими маркерами будет считаться частью комментария.

Использование пробелов

Сразу после комментариев в нашей программе идет пустая строка. Компилятор игнорирует пустые строки. На самом деле компилятор игнорирует любое пустое пространство — пробелы, табуляцию и символы перехода на новую строку. Как и комментарии, пробелы предназначены не для машины, а для человека.

При разумном использовании пробелов повышается удобочитаемость программы. Например, можно оставлять пустые строки между большими блоками кода, если весь код в конкретном блоке взаимосвязан. Я также использую пробелы (точнее, табуляцию) в начале каждой из двух строк между фигурными скобками, чтобы немного развести эти скобки.

Включение других файлов

Следующая строка программы — это директива препроцессора. Строки с такими директивами начинаются с символа #:

```
#include<iostream>
```

Препроцессор начинает действовать еще до того, как к работе подключится компилятор. Он выполняет подстановку текста в зависимости от различных директив. В данном случае строка начинается с директивы `#include`, приказывающей препроцессору включить в программу содержимое другого файла.

Здесь я включаю файл `iostream`, входящий в состав стандартной библиотеки, поскольку в этом файле содержится код, упрощающий отображение вывода. Я заключаю имя файла в угловые скобки — это сигнал компилятору искать этот файл среди тех, что поставляются вместе с компилятором. Файл, включаемый в ваши программы таким образом, называется *заголовочным*.

Определение функции main()

Следующая непустая строка — это заголовок функции `main()`:

```
intmain()
```

Функция — это единица в составе программного кода. Функция должна выполнить определенную работу и вернуть значение. В данном случае код `int` означает, что функция должна вернуть целочисленное значение. Во всех заголовках функций после имени функции ставятся две круглые скобки.

Во всех программах на языке C++ должна быть функция `main()` — это своеобразная исходная точка программы. Здесь начинается действие.

Следующая строка отмечает начало функции:

```
{
```

А самая последняя строка — конец функции:

```
}
```

Все функции начинаются и заканчиваются фигурной скобкой, а весь код между ними относится к функции. Код между двумя фигурными скобками называется блоком, обычно он снабжается отступами, чтобы было понятнее, что этот блок — цельная единица. Блок кода, образующий целую функцию, называется телом функции.

Отображение текста в стандартном выводе

Первая строка в теле функции `main()` содержит слова `GameOver!`, затем идет новая строка. Все это мы видим в окне консоли:

```
std::cout<< "GameOver!" <<std::endl;
```

Слова `GameOver!` — это *строка* или *последовательность символов*, точнее, последовательность символов, которые отображаются при печати. Технически данная последовательность является *строковым литералом* — то есть перед нами в буквальном смысле находятся именно эти символы в кавычках.

`cout` — это объект, определенный в файле `iostream` и используемый для отправки данных в поток стандартного вывода. В большинстве программ (включая эту) поток стандартного вывода — это просто окно консоли на экране компьютера.

Я применяю *оператор вывода* (`<<`) для отправки строки в `cout`. Оператор вывода можно сравнить с воронкой: он берет все, что наливают в горлышко, и перенаправляет эту информацию в указанное место. Эта строка направляется в стандартный вывод, то есть в окно консоли на экране.

Перед объектом `cout` я ставлю префикс `std`, чтобы подсказать компилятору, что имею в виду объект `cout` из стандартной библиотеки. `std` — это *пространство имен*. Пространство имен можно сравнить с региональным телефонным кодом. Оно обозначает группу, к которой относится тот или иной объект. Пространство имен `std` отделяется от собственно имени `cout` оператором разрешения видимости. Перед пространством имен ставится *оператор разрешения* (`::`).

Наконец, я отправляю в стандартный вывод код `std::endl`. `endl` также является объектом из пространства имен `std` и определяется в файле `iostream`. Операция отправки `endl` в стандартный вывод аналогична тому, как если бы мы нажали клавишу **Enter** в окне консоли. Кроме того, если бы я отправил в окно консоли еще одну последовательность символов, то она началась бы с новой строки.

Не исключаю, что запомнить весь этот материал сразу сложно. Поэтому изучите рис. 1.3, где наглядно представлены взаимосвязи всех описанных элементов.

Рис. 1.3. В стандартной реализации языка C++ имеется набор файлов, называемый «стандартная библиотека». В стандартной библиотеке есть файл iostream, в котором определяются различные сущности, в том числе объект cout

Завершение инструкций

Вероятно, вы заметили, что первая строка функции заканчивается точкой с запятой (;). Дело в том, что эта строка является *инструкцией* — основной единицей, отвечающей за управление исполнением кода. Все инструкции должны завершаться точками с запятой, иначе компилятор будет генерировать сообщения об ошибках, а ваша программа компилироваться не станет.

Возвращение значения от функции main()

Последняя инструкция в функции возвращает операционной системе значение 0:

```
return 0;
```

Возврат 0 от функции `main()` — это способ сообщить, что работа программы завершилась в штатном режиме. Операционной системе ничего не требуется делать с возвращаемым значением. В принципе, можно просто вернуть 0, как сделано здесь.

ПРИЕМ

Запустив программу Game Over, вы едва можете заметить, как на экране открывается и сразу же исчезает окно консоли. Все дело в том, как стремительно работает язык C++: на открытие окна консоли, отображение фразы «Game Over!» и закрытие окна у программы уходят считаные доли секунды. Правда, в операционной системе Windows можно создать пакетный файл, запускающий консольную программу и приостанавливающийся. При этом окно консоли остается открытым и вы можете просмотреть результат выполнения вашей программы. Поскольку скомпилированная программа называется game_over.exe, можно создать пакетный файл всего из двух строк:

```
game_over.exe
pause
```

Для создания пакетного файла сделайте следующее.

1. Откройте текстовый редактор, например Notepad (Word или WordPad не подходят).
2. Введите текст.
3. Сохраните файл в том же каталоге, что и `game_over.exe`. Этот файл должен иметь расширение `.bat`, то есть вполне подойдет имя `game_over.bat`.

Наконец, запустите пакетный файл, дважды щелкнув на его ярлыке. Вы должны увидеть результаты выполнения программы, поскольку пакетный файл оставляет окно консоли открытым.

Работа с пространством имен std

Поскольку элементы из пространства имен `std` используются довольно часто, я покажу два различающихся метода непосредственного доступа к этим элементам. Они позволяют во многих случаях обходиться без префикса `std::`.

Знакомство с программой Game Over 2.0

Программа Game Over 2.0 дает точно такие же результаты, что и программа Game Over (см. рис. 1.2). Но в случае использования Game Over 2.0 мы будем иным способом обращаться к элементам из пространства имен std. Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 1, имя файла — game_over2.cpp.

```
// Программа Game Over 2.0
// Демонстрирует работу с директивой using
#include <iostream>
using namespace std;
int main()
{
    cout << "Game Over!" << endl;
    return 0;
}
```

Использование директивы using

Программа запускается так же, как и в первой версии. Я пишу два вводных комментария, а затем включаю файл iostream для стандартного вывода. Но далее идет инструкция нового типа:

```
using namespace std;
```

Это директива using, дающая непосредственный доступ к элементам из пространства имен std. Повторю: пространство имен напоминает региональный телефонный код. Соответственно, эта строка означает, что теперь все элементы из пространства имен std в моей программе похожи на телефонные номера из одного города. Таким образом, я могу пользоваться ими, не набирая международного кода (префикса std::).

Я могу использовать объекты cout и endl без какого-либо префикса. Возможно, пока этот факт вас не впечатляет, но когда у вас будут сотни или даже тысячи ссылок на эти объекты, вы скажете мне спасибо.

Знакомство с программой Game Over 3.0

Есть и другой способ решения той задачи, которую мы решали в примере с Game Over 2.0. Нужно написать файл так, чтобы можно было обращаться к объектам cout и endl и без префикса std::. Именно это мы и сделаем в программе Game Over 3.0, которая выводит на экран такой же текст, как и предыдущие версии. Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 1, имя файла — game_over3.cpp.

```
// Программа Game Over 3.0
// Демонстрирует работу с объявлениями using
#include <iostream>
using std::cout;
```

```
using std::endl;
int main()
{
    cout << "Game Over!" << endl;
    return 0;
}
```

Использование объявлений using

В этой версии напишем два объявления using:

```
using std::cout;
using std::endl;
```

Непосредственно объявляя, какие элементы из пространства имен std я собираюсь локально использовать в этой программе, я смогу обращаться к ним напрямую, так же как в игре Game Over 2.0. Хотя в данном случае приходится набирать больше текста, чем при работе с директивой using, преимущество такого варианта заключается в следующем: сразу точно известно, с какими элементами предполагается работать. Кроме того, в таком случае не становятся локальными всевозможные другие элементы, которые нам не нужны.

Когда следует использовать using

Итак, мы рассмотрели два способа, позволяющих сделать элементы из пространства имен локальными для нашей программы. Но какой вариант лучше?

Строгий пурист сказал бы, что не следует применять ни одного варианта с using, а всегда сопровождать префиксом-идентификатором все элементы из пространства имен. Мне кажется, что такой подход выглядит нелепо, как если бы вы постоянно называли лучшего друга по имени и фамилии. Налицо лишний официоз.

Если вы ни за что не хотите набирать лишний код, смело используйте директиву using. Разумный компромисс — пользоваться объявлениями using. В этой книге для краткости я в большинстве случаев применяю директиву using.

НА ПРАКТИКЕ

Ранее я изложил несколько вариантов работы с пространствами имен. А также постарался объяснить достоинства каждого из способов, чтобы вы могли сами обдумать, какую стратегию избрать в своих программах. Но ведь такое решение можете принимать и не вы. Если вы работаете на проекте (учебном или боевом), то, скорее всего, должны придерживаться стандартов написания кода, подготовленных начальником. Независимо от личных предпочтений всегда лучше слушаться тех, кто ставит оценки или платит.

Работа с арифметическими операторами

Когда вы подсчитываете количество истребленных врагов или снижаете уровень здоровья у персонажа, ваша программа занимается математикой. Как и в других языках, в C++ есть встроенные арифметические операторы.

Знакомство с программой Expensive Calculator

Большинство бывалых компьютерных геймеров щедро тратятся на высококлассную мощную игровую экипировку. Следующая программа, Expensive Calculator, может превратить эту машинерию в простой калькулятор. Эта программа демонстрирует работу с арифметическими операторами, результаты показаны на рис. 1.4.

Рис. 1.4. Язык C++ позволяет складывать, вычитать, умножать, делить и даже вычислять остаток (опубликовано с разрешения компании Microsoft)

Код программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 1, имя файла — `expensive_calculator.cpp`.

```
// Дорогой калькулятор
// Демонстрирует работу со встроенными арифметическими операторами
#include <iostream>
using namespace std;
int main()
{
    cout << "7 + 3 = " << 7 + 3 << endl;
    cout << "7 - 3 = " << 7 - 3 << endl;
    cout << "7 * 3 = " << 7 * 3 << endl;
    cout << "7 / 3 = " << 7 / 3 << endl;
    cout << "7.0 / 3.0 = " << 7.0 / 3.0 << endl;
    cout << "7 % 3 = " << 7 % 3 << endl;
    cout << "7 + 3 * 5 = " << 7 + 3 * 5 << endl;
    cout << "(7 + 3) * 5 = " << (7 + 3) * 5 << endl;
    return 0;
}
```

Сложение, вычитание, умножение

Я использую встроенные арифметические операторы для сложения (+), вычитания (-) и умножения (*). На рис. 1.4 представлены вполне ожидаемые результаты.

Каждый арифметический оператор входит в состав *выражения*. При вычислении выражения получается конкретное значение. Например, выражение 7 + 3 дает 10, именно это значение отсылается в cout.

Понятие о делении без остатка и делении с плавающей запятой

Символ деления — прямой слеш. Но вывод в окне может вас удивить. По правилам C++ (и вышеупомянутой игровой экипировки) 7, деленное на 3, равно 2. Что такое? Все дело в том, что, когда делимое и делитель являются *целыми числами* (не содержащими дробных частей), частное также должно быть целым числом. Поскольку и 7 и 3 — целые числа, в результате тоже имеем целое число. Дробная часть результата отбрасывается.

Чтобы получить дробный результат, как минимум одно из исходных значений должно быть *числом с плавающей запятой* (то есть с дробной частью). Этот случай будет продемонстрирован на примере выражения 7,0 / 3,0. На этот раз результат более точный — 2,33333..

ОСТОРОЖНО!

Вероятно, вы обратили внимание на то, что частное в выражении 7,0 / 3,0 (2,33333) действительно включает дробную составляющую, однако большая часть знаков после запятой отбрасывается (на самом деле здесь должно быть бесконечное количество троек). Важно учитывать, что компьютеры, как правило, хранят для чисел с плавающей запятой ограниченное количество значимых разрядов. Но в C++ используются такие дробные числа, которые отлично подходят даже для самых сложных трехмерных игр, требующих невероятно интенсивных вычислений.

Работа с оператором деления по модулю

В следующей инструкции используется оператор, который, возможно, вам пока не знаком. Это оператор деления по модулю (%). Оператор деления по модулю возвращает результат остатка от целочисленного деления. В данном случае операция 7 % 3 дает результат от деления 7 / 3, то есть 1.

Понятие о последовательности операций

Как и в алгебре, в C++ арифметические выражения вычисляются слева направо. Но одни действия имеют приоритет над другими и выполняются в первую очередь, несмотря на расположение. Умножение, деление и деление по модулю имеют одинаковый приоритет, на один уровень ниже по приоритету стоят операции сложения и вычитания.

В следующей строке кода приводится пример, помогающий усвоить эти правила. Поскольку умножение приоритетнее сложения, сначала вычисляется результат умножения. Итак, выражение 7 + 3 * 5 эквивалентно 7 + 15, что равно 22.

Если вы хотите, чтобы сначала была выполнена операция с более низким приоритетом, нужно заключить это действие в круглые скобки, которые имеют более

высокий приоритет, чем любой арифметический оператор. Поэтому следующая инструкция, выражение `(7 + 3) * 5`, эквивалентна $10 \cdot 5$, что равно 50.

СОВЕТ

Список операторов языка C++ с указанием приоритета каждого из них приведен в приложении 2 «Приоритет операторов языка C++».

Объявление и инициализация переменных

Переменная представляет собой фрагмент памяти вашего компьютера, выделенный специально для того, чтобы вы могли хранить в переменной данные, извлекать их оттуда и манипулировать ими. Так, если вы хотите вести счет очков пользователя, то можете создать для этого специальную переменную, а потом извлекать из нее текущее количество очков и отображать на экране. Кроме того, можно обновлять счет, когда персонаж сбивает летающего пришельца.

Знакомство с программой Game Stats

Программа Game Stats выводит на экран информацию, которую, вероятно, потребуется отслеживать в игре-шутере на космическую тематику. К такой информации относятся очки пользователя, количество уничтоженных врагов, а также количество слоев брони, которыми (возможно) обладает пользователь. Для решения всех этих задач в программе используется набор переменных. Программа показана на рис. 1.5.

Рис. 1.5. Все данные игровой статистики хранятся в специальных переменных (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 1, имя файла — `game_stats.cpp`.

```
// Программа Game Stats
// Демонстрирует объявление и инициализацию переменных
#include <iostream>
```

```
using namespace std;
int main()
{
    int score;
    double distance;
    char playAgain;
    bool shieldsUp;
    short lives, aliensKilled;
    score = 0;
    distance = 1200.76;
    playAgain = 'y';
    shieldsUp = true;
    lives = 3;
    aliensKilled = 10;
    double engineTemp = 6572.89;
    cout << "\nscore: " << score << endl;
    cout << "distance: " << distance << endl;
    cout << "playAgain: " << playAgain << endl;
    // пропускаем shieldsUp, поскольку булевы значения,
    // как правило, на экран не выводятся
    cout << "lives: " << lives << endl;
    cout << "aliensKilled: "<< aliensKilled << endl;
    cout << "engineTemp: " << engineTemp << endl;
    int fuel;
    cout << "\nHow much fuel? ";
    cin >> fuel;
    cout << "fuel: " << fuel << endl;
    typedef unsigned short int ushort;
    ushort bonus = 10;
    cout << "\nbonus: " << bonus << endl;
    return 0;
}
```

Понятие о примитивах

Каждая создаваемая вами переменная имеет *тип*, то есть тип информации, которую можно хранить в этой переменной. По типу компилятор узнает, сколько памяти выделено под переменную, тип строго определяет, какие операции можно выполнять с данной переменной.

Примитивы C++, то есть типы, встроенные в язык, — это `bool` для булевых значений (`true` или `false`), `char` для односимвольных значений, `int` для целых чисел, `float` для чисел одинарной точности с плавающей запятой, `double` для чисел двойной точности с плавающей запятой.

Понятие о модификаторах типов

Модификаторы позволяют изменять типы. Модификатор `short` сокращает общее количество значений, которые могут содержаться в переменной. Модификатор `long` увеличивает общее количество значений, которые могут содержаться

в переменной. short уменьшает пространство, отводимое под переменную в памяти компьютера, а long — увеличивает. Модификаторы short и long применимы к int, а long — также к double.

Модификаторы signed и unsigned работают только с целочисленными типами. Модификатор signed означает, что переменная может хранить как положительные, так и отрицательные значения, а unsigned — только положительные. Ни signed, ни unsigned не меняют общего количества значений, которые могут содержаться в переменной, они изменяют только диапазон значений. По умолчанию с целочисленными типами применяется модификатор signed.

Что, уже запутались с типами? В табл. 1.1 собраны наиболее употребительные типы, а также некоторые модификаторы к ним. Кроме того, здесь вы найдете диапазон значений к каждому из типов.

Таблица 1.1. Наиболее распространенные типы

Тип	Значения
short int	От –32 768 до 32 767
unsigned short int	От 0 до 65 535
int	От –2 147 483 648 до 2 147 483 647
unsigned int	От 0 до 4 294 967 295
long int	От –2 147 483 648 до 2 147 483 647
unsigned long int	От 0 до 4 294 967 295
float	3.4E ± 38 (семь цифр после запятой)
double	1,7E ± 308 (15 цифр после запятой)
long double	1,7E ± 308 (15 цифр после запятой)
char	256 символьных значений
bool	true или false

ОСТОРОЖНО!

Диапазон значений, приведенных в этой таблице, продиктован моим компилятором. Возможно, вы работаете с другим компилятором, так что сверьтесь с документацией по нему.

СОВЕТ

Для краткости тип short int можно записывать просто как short, а long int — как long.

Объявление переменных

Вот мы и познакомились с типами. Пора вернуться к программе. Первым делом мне требуется *объявить* переменную (сделать запрос на ее создание), вот так:

```
int score;
```

Здесь я объявляю переменную типа int и называю ее score. Имя переменной используется для доступа к ней. Как видите, при объявлении переменной сначала

указывается выбранное вами имя, а затем — ее тип. Поскольку объявление — это инструкция, оно должно заканчиваться точкой с запятой.

В следующих трех строках я объявляю еще три переменные трех новых типов. Переменная `distance` относится к типу `double`, `playAgain` — к типу `char`, а `shieldsUp` — к типу `bool`.

В играх (как и в большинстве приложений) используется множество типов переменных. К счастью, язык C++ позволяет объявить сразу несколько типов переменных в рамках одной инструкции. Именно это мы сделаем в следующей строке:

```
short lives, aliensKilled;
```

В этой строке создаются две переменные типа `short` — `lives` и `aliensKilled`.

Хотя я и определил группу переменных в верхней части функции `main()`, вы не обязаны объявлять все переменные в одном месте. Как будет показано далее в нашей программе, в некоторых случаях я объявляю переменную непосредственно перед тем, как ее использовать.

Именование переменных

Чтобы объявить переменную, необходимо присвоить ей имя, которое также называется *идентификатором*. Чтобы создать допустимый идентификатор, нужно придерживаться всего нескольких простых правил.

- Идентификатор может содержать только цифры, буквы и нижние подчеркивания.
- Идентификатор не может начинаться с цифры.
- В качестве идентификаторов нельзя использовать ключевые слова C++.

Ключевое слово — это особое слово, зарезервированное в C++ для внутриязыкового использования. Этих слов не так уж много, их полный список приведен в приложении 3, которое называется «Ключевые слова языка C++».

Наряду с правилами создания *допустимых* имен переменных существуют и следующие рекомендации, как подбирать для переменных *хорошие* имена.

- Имена переменных должны быть информативными. Они должны быть понятны другому программисту. Например, используйте имя `score`, а не `s`. Исключение из этого правила составляют переменные, используемые в течение краткого периода. В таких случаях вполне уместны однобуквенные имена переменных, например `x`.
- Будьте последовательны. Существуют различные традиции написания имен переменных, состоящих из нескольких слов. Как лучше, `high_score` или `highScore`? В этой книге я придерживаюсь второго стиля, где первая буква второго слова (а также третьего и всех последующих) делается заглавной. Такой стиль еще называется *«верблюжий регистр»*. Но если вы последовательны в именовании переменных, то конкретный стиль не так уж важен.
- Придерживайтесь традиций языка. Некоторые соглашения об именовании просто прижились и стали традиционными. Например, в большинстве языков, и в C++ в том числе, имена переменных принято начинать со строчной буквы. Другая традиция — стараться не начинать имени переменной с нижнего подчеркивания.

Имена, начинающиеся с нижнего подчеркивания, могут иметь специальное применение.

- Следите за длиной имени. Хотя имя `playerTwoBonusForRoundOne` исчерпывающе характеризует переменную, читать такие имена порой сложно. Кроме того, возрастает риск при наборе такого длинного имени сделать опечатку. Старайтесь не давать переменным имен длиннее 15 символов. Правда, истинный предел длины для имен переменных определяется компилятором.

СОВЕТ

Код называют самодокументируемым, если программист пишет его таким образом, что логика программы понятна из кода без всяких комментариев. Грамотный подбор имен для переменных — серьезный шаг к созданию именно такого кода.

Присваивание значений переменным

В следующей группе инструкций я присваиваю значения шести переменным, которые уже объявил. Я разберу здесь несколько операций присваивания и немного расскажу обо всех типах переменных.

Присваивание значений целочисленным переменным

В следующей инструкции присваивания я задаю значение `0` для переменной `score`:

```
score = 0;
```

Теперь в переменной `score` хранится значение `0`.

Чтобы присвоить переменной значение, нужно записать имя этой переменной, далее поставить оператор присваивания (`=`), а затем — выражение. Строго говоря, `0` — это тоже выражение, оно дает в результате `0`.

Присваивание значений переменным с плавающей запятой

В следующей инструкции я присваиваю переменной `distance` значение `1200.76`:

```
distance = 1200.76;
```

Поскольку переменная `distance` относится к типу `double`, я могу хранить в ней число с дробной частью — что и делаю.

Присваивание значений символьным переменным

В следующей инструкции я присваиваю переменной `playAgain` односимвольное значение `y`:

```
playAgain = 'y';
```

Так же, как я сделал здесь, вы можете присваивать символьное значение переменной типа `char`, заключив этот символ (значение) в одиночные кавычки.

Переменные типа `char` могут хранить 128 символьных значений в кодировке ASCII (предполагается, что в вашей системе используется эта кодировка). Аббревиатура ASCII расшифровывается как *американский стандартный код для обмена информацией*. Это код для представления символов. Полный список символов ASCII дается в приложении 4 «Таблица символов ASCII».

Присваивание значений булевым переменным

В следующей инструкции я присваиваю переменной shieldsUp значение true:

```
shieldsUp = true;
```

В моей игре это означает, что у персонажа еще есть броня.

Переменная shieldsUp является булевой (относится к типу bool). Она может принимать одно из двух значений: true или false. Разумеется, такие переменные очень интересны, но подробнее познакомиться с ними вам предстоит только в главе 2.

Инициализация переменных

Можно одновременно и объявить переменную, и присвоить ей значение. Для этого используются инструкции инициализации. Именно это я и делаю далее:

```
double engineTemp = 6572.89;
```

В этой строке создается переменная engineTemp типа double, в которой хранится значение 6572,89.

Как вы помните, можно объявить в одной инструкции несколько переменных. Аналогично в одной инструкции можно инициализировать несколько переменных. В одной инструкции можно даже одновременно и объявлять, и инициализировать разные переменные. Комбинируйте, как хотите!

СОВЕТ

Хотя и можно объявить переменную, не присваивая ей значения, по возможности старайтесь инициализировать новую переменную с начальным значением. Во-первых, так получается более понятный код, а во-вторых исключается возможность доступа к неинициализированной переменной, которая может содержать любое значение.

Отображение значений переменных

Чтобы отобразить значение переменной, относящейся к любому из примитивов, просто отправьте ее в cout. Именно это я и делаю в программе. Обратите внимание: я не пытаюсь отобразить значение shieldsUp, поскольку значения типа bool на экран обычно не выводятся.

ПРИЕМ

В первой инструкции этого раздела применяется так называемая управляющая последовательность (escape sequence) — пара символов, которая начинается с обратного слеша (\) и представляет особые печатные символы:

```
cout<< "\nscore: " <<score<<endl;
```

Здесь я использую управляющую последовательность \n, которая соответствует новой строке. Отправляя эту информацию в cout в составе символьной последовательности, мы как будто нажимаем в окне консоли клавишу Enter. Еще одна полезная управляющая последовательность, \t, действует как клавиша табуляции.

Существуют и другие управляющие последовательности. Их список приведен в приложении 5 «Управляющие последовательности».

Получение пользовательского ввода

Мы также можем присвоить переменной значение с помощью пользовательского ввода. Далее я присваиваю значение новой переменной `fuel`, это значение зависит от пользовательского ввода. Для этого используется следующая строка кода:

```
cin>>fuel;
```

Точно так же, как и `cout`, `cin` — это объект из файла `iostream`, относящийся к пространству имен `std`. Чтобы сохранить значение в переменной, я ставлю после объекта `cin` символы >> (оператор извлечения), за которым следует имя переменной. Можно применять объект `cin` и оператор извлечения и для записи пользовательского ввода в переменные других примитивов. Чтобы убедиться, что все работает, я отображаю для пользователя переменную `fuel`.

Определение новых имен для типов

Можно создать новое имя для уже существующего типа. Именно это я и делаю в следующей строке:

```
typedef unsigned short int ushort;
```

Этот код определяет идентификатор `ushort` как альтернативное имя для типа `unsignedshortint`. Для определения новых имен существующих типов используется ключевое слово `typedef`, за которым идет сначала известный тип, а затем — новое имя для этого типа. Ключевое слово `typedef` часто применяется при создании более коротких имен для типов с длинными именами.

Новое имя типа можно использовать точно так же, как и исходное имя этого типа. Так, я инициализирую переменную типа `ushort` (на самом деле речь идет о типе `unsignedshortint`) с именем `bonus` и отображаю ее значение.

Типы, которые следует использовать

Программист работает с множеством примитивов, из которых может выбрать нужный. Как вы узнаете, какой тип использовать? Конечно, если требуется целочисленный тип, то лучше всего прибегнуть к `int`. Дело в самой реализации `int`: обычно переменные этого типа занимают в памяти как раз такие фрагменты, которые обрабатываются компьютером наиболее эффективно. Если требуется представлять такие целочисленные значения, которые превышают максимальный предел `int`, либо значения, которые по определению не могут быть отрицательными, — смело используйте `unsignedint`.

Если памяти в обрез, то можно работать с типом, расходующим память более экономно. Правда, на большинстве компьютеров проблем с памятью возникать не должно. Конечно, бывает еще программирование для игровых приставок и мобильных устройств, но это совсем другая история.

Наконец, если вам требуется число с плавающей запятой, то лучше всего работать с типом `float`. Он реализуется так, что занимаемые им в памяти компьютера фрагменты удобны при обработке.

Выполнение арифметических операций с применением переменных

Определив переменные и задав для них значения, вы планируете изменять эти значения по ходу игры. Например, можно присудить персонажу бонус за победу над боссом — добавить очков. Либо понизить уровень кислорода в гермошлюзе. Пользуясь операторами, рассмотренными ранее (а также некоторыми другими), можно решить любые подобные задачи.

Знакомство с программой Game Stats 2.0

Программа Game Stats 2.0 оперирует переменными, представляющими игровую статистику, и выводит результаты на экран. На рис. 1.6 показано, как работает эта программа.

Рис. 1.6. Каждая переменная изменяется по-своему (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 1, имя файла — game_stats2.cpp.

```
// Программа Game Stats 2.0
// Демонстрирует арифметические операции с переменными
#include <iostream>
using namespace std;
int main()
{
    unsigned int score = 5000;
    cout << "score: " << score << endl;
    // изменение значения переменной
    score = score + 100;
    cout << "score: " << score << endl;
    // комбинированный оператор присваивания
    score += 100;
```

```
    cout << "score: " << score << endl;
    // операторы инкремента
    int lives = 3;
    ++lives;
    cout << "lives: " << lives << endl;
    lives = 3;
    lives++;
    cout << "lives: " << lives << endl;
    lives = 3;
    int bonus = ++lives * 10;
    cout << "lives, bonus = " << lives << ", " << bonus << endl;
    lives = 3;
    bonus = lives++ * 10;
    cout << "lives, bonus = " << lives << ", " << bonus << endl;
    // целочисленное переполнение
    score = 4294967295;
    cout << "\nscore: " << score << endl;
    ++score;
    cout << "score: " << score << endl;
    return 0;
}
```

ОСТОРОЖНО!

При компилировании этой программы вы можете получить предупреждение примерно такого характера: [Warning] this decimal constant is unsigned ([Внимание] эта десятичная константа является беззнаковой). К счастью, такое предупреждение не прерывает компилирования и выполнения программы. Подобные предупреждения возникают в результате так называемого целочисленного переполнения (integer wrap), которого желательно избегать в программе. Однако в этой программе переполнение сделано намеренно, чтобы показать результат события. Мы подробнее обсудим целочисленное переполнение в разделе «Что делать с целочисленным переполнением».

Изменение значения переменной

Итак, у меня есть переменная для хранения пользовательских баллов и их вывода на экран. Я изменяю счет, добавляя пользователю 100 баллов:

```
score = score + 100;
```

Эта инструкция присваивания означает: «Возьми текущее значение score, прибавь к нему 100, а результат присвой score как новое значение». Фактически эта строка увеличивает значение score на 100.

Использование комбинированных операторов присваивания

Предыдущую строку можно переписать еще короче, вот так:

```
score += 100;
```

Эта инструкция дает такой же результат, как и `score = score+100`. Оператор += называется *комбинированным оператором присваивания*, так как он одновременно выполняет и арифметическую операцию (в данном случае сложение), и присваивание. Можно сказать, что этот оператор кратко формулирует такую команду: «Сложи то, что находится справа от оператора, с тем, что находится слева от оператора, полученный результат присвой тому, что находится слева от оператора».

Для всех арифметических операторов есть свои разновидности комбинированных операторов. Их список приведен в табл. 1.2.

Табл. 1.2. Комбинированные операторы присваивания

Оператор	Пример	Эквивалентно
+=	x += 5;	x = x + 5;
-=	x -= 5;	x = x - 5;
*=	x *= 5;	x = x * 5;
/=	x /= 5;	x = x / 5;
%=	x %= 5;	x = x % 5;

Использование операторов инкремента и декремента

Далее я использую *оператор инкремента* (++), увеличивающий значение переменной на единицу. С помощью этого оператора я дважды увеличиваю значение переменной `lives`. Сначала использую этот оператор в следующей строке:

```
++lives;
```

А затем — в следующей:

```
lives++;
```

Обе эти строки дают один и тот же эффект — увеличивают значение `lives` с 3 до 4.

Как видите, оператор можно поставить до или после переменной, которую вы увеличиваете на единицу. Если оператор ставится перед переменной, то называется *префиксным инкрементом*, если после переменной — *постфиксным инкрементом*.

Пока вам может показаться, что префиксный и постфиксный варианты этого оператора ничем не отличаются друг от друга, однако это не так. В ситуации, когда требуется увеличить на единицу всего одну переменную, оба оператора действительно дают одинаковый результат. Но в более сложном выражении результаты могут различаться.

Чтобы продемонстрировать это важное различие, выполним вычисление, которое может потребоваться в финале игрового уровня. Определим бонус, зависящий

от количества жизней, сохранившихся у игрока, и увеличим на единицу количество жизней. В первом случае я использую префиксный оператор инкремента:

```
int bonus = ++lives * 10;
```

Префиксный оператор инкремента увеличивает переменную на единицу до того, как будет вычислено большое выражение, содержащее эту переменную. Выражение `++lives*10` вычисляется так: сначала `lives` увеличивается на единицу, а затем полученный результат умножается на 10. Следовательно, этот код эквивалентен 4 · 10, что равно 40. Таким образом, теперь `lives` равно 4, а `bonus` равно 40.

Вновь задав для переменной `lives` значение 3, я еще раз вычисляю `bonus`, на этот раз с постфиксным оператором инкремента:

```
bonus = lives++ * 10;
```

Постфиксный оператор инкремента увеличивает переменную на единицу после того, как будет вычислено большое выражение, содержащее эту переменную. Выражение `++lives*10` вычисляется путем умножения текущего значения `lives` на 10. Следовательно, этот код эквивалентен 3 · 10, что равно 30. Таким образом, теперь `lives` равно 4, а `bonus` — 30.

В языке C++ также существует оператор декремента (`--`). Он действует в точности как оператор инкремента, но не увеличивает, а уменьшает значение на единицу. Он также существует в двух вариантах: как префиксный и как постфиксный.

Что делать с целочисленным переполнением

Что происходит, если значение целочисленной переменной превышает максимальный допустимый для нее предел? Оказывается, никакой ошибки не генерируется. Вместо этого значение сбрасывается на минимум, предусмотренный для данного типа. Далее я продемонстрирую этот феномен. Сначала присвою переменной `score` максимальное значение, которое она может содержать:

```
score = 4294967295;
```

Затем увеличу значение переменной на единицу:

```
++score;
```

В результате значение `score` становится равным 0, так как значение сбрасывается на минимум, примерно как в автомобильном одометре, достигающем максимального значения (рис. 1.7).

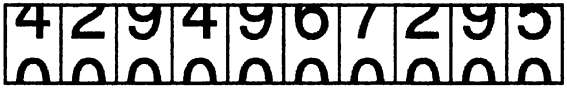


Рис. 1.7. Наглядное представление переполнения переменной unsigned int — переход от максимального значения к минимальному

При декременте целочисленного значения ниже его минимального предела переменная получает максимальное значение, предусмотренное для данного типа.

СОВЕТ

Обязательно подбирайте для работы такой целочисленный тип, который обладает достаточным диапазоном значений для решения стоящих перед вами задач.

Работа с константами

Константа — это неизменяемое значение, которому вы даете имя. Константы удобны в тех случаях, когда в программе есть часто встречающееся постоянное значение. Например, если вы программируете космический шутер, где за каждого сбитого пришельца дается 150 очков, то можно определить константу ALIEN_POINTS, равную 150. Таким образом, всякий раз, когда вам понадобится значение пришельца, можно использовать константу ALIEN_POINTS, а не литерал 150.

Константы обладают двумя важными достоинствами. Во-первых, с ними программа становится яснее. Вы видите ALIEN_POINTS — и сразу понимаете, что это значит. Ведь если вы посмотрите в код и увидите значение 150, то можете и не понять, чему оно соответствует. Во-вторых, константы упрощают изменения. Допустим, вы проходите игру в тестовом режиме и решаете, что каждый сбитый пришелец должен давать 250 очков. При работе с константами для этого достаточно изменить в программе инициализацию ALIEN_POINTS. Без констант придется выловить в коде все значения 150 и заменить каждое из них на 250.

Знакомство с программой Game Stats 3.0

В программе Game Stats 3.0 для представления значений используются константы. Сначала программа подсчитывает очки пользователя, а затем вычисляет стоимость усовершенствования юнита в стратегической игре. Результат показан на рис. 1.8.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 1, имя файла — game_stats3.cpp.

```
// Программа Game Stats 3.0
// Демонстрирует работу с константами
#include <iostream>
using namespace std;
int main()
{
    const int ALIEN_POINTS = 150;
    int aliensKilled = 10;
    int score = aliensKilled * ALIEN_POINTS;
    cout << "score: " << score << endl;
    enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};
    difficulty myDifficulty = EASY;
```

```
    enum shipCost {FIGHTER_COST = 25, BOMBER_COST, CRUISER_COST = 50};
    shipCost myShipCost = BOMBER_COST;
    cout << "\nTo upgrade my ship to a Cruiser will cost "
    << (CRUISER_COST - myShipCost) << " Resource Points.\n";
    return 0;
}
```

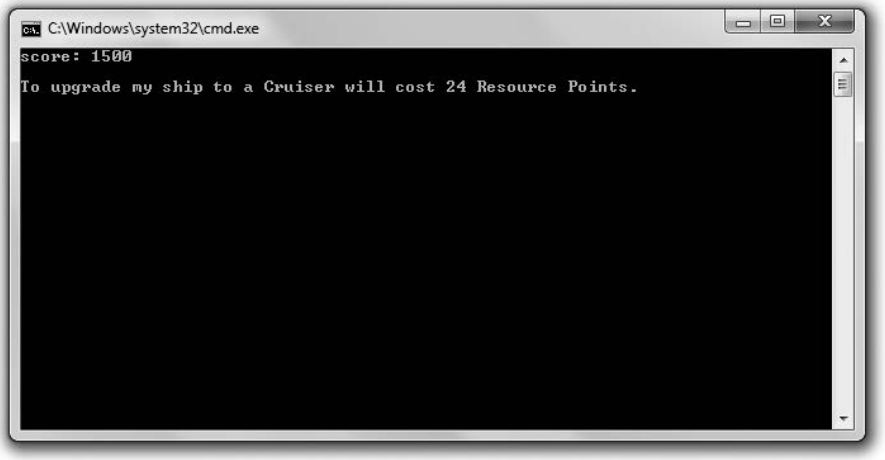


Рис. 1.8. Во всех расчетах используются константы, поэтому на внутрисистемном уровне код становится яснее (опубликовано с разрешения компании Microsoft)

Использование констант

Я определяю константу ALIEN_POINTS, чтобы указать, сколько очков стоит один сбитый пришелец:

```
const int ALIEN_POINTS = 150;
```

Чтобы изменить определение, я просто применяю ключевое слово const. Теперь я могу работать с ALIEN_POINTS как с обычным числовым литералом. Обратите внимание: в имени константы использованы только заглавные буквы. Это просто соглашение, но повсеместно распространенное. Если в идентификаторе только заглавные буквы, то программист сразу понимает, что это константа.

Далее я использую константу в следующей строке:

```
int score = aliensKilled * ALIEN_POINTS;
```

Я вычисляю баллы игрока, умножая количество сбитых пришельцев на «стоимость» одного пришельца. В данном случае константа значительно упрощает код.

ОСТОРОЖНО!

Константе нельзя присвоить новое значение. Если попытаетесь это сделать, получите ошибку компиляции.

Работа с перечислениями

Перечисление — это множество констант unsigned int, называемых *перечислителями*. Как правило, перечислители взаимосвязаны и располагаются в определенном порядке. Вот пример перечисления:

```
enum difficulty {NOVICE, EASY, NORMAL, HARD, UNBEATABLE};
```

Здесь определено перечисление под названием difficulty. По умолчанию значение перечислителей начинается с нуля и у каждого следующего перечислителя увеличивается на единицу. Так, NOVICE равно 0, EASY — 1, NORMAL — 2, HARD — 3, UNBEATABLE — 4. Чтобы самостоятельно определить перечисление, используйте ключевое слово enum, за которым идет идентификатор, а далее — список перечислителей, заключенных в фигурные скобки.

Далее я создаю переменную этого нового для нас перечисляемого типа:

```
DifficultymyDifficulty = EASY;
```

Переменная myDifficulty устанавливается в значение EASY (равное 1). Переменная myDifficulty относится к типу difficulty, поэтому может содержать только одно значение, причем одно из тех, которые определены в перечислении. Таким образом, переменной myDifficulty может быть присвоено только одно из следующих значений: NOVICE, EASY, NORMAL, HARD, UNBEATABLE, 0, 1, 2, 3 или 4.

Далее я определяю другое перечисление:

```
enum shipCost {FIGHTER_COST = 25, BOMBER_COST, CRUISER_COST = 50};
```

Этот код определяет перечисление shipCost, отражающее стоимость трех типов кораблей в стратегической игре в пересчете на ресурсные баллы. Здесь я присваиваю некоторым из перечислителей конкретные целочисленные значения. Числа соответствуют стоимости корабля каждого типа в пересчете на ресурсные баллы. Если хотите, можете присваивать перечислителям значения. Любой перечислитель, которому не присвоено значение, получает значение предыдущего перечислителя, увеличенное на единицу. Поскольку я не присваиваю значения перечислителю BOMBER_COST, он инициализируется со значением 26.

Далее я определяю переменную этого нового перечисляемого типа:

```
shipCostmyShipCost = BOMBER_COST;
```

Вот как перечислители можно использовать при арифметических вычислениях:

```
(CRUISER_COST - myShipCost)
```

Этот код вычисляет, сколько стоит усовершенствовать бомбардировщик до крейсера. Фактически здесь выполняется операция 50 – 26, результат которой равен 24.

Игра «Утраченный клад»

Заключительный проект этой главы — персонализированная приключенческая игра Lost Fortune («Утраченный клад»). Здесь пользователь вводит в систему

определенную информацию, на основании которой компьютер модифицирует приключенческий сюжет. Примерный фрагмент сюжета этой игры показан на рис. 1.9.

Рис. 1.9. Сюжет игры изменен с учетом информации, предоставленной пользователем (опубликовано с разрешения компании Microsoft)

Я покажу код игры не целиком, а частями, отдельно прокомментировав каждый фрагмент. Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 1, имя файла — `lost_fortune.cpp`.

Настройка параметров программы

Сначала я пишу вводные комментарии, включаю два необходимых файла, задаю несколько директив `using`:

```
// Утраченный клад
// Персонализированная приключенческая игра
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::endl;
usingstd::string;
```

Я включаю файл `string`, входящий в состав стандартной библиотеки. Теперь я могу использовать объект `string` для доступа к строке через переменную. Объекты `string` гораздо более многофункциональны, но мы подробнее поговорим о них только в главе 3, где будем программировать игру «Словомеска».

Кроме того, я использую директивы `using`, с помощью которых называю объекты из пространства имен `std`, так как планирую обращаться к этим объектам. Итак, вы можете убедиться, что объект `string` относится к пространству имен `std`.

Получение информации от игрока

Далее я получаю от игрока определенную информацию:

```
int main()
{
    const int GOLD_PIECES = 900;
    int adventurers, killed, survivors;
    string leader;
    // получаю информацию
    cout << "Welcome to Lost Fortune\n\n";
    cout << "Please enter the following for your personalized adventure\n";
    cout << "Enter a number: ";
    cin >> adventurers;
    cout << "Enter a number, smaller than the first: ";
    cin >> killed;
    survivors = adventurers - killed;
    cout << "Enter your last name: ";
    cin>>leader;
```

GOLD_PIECES — это константа, в которой хранится количество кусков золота, спрятанных в кладе, который ищет персонаж. В переменной adventurers хранится количество героев, отправляющихся на поиски клада. В переменной killed хранится количество погибших на пути к сокровищу. В переменной survivors я вычисляю количество путешественников, которые пока остаются в живых. Наконец, я получаю имя пользователя, к этому значению смогу обращаться через переменную leader.

ОСТОРОЖНО!

Объект cin позволяет получить от пользователя символьную строку, только если в этой строке нет пробелов (или табуляции). Существуют способы, позволяющие обойти это ограничение, однако они связаны с использованием потоков (streams), обсуждение которых выходит за рамки этой главы. Итак, если используете объект cin в таком качестве, помните о связанных с ним ограничениях.

Сюжет

Далее с помощью переменных я излагаю сюжет игры:

```
    // сюжет
    cout << "\nA brave group of " << adventurers << " set out on a quest ";
    cout << "-- in search of the lost treasure of the Ancient Dwarves. ";
    cout << "The group was led by that legendary rogue, " << leader << ".\n";
    cout << "\nAlong the way, a band of marauding ogres ambushed the party. ";
    cout << "All fought bravely under the command of " << leader;
    cout << ", and the ogres were defeated, but at a cost. ";
    cout << "Of the adventurers, " << killed << " were vanquished, ";
    cout << "leaving just " << survivors << " in the group.\n";
```

```
    cout << "\nThe party was about to give up all hope. ";
    cout << "But while laying the deceased to rest. ";
    cout << "they stumbled upon the buried fortune. ";
    cout << "So the adventurers split " << GOLD_PIECES << " gold pieces.";
    cout << leader << " held on to the extra " << (GOLD_PIECES % survivors);
    cout << " pieces to keep things fair of course.\n";
    return 0;
}
```

Код и захватывающий сюжет предельно ясны. Однако я остановлюсь еще на одном моменте. Чтобы подсчитать количество кусков золота, которые достаются предводителю, я использую оператор деления по модулю в выражении `GOLD_PIECES % survivors`. Это выражение и дает остаток `GOLD_PIECES / survivors`, получающийся после того, как каждый из уцелевших кладоискателей получит свою долю.

Резюме

Прочитав эту главу, вы должны были усвоить следующий материал.

- C++ — это основной язык, используемый для программирования первоклассных компьютерных игр.
- Программа на языке C++ представляет собой серию инструкций.
- Основные элементы жизненного цикла программы на C++ таковы: идея, план, исходный код, объектный файл, исполняемый файл.
- Большинство ошибок, допускаемых при программировании, относятся к одной из трех категорий: ошибки компиляции, ошибки компоновки и ошибки времени исполнения.
- Функция — это группа программных инструкций, которые могут выполнить определенную работу и вернуть значение.
- Любая программа должна содержать функцию `main()`, которая является исходной точкой всей программы.
- Директива `#include` приказывает препроцессору включить в данный файл какой-то другой файл.
- Стандартная библиотека — это набор функций, которые можно включить в файлы вашей программы для выполнения основного функционала, например ввода и вывода.
- Файл `iostream`, входящий в состав стандартной библиотеки, содержит код, помогающий реализовать стандартный ввод/вывод.
- Пространство имен `std` включает в себя элементы из стандартной библиотеки. Чтобы обратиться к элементу из этого пространства имен, перед элементом нужно поставить префикс `std::` либо воспользоваться директивой `using`.
- `cout` — это объект, определенный в файле `iostream` и используемый для отправки данных в поток стандартного вывода (как правило, на экран компьютера).

- `cin` — это объект, определенный в файле `iostream` и используемый для получения данных из потока стандартного ввода (как правило, с клавиатуры).
- В языке C++ есть встроенные арифметические операторы: всем известные действия сложения, вычитания, умножения, деления, а также менее известный оператор деления по модулю.
- В языке C++ определяются примитивы (базовые типы) для булевых, односимвольных, целочисленных значений, а также для значений с плавающей запятой.
- В стандартной библиотеке C++ есть особый тип объектов (`string`) для работы с символьными строками.
- С помощью ключевого слова `typedef` можно создать новое имя для уже существующего типа.
- Константа — это имя неизменяемого значения.
- Перечисление — это последовательность констант типа `unsigned int`.

Вопросы и ответы

1. *Почему компании, занимающиеся разработкой компьютерных игр, используют язык C++?*

   Язык C++ обладает такими достоинствами, как высокая скорость, низкоуровневый доступ к оборудованию, а также высокоуровневые конструкции, причем эта комбинация лучше, чем в любом другом языке. Кроме того, многие игровые компании активно инвестируют в ресурсы для языка C++ (как в создание кода для многоразового использования, так и в обучение программистов).

2. *Чем язык C++ отличается от языка C?*

   Язык C++ можно назвать следующей итерацией языка C. Чтобы получить широкое признание, язык C++ сохранил практически весь материал C. Однако многие задачи решаются в C++ по-новому и более эффективно, чем в языке C. Кроме того, на C++ удобно писать объектно-ориентированные программы.

3. *Чем C++ отличается от языка C#?*

   Язык C# был разработан компанией Microsoft и задумывался одновременно как простой и универсальный. C# испытал значительное влияние C++, но при всей схожести это два самостоятельных языка.

4. *Как пользоваться комментариями?*

   Комментарии нужны для объяснения необычного или сложного кода. Комментировать тривиальный код не следует.

5. *Что такое программный блок?*

   Это одна или несколько инструкций, образующие целостную единицу и заключенные в фигурные скобки.

6. *Что такое предупреждение компилятора?*

   Это сообщение компилятора, указывающее на потенциальную проблему. Предупреждение не прерывает процесс компиляции.

7. *Можно ли игнорировать предупреждения компилятора?*

   Можно, но, как правило, не следует. Нужно разобраться с предупреждением и исправить тот код, который его вызвал.

8. *Что такое символы пробела?*

   Это набор символов, которые не выводятся на печать и поэтому образуют пробелы в файлах с исходным кодом. Примеры таких символов — это пробелы, табуляция и символы перехода на новую строку.

9. *Что такое литералы?*

   Это объекты, буквально соответствующие своим значениям. Так, `"Game Over!"` — это строковый литерал, `32` и `98.6` — числовые литералы.

10. *Почему новую переменную всегда следует инициализировать со значением?*

    Дело в том, что содержимым неинициализированной переменной может оказаться любое значение, даже такое, которое совершенно бессмысленно в рамках вашей программы.

11. *Для чего предназначены переменные типа bool?*

    Такие переменные представляют условия, которые могут соблюдаться или не соблюдаться, то есть находиться в одном из двух состояний: «истина» или «ложь». Например, с помощью таких переменных можно указать, закрыт ли ящик, лежит ли карта рубашкой вниз.

12. *Почему булевы значения получили такое название?*

    Этот тип значений назван в честь английского математика Джорджа Буля.

13. *Обязательно ли записывать имена констант прописными буквами?*

    Нет, просто так принято. Тем не менее этой практики следует придерживаться, чтобы другим программистам было проще читать ваш код.

14. *Как можно сохранить в одной строковой переменной не один, а несколько символов?*

    С помощью объекта `string`.

Вопросы для обсуждения

1. Как общепринятый стандарт C++ помогает работать программистам игр?
2. Каковы достоинства и недостатки, связанные с использованием директивы `using`?
3. Почему может потребоваться определить новое имя для уже существующего типа?
4. Почему существуют две версии оператора инкремента? В чем заключается разница между ними?
5. Как можно улучшить свой код с помощью констант?

Упражнения

1. Составьте список из шести допустимых имен переменных, причем три имени должны быть качественными, а три — некачественными. Объясните, почему каждое из имен относится к той или иной категории.
2. Какая информация содержится в следующем блоке кода? Объясните каждый из результатов.

```
cout << "Seven divided by three is " << 7 / 3 << endl;
cout << "Seven divided by three is " << 7.0 / 3 << endl;
cout << "Seven divided by three is " << 7.0 / 3.0 << endl;
```

3. Напишите программу, которая принимает у пользователя три суммы очков за разные попытки в игре и выводит среднее значение на основе трех.

2 Истина, ветвление и игровой цикл. Игра «Угадай число»

Все программы, которые мы писали до сих пор, были линейными. В линейной программе все инструкции выполняются по порядку сверху вниз. Но для создания игр требуется писать такие программы, которые выполняют или пропускают те или иные блоки кода в зависимости от выполнения/невыполнения определенного условия. Это основная тема данной главы. В частности, в этой главе вы научитесь:

- понимать истину (в контексте языка C++);
- использовать инструкции `if` для ветвления программы между блоками кода;
- использовать инструкции `switch`, выбирая блок кода для выполнения;
- использовать циклы `while` и `do` для многократного выполнения блоков кода;
- генерировать случайные числа.

Понятие об истине

Существует либо истина, либо ложь, никаких полутонов. Именно так обстоит ситуация в языке C++. Истина и ложь в этом языке обозначаются соответствующими ключевыми словами: `true` и `false`. Такое (булево) значение можно хранить в переменных типа `bool`, которые были рассмотрены в главе 1. Напомню, как это делается:

```
bool fact = true, fiction = false;
```

В данном коде создаются две переменные типа `bool`: `fact` и `fiction`. `fact` равно `true`, `fiction` равно `false`. Хотя ключевые слова `true` и `false`, конечно, удобны, любое выражение или значение также может быть интерпретировано как `true` или `false`. Любое ненулевое значение может соответствовать `true`, а `0` — `false`.

Типичные выражения, интерпретируемые как `true` или `false`, связаны со сравнением сущностей. Сравнения часто выполняются с помощью встроенных реляционных операторов. В табл. 2.1 приведены эти операторы и примеры выражений с ними.

Таблица 2.1. Реляционные операторы

Оператор	Значение	Пример выражения	Вычисляется в
==	Равно	5 == 5 5 == 8	true false
!=	Не равно	5 != 8 5 != 5	true false
>	Больше	8 > 5 5 > 8	true false
<	Меньше	5 < 8 8 < 5	true false
>=	Больше или равно	8 >= 5 5 >= 8	true false
<=	Меньше или равно	5 <= 8 8 <= 5	true false

Использование инструкции if

Итак, давайте рассмотрим концепции `true` и `false` на практике. Инструкцию `if` можно использовать для проверки того, истинно ли выражение, и в зависимости от этого выполнить тот или иной код. Вот пример инструкции `if`:

```
if (expression)
statement;
```

Если `expression` равно `true`, то инструкция выполняется. В противном случае `statement` пропускается и программа переходит к выполнению первой инструкции, следующей за блоком `if`.

СОВЕТ

Если вы видите обобщенное слово statement, как в предыдущем примере кода, то можете заменить его как отдельной инструкцией, так и блоком инструкций, поскольку блок интерпретируется как единое целое.

Знакомство с программой Score Rater

Программа Score Rater комментирует баллы, набранные пользователем. При этом применяется инструкция `if`. На рис. 2.1 эта программа показана в действии.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `score_rater.cpp`.

```
// Программа Score Rater
// Демонстрирует работу с инструкцией if
#include <iostream>
using namespace std;
int main()
{
    if (true)
```

```
    {
        cout << "This is always displayed.\n\n";
    }
    if (false)
    {
        cout << "This is never displayed.\n\n";
    }
    int score = 1000;
    if (score)
    {
    cout << "At least you didn't score zero.\n\n";
    }
    if (score >= 250)
    {
    cout << "You scored 250 or more. Decent.\n\n";
    }
    if (score >= 500)
    {
        cout << "You scored 500 or more. Nice.\n\n";
        if (score >= 1000)
        {
            cout << "You scored 1000 or more. Impressive!\n";
        }
    }
    return 0;
}
```

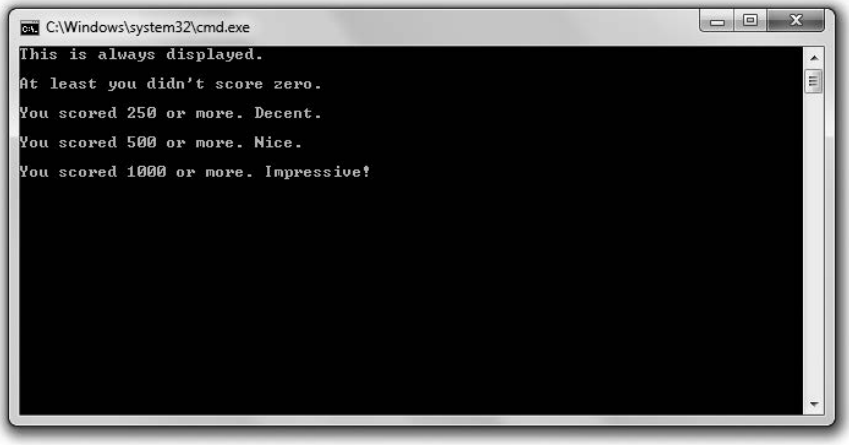


Рис. 2.1. В зависимости от различных инструкций if на экране отображаются (или не отображаются) сообщения (опубликовано с разрешения компании Microsoft)

Проверка условий true и false

В первой инструкции `if` я проверяю `true`. Поскольку `true` — это истина, программа выводит сообщение `This is always displayed`:

```
if (true)
{
    cout << "This is always displayed.\n\n";
}
```

В следующей инструкции `if` я проверяю `false`. Поскольку `false` — это не истина, программа не выводит сообщение `This is never displayed`:

```
if (false)
{
    cout << "This is never displayed.\n\n";
}
```

ОСТОРОЖНО!

Обратите внимание: мы не ставим точку с запятой после закрывающей фигурной скобки выражения, тестируемого в инструкции if. Если бы мы поставили точку с запятой, то создали бы пустую инструкцию, спаренную с инструкцией if и, в сущности, сводящую на нет всю пользу от if. Вот пример:

```
if (false);
{
cout << "This is never displayed.\n\n";
}
```

Поставив точку с запятой после (false), я создаю пустую инструкцию, ассоциированную с инструкцией if. Предыдущий код эквивалентен следующему:

```
if (false)
; // пустая инструкция, которая ничего не делает
{
cout << "This is never displayed.\n\n";
}
```

Здесь я просто поиграл с пробелами, не меняющими значения кода. Итак, теперь проблема должна быть ясна. Инструкция if встречает значение false и пропускает следующую инструкцию (пустую). Далее программа вновь проделывает «полный круг» до инструкции, следующей за инструкцией if, после чего на экране появляется сообщение This is never displayed.

Остерегайтесь такой ошибки. Она не считается недопустимой, поэтому не вызывает и ошибок компиляции.

Интерпретация значения как истинного или ложного

Любое значение можно интерпретировать как истинное или ложное (`true` или `false`). Например, любое ненулевое значение может быть расценено как `true`, а 0 — как `false`. Эту зависимость я проверяю в следующей инструкции `if`:

```
if (score)
{
    cout << "At least you didn't score zero.\n\n";
}
```

Значение `score` составляет 1000, то есть оно ненулевое и равно `true`. В результате на экран выводится сообщение «Ваш результат ненулевой, уже хорошо».

Работа с реляционными операторами

Вероятно, чаще всего с инструкциями `if` используются такие выражения, в которых значения сравниваются с помощью реляционных операторов. Именно такой случай рассмотрен далее. Я проверю, равны набранные очки значению 250 или превышают его:

```
if (score >= 250)
{
    cout << "You scored 250 or more. Decent.\n\n";
}
```

Поскольку значение `score` равно 1000, блок выполняется и на экран выводится сообщение о том, сколько очков честно заработал пользователь. Если бы было набрано менее 250 очков, этот блок был бы пропущен и выполнение программы продолжилось бы с первой инструкции, следующей за этим блоком.

ОСТОРОЖНО!

Реляционный оператор равенства — это == (два знака равенства подряд). Не путайте его с = (это оператор присваивания).

Использовать оператор присваивания вместо реляционного оператора равенства не запрещено, однако результат, вероятно, будет не таким, на который вы рассчитывали. Рассмотрим следующий код:

```
int score = 500;
if (score = 1000)
{
    cout << " You scored 1000 or more. Impressive!\n";
}
```

В результате выполнения этого кода score получает значение 1000, а на экране появляется сообщение You scored 1000 or more. Impressive!. Вот что здесь происходит: хотя до выполнения инструкции if значение score равно 500, эта ситуация меняется. На этапе вычисления выражения из инструкции if (score = 1000) переменная score получает значение 1000. Инструкция присваивания вычисляется в 1000, а поскольку это ненулевое значение, выражение интерпретируется как true. В результате на экране отображается строка.

Остерегайтесь такой ошибки. Ее легко допустить, а в некоторых случаях (таких как рассмотренный) она не вызывает ошибок компиляции.

Вложение инструкций if

Под действием инструкции `if` программа может выполнить инструкцию или блок инструкций, в том числе другие инструкции `if`. Ситуация, когда одна инструкция `if` написана внутри другой, называется *вложением*. В следующем коде инструкция `if`, начинающаяся с `if (score >= 1000)`, вложена в инструкцию `if`, начинающуюся с `if (score >= 500)`:

```
if (score >= 500)
{
    cout << "You scored 500 or more. Nice.\n\n";
    if (score >= 1000)
```

```
    {
        cout << "You scored 1000 or more. Impressive!\n";
    }
}
```

Поскольку значение score превышает 500, программа переходит к блоку инструкций и выводит на экран сообщение You scored 500 or more. Nice. Затем во внутренней инструкции if программа сравнивает score и 1000. Поскольку score больше или равно 1000, программа выводит на экран сообщение You scored 1000 or more. Impressive!.

СОВЕТ

Глубина вложений не ограничена. Правда, если в вашем коде есть слишком глубокие вложения, его становится сложнее читать. В принципе, чем меньше уровней вложения в коде, тем лучше.

Работа с условием else

В инструкцию if можно добавить условие else. Так мы получим код, который будет выполняться лишь в том случае, если проверенное условие равно false. Ниже показана схема инструкции if, содержащей условие else:

```
if (expression)
    statement1;
else
    statement2;
```

Если expression равно true, то выполняется инструкция statement1. Затем программа пропускает statement2 и выполняет ту инструкцию, которая идет за блоком if. Если expression равно false, то инструкция statement1 пропускается и выполняется инструкция statement2. Когда выполнение statement2 завершится, программа выполнит ту инструкцию, которая следует сразу за блоком if.

Знакомство с программой Score Rater 2.0

Программа Score Rater 2.0 также оценивает набранные баллы, вводимые пользователем. Но на этот раз программа использует инструкцию if с условием else. На рис. 2.2 и 2.3 показаны три разных сообщения, которые может вывести на экран программа. Конкретное сообщение выбирается в зависимости от того, сколько баллов набрал пользователь.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — score_rater2.cpp.

```
// Программа Score Rater 2.0
// Демонстрирует работу с условием else
#include <iostream>
using namespace std;
int main()
{
```

```
    int score;
    cout << "Enter your score: ";
    cin >> score;
    if (score >= 1000)
    {
        cout << "You scored 1000 or more. Impressive!\n";
    }
    else
    {
        cout << "You scored less than 1000.\n";
}
return 0;
}
```

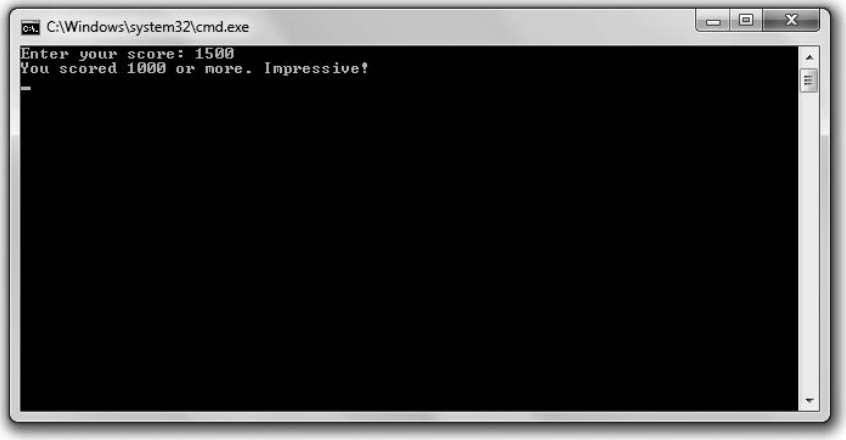


Рис. 2.2. Если пользователь набирает 1000 очков или больше, то получает поздравление (опубликовано с разрешения компании Microsoft)

Рис. 2.3. Если пользователь набирает менее 1000 очков, то поздравления не получает (опубликовано с разрешения компании Microsoft)

Создание двух способов ветвления

Вы уже видели первую часть инструкции if, она работает точно так же, как и в предыдущих примерах. Если значение score больше 1000, то на экране отображается сообщение You scored 1000 or more. Impressive!:

```
if (score >= 1000)
{
    cout << "You scored 1000 or more. Impressive!\n";
}
```

Вот что здесь происходит. Условие else предоставляет инструкцию, по которой должно пойти (ответвиться) выполнение программы, если выражение дает в результате false. Итак, если (score >= 1000) равно false, то программа пропустит первое сообщение, а вместо него выведет на экран фразу You scored less than 1000:

```
else
{
    cout << "You scored less than 1000.\n";
}
```

Использование последовательности инструкций с помощью условий else

Можно сочленять инструкции if, сцепляя их условиями else. В результате получается последовательность выражений, которые проверяются по порядку. Инструкция, ассоциированная с первым проверяемым выражением, выполняется, если это выражение дает в результате true, в противном случае выполняется инструкция, ассоциированная с последним (необязательным) условием else. Вот какой вид может принять подобная последовательность:

```
if (expression1)
    statement1;
else if (expression2)
    statement2;
...
else if (expressionN)
    statementN;
else
    statementN+1;
```

Если expression1 равно true, то выполняется инструкция statement1, а весь остальной код в последовательности пропускается. В противном случае проверяется выражение expression2. Если оно равно true, то выполняется инструкция statement2, а весь остальной код в последовательности пропускается. Компьютер продолжает проверять все выражения по очереди (вплоть до выражения expressionN) и выполнит инструкцию, связанную с первым же выражением, которое дает в результате true. Если ни одно из выражений не окажется истинным, то будет выполнена инструкция, ассоциированная с заключительным условием else, то есть statementN+1.

Знакомство с программой Score Rater 3.0

Программа Score Rater 3.0 также оценивает балл, вводимый пользователем. Но на этот раз программа использует последовательность инструкций `if` с условиями `else`. На рис. 2.4 показан результат выполнения этой программы.

Рис. 2.4. Пользователь может получить одно из нескольких сообщений в зависимости от того, сколько баллов он набрал (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `score_rater3.cpp`.

```
// Программа Score Rater 3.0
// Демонстрирует работу с последовательностью ifelse-ifelse
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "Enter your score: ";
    cin >> score;
    if (score >= 1000)
    {
        cout << "You scored 1000 or more. Impressive!\n";
    }
    else if (score >= 500)
    {
        cout << "You scored 500 or more. Nice.\n";
    }
    else if (score >= 250)
    {
        cout << "You scored 250 or more. Decent.\n";
    }
```

```
    else
    {
        cout << "You scored less than 250. Nothing to brag about.\n";
}
    return 0;
}
```

Создание последовательности инструкций if с применением условий else

Мы уже дважды рассмотрели первую часть этой последовательности, в данном варианте этот код будет работать точно так же, как и в предыдущих. Если значение переменной score больше или равно 1000, то на экран выводится сообщение You scored 1000 or more. Impressive!. Программа ветвится, и компьютер выполняет инструкцию return:

```
if (score>= 1000)
```

Но если выражение равно false, то мы знаем, что score меньше 1000, и компьютер интерпретирует следующее выражение в последовательности:

```
else if (score >= 500)
```

Если score больше или равно 500, то на экран выводится фраза You scored 500 or more. Nice, программа ветвится и компьютер выполняет инструкцию return. Но если выражение равно false, то мы знаем, что score меньше 500, и компьютер интерпретирует следующее выражение в последовательности:

```
else if (score >= 250)
```

Если score больше или равно 250, то на экран выводится фраза You scored 250 or more. Decent, программа ветвится и компьютер выполняет инструкцию return. Но если выражение равно false, то мы знаем, что score меньше 250, выполняется инструкция, ассоциированная с последним условием else, и на экран выводится сообщение You scored less than 250. Nothing to brag about.

СОВЕТ

Хотя последнее выражение else в этом наборе инструкций if else... if не является обязательным, именно так должен был бы выполняться код, если в последовательности нет ни одного верного выражения.

Использование инструкции switch

Инструкция switch используется для создания в коде множества точек ветвления. Вот упрощенное представление инструкции switch:

```
switch (choice)
{
    case value1:
        statement1;
```

```
        break;
    case value2:
        statement2;
        break;
    case value3:
        statement3;
        break;
    case valueN:
statementN;
break;
    default:
statementN + 1;
}
```

Данная инструкция по порядку сравнивает варианты (choice) с возможными значениями (value1, value2 и value3). Если вариант равен одному из значений, то программа выполняет инструкцию (statement), соответствующую этому значению. Встретив инструкцию break, программа покидает структуру switch. Если choice не совпадает ни с одним значением, то выполняется инструкция, ассоциированная с факультативным вариантом default.

Использовать break и default необязательно. Но если опустить break, то программа продолжит выполнять остальные инструкции, пока не встретит break или default либо пока не закончится инструкция switch. Как правило, инструкция break ставится в конце каждого условия case.

СОВЕТ

Хотя условие default не является обязательным, его удобно ставить в конце как универсальный уловитель (catchall).

Закрепим эту теорию на примере. Допустим, choice равно value2. Сначала программа сравнит choice с value1. Поскольку два этих значения не равны, выполнение программы продолжится. Далее программа сравнит choice с value2. Поскольку эти значения равны, программа выполнит statement2. Затем программа дойдет до инструкции break и выйдет из структуры switch.

ОСТОРОЖНО!

Инструкция switch подходит только для тестирования целочисленных значений int либо любого значения, которое можно интерпретировать как int — например, char или enumerator. Инструкция switch не сработает с данными других типов.

Знакомство с программой Menu Chooser

Программа Menu Chooser предлагает пользователю меню, в котором указаны три уровня сложности, один из которых нужно выбрать. Если пользователь вводит число, соответствующее тому или иному варианту, то получает сообщение, подтверждающее сделанный выбор. Если он вводит какое-то другое число, то программа сообщает, что такой вариант отсутствует. На рис. 2.5 эта программа показана в действии.

Рис. 2.5. По всей видимости, я легко отделался
(опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `menu_chooser.cpp`.

```
// Программа Menu Chooser
// Демонстрирует работу с инструкцией switch
#include <iostream>
using namespace std;
int main()
{
    cout << "Difficulty Levels\n\n";
    cout << "1 - Easy\n";
    cout << "2 - Normal\n";
    cout << "3 - Hard\n\n";
    int choice;
    cout << "Choice: ";
    cin >> choice;
    switch (choice)
    {
        case 1:
  cout << "You picked Easy.\n";
            break;
        case 2:
            cout << "You picked Normal.\n";
            break;
        case 3:
            cout << "You picked Hard.\n";
            break;
        default:
            cout << "You made an illegal choice.\n";
}
return 0;
}
```

Создание нескольких вариантов ветвления

Инструкция switch создает четыре возможные точки ветвления. Если пользователь вводит 1, выполняется код, связанный с условием case 1, и на экран выводится сообщение You picked Easy. Если пользователь вводит 2, выполняется код, связанный с условием case 2, и на экран выводится сообщение You picked Normal. Если пользователь вводит 3, выполняется код, связанный с условием case 3, и на экран выводится сообщение You picked Hard. Если пользователь вводит любое другое значение, срабатывает вариант default и на экран выводится сообщение You made an illegal choice.

ОСТОРОЖНО!

Практически все условия всегда нужно оканчивать инструкцией break. Не забывайте ее, иначе ваш код может сработать непредсказуемым образом.

Использование циклов while

Циклы while позволяют многократно выполнять одни и те же блоки кода, пока определенное выражение не даст в результате true. Вот упрощенная форма цикла while:

```
while (expression)
statement;
```

Если expression становится равно false, то программа переходит к выполнению инструкции, следующей за циклом. Если expression равно true, то программа выполняет statement и возвращается на шаг назад — снова тестирует верность expression. Цикл продолжается до тех пор, пока expression не окажется равно false, после чего цикл завершится.

Знакомство с программой Play Again

Программа Play Again имитирует партию в увлекательную игру. То есть она просто выводит на экран сообщение **Played an exciting game**. Затем программа предлагает пользователю сыграть еще раз. Пока пользователь вводит y, он продолжает «играть». Такое повторение обеспечивается в программе с помощью цикла while. На рис. 2.6 эта программа показана в действии.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — play_again.cpp.

```
// Программа Play Again
// Демонстрирует работу с циклами while
#include <iostream>
using namespace std;
int main()
{
    char again = 'y';
    while (again == 'y')
```

```
    {
        cout << "\n**Played an exciting game**";
        cout << "\nDo you want to play again? (y/n): ";
        cin >> again;
    }
    cout << "\nOkay, bye.";
    return 0;
}
```

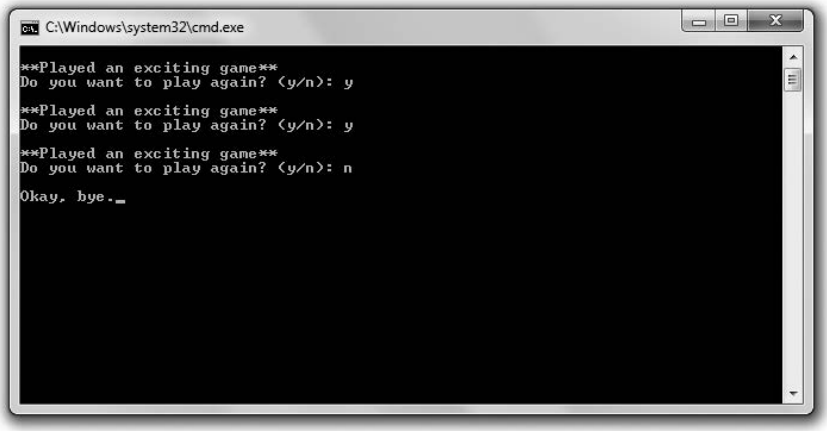


Рис. 2.6. Многократное выполнение кода осуществляется с помощью цикла while (опубликовано с разрешения компании Microsoft)

Работа с циклом while

В самом начале функции `main()` программа объявляет переменную типа `char` под именем `again` и инициализирует ее в значении `y`. Затем программа начинает цикл `while`, проверяя значение `again` и узнавая, равно ли оно `y`. Поскольку так и есть, программа отображает на экране сообщение `**Played an exciting game**`, предлагает пользователю сыграть снова и сохраняет его реакцию в `again`. Цикл продолжается до тех пор, пока в ответ на предложение сыграть пользователь вводит символ `y`.

Как вы заметили, мне пришлось инициализировать переменную `again` до начала цикла, так как эта переменная используется в выражении самого цикла. Поскольку цикл `while` вычисляет свои выражения до начала *тела цикла* (группы повторяемых инструкций), необходимо гарантировать, что еще до начала цикла у всех переменных в выражении будут заданы значения.

Использование циклов do

Как и циклы `while`, циклы `do` позволяют многократно выполнять блок кода, пока выражение дает в результате определенные значения. Разница заключается в том, что цикл `do` проверяет свое выражение после каждой итерации. Таким образом,

тело цикла обязательно выполняется как минимум один раз. Вот упрощенная форма цикла do:

```
do
    statement;
while (expression)
```

Программа выполняет statement, и если после этого expression оказывается равно true, цикл повторяется. Как только expression окажется равно false, цикл завершается.

Знакомство с программой Play Again 2.0

Программа Play Again 2.0 выглядит точно так же, как и исходная версия программы. Она, как и программа-предшественница, имитирует партию в увлекательную игру, выводя на экран сообщение **Played an exciting game** и предлагая пользователю сыграть еще раз. Пока пользователь вводит y, он «продолжает играть». Однако на этот раз программа выполняется на основе цикла do. Она продемонстрирована на рис. 2.7.

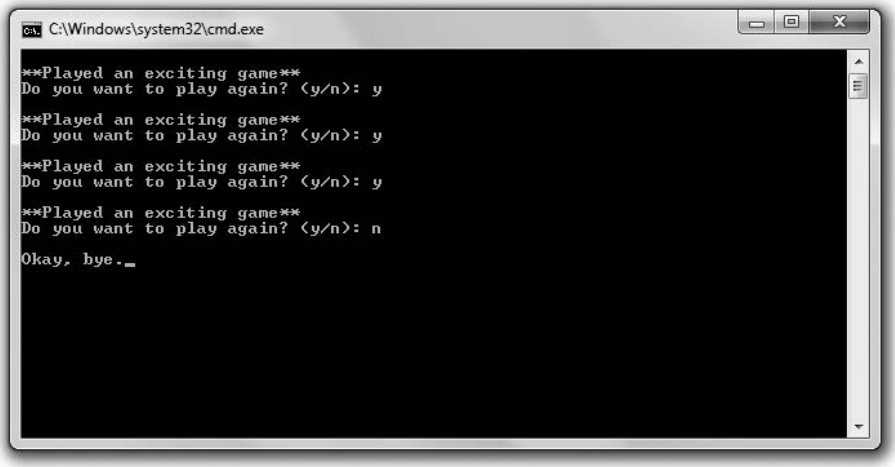


Рис. 2.7. Повторные операции в программе выполняются с помощью цикла do (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — play_again2.cpp.

```
// Программа Play Again 2.0
// Демонстрирует работу с циклами do
#include <iostream>
using namespace std;
int main()
{
    char again;
```

```
    do
    {
        cout << "\n**Played an exciting game**";
        cout << "\nDo you want to play again? (y/n): ";
        cin >> again;
    } while (again == 'y');
    cout << "\nOkay, bye.";
    return 0;
}
```

Работа с циклом do

Перед началом цикла do я объявляю символ again. Правда, мне не требуется его инициализировать, так как он будет проверен только после выполнения первой итерации цикла. Новое значение для переменной again я получаю от пользователя в теле цикла. Затем проверяю again в теле цикла. Если again равно y, то цикл повторяется, в противном случае он завершается.

НА ПРАКТИКЕ

Хотя на практике вполне можно использовать циклы while и do вперемежку друг с другом, программисты предпочитают цикл while. В некоторых случаях цикл do действительно может смотреться более естественно, но цикл while имеет важное достоинство: здесь проверяемое выражение находится в самом начале цикла, а не в конце.

ОСТОРОЖНО!

Если когда-нибудь в компьютерной игре вам приходилось попадать в бесконечную петлю событий, то вы знаете, что такое бесконечный цикл. Вот простой пример такого цикла:

```
int test = 10;
while (test == 10)
{
cout<<test;
}
```

В данном случае цикл начинается потому, что test равно 10. Но поскольку значение test никогда не меняется, этот цикл никогда не остановится. Единственный выход — принудительно завершить программу. Вывод? Необходимо гарантировать, что рано или поздно выражение в цикле обязательно даст в результате false, либо применить другие способы завершения циклов, которые описаны в разделе «Использование инструкций break и continue».

Использование инструкций break и continue

Бесконечного зацикливания, которое описано ранее, легко избежать. Из цикла можно немедленно выйти с помощью инструкции break, а к началу цикла можно сразу же перейти с помощью инструкции continue. Хотя эти палочки-выручалочки следует использовать экономно, иногда они действительно приходятся очень кстати.

Знакомство с программой Finicky Counter

Программа Finicky Counter считает от 1 до 10, это делается с помощью цикла `while`. Название программы означает «Разборчивый счетчик». Действительно, ей не нравится число 5, поэтому она его пропускает. Работа программы проиллюстрирована на рис. 2.8.

Рис. 2.8. Цифра 5 пропускается с помощью инструкции continue, цикл завершается инструкцией break (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `finicky_counter.cpp`.

```
// Программа Finicky Counter
// Демонстрирует работу с инструкциями break и continue
#include <iostream>
using namespace std;
int main()
{
    int count = 0;
    while (true)
{
        count += 1;
// заканчиваем цикл, если значение count превышает 10
        if (count > 10)
        {
            break;
        }
        // пропускаем число 5
        if (count == 5)
        {
            continue;
        }
        cout << count << endl;
    }
    return 0;
}
```

Создание цикла while (true)

Я создаю цикл с помощью следующей строки:

```
while (true)
```

Строго говоря, так возникает бесконечный цикл. Это может показаться странным, ведь еще недавно я предостерегал вас от бесконечных циклов. Но дело в том, что именно этот цикл не будет бесконечным, ведь я запрограммировал в его теле условие выхода.

СОВЕТ

Хотя цикл while (true) порой кажется понятнее обычных, старайтесь свести использование циклов while (true) к минимуму.

Использование инструкции break для выхода из цикла

Вот условие выхода, которое я помещаю в цикле:

```
// заканчиваем цикл, если значение count превышает 10
if (count > 10)
{
    break;
}
```

Поскольку значение переменной count увеличивается на единицу всякий раз, когда начинается тело цикла, через некоторое время эта переменная достигнет значения 11. Как только это произойдет, будет выполнена инструкция break, означающая выход из цикла, и цикл закончится.

Использование инструкции continue для перехода в начало цикла

Прямо перед отображением count я поставил в коде следующие строки:

```
// пропускаем число 5
if (count == 5)
{
    continue;
}
```

Инструкция continue означает «вернуться к началу цикла». В начале цикла проверяется выражение while. Если оно оказывается истинным, цикл начинается заново. Итак, когда count равно 5, программа не выполняет инструкцию cout<<count<<endl;. Вместо этого она начинает цикл заново. Таким образом, цифра 5 пропускается и на экране не отображается.

Когда следует использовать инструкции break и continue

Инструкции `break` и `continue` можно использовать в любом цикле, а не только в циклах `while (true)`. Но следует учитывать, что из-за обеих этих инструкций (`break` и `continue`) программисту становится сложнее читать код и отслеживать в нем развитие цикла.

Использование логических операторов

До сих пор мы рассматривали лишь тривиальные выражения, дающие в результате значения «истина» или «ложь». Однако простые выражения можно комбинировать с помощью *логических операторов*, создавая более сложные. Логические операторы перечислены в табл. 2.2.

Таблица 2.2. Логические операторы

Оператор	Описание	Образец выражения
!	Логическое «НЕ»	!expression
&&	Логическое «И»	expression1 && expression2
||	Логическое «ИЛИ»	expression1 || expression2

Знакомство с программой Designers Network

Программа Designers Network имитирует работу компьютерной сети, членами которой являются лишь немногие избранные дизайнеры игр. Как и в настоящей компьютерной сети, для входа в эту сеть пользователь должен указать свое имя (логин) и пароль. При успешном входе в сеть пользователь получает персональное приветствие. Чтобы войти в сеть как гость, пользователь должен ввести в поле логина или пароля слово `guest`. Программа проиллюстрирована на рис. 2.9–2.11.

Рис. 2.9. Если вы не являетесь членом сети или гостем, то не можете войти (опубликовано с разрешения компании Microsoft)

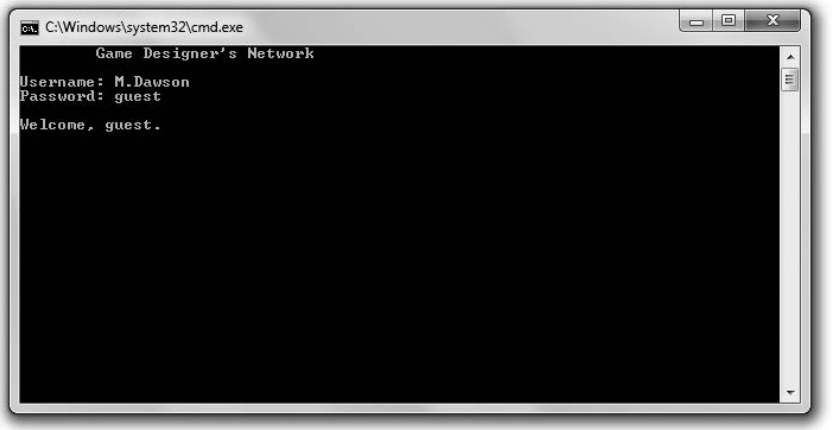


Рис. 2.10. Можно войти в сеть на правах гостя (опубликовано с разрешения компании Microsoft)

Рис. 2.11. По-видимому, в сеть сегодня заходил кто-то из избранных (опубликовано с разрешения компании Microsoft)

Код программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `designers_network.cpp`.

```
// Программа Designers Network
// Демонстрирует работу с логическими операторами
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string username;
    string password;
    bool success;
    cout << "\tGame Designer's Network\n";
    do
    {
        cout << "\nUsername: ";
```

```
        cin >> username;
        cout << "Password: ";
        cin >> password;
        if (username == "S.Meier" && password == "civilization")
        {
            cout << "\nHey, Sid.";
            success = true;
        }
        else if (username == "S.Miyamoto" && password == "mariobros")
        {
            cout << "\nWhat's up, Shigeru?";
            success = true;
        }
        else if (username == "W.Wright" && password == "thesims")
        {
            cout << "\nHow goes it, Will?";
            success = true;
        }
        else if (username == "guest" || password == "guest")
        {
            cout << "\nWelcome, guest.";
            success = true;
        }
        else
        {
            cout << "\nYour login failed.";
            success = false;
        }
    } while (!success);
    return 0;
}
```

Использование логического оператора «И»

Логический оператор «И», &&, позволяет объединить два выражения в одно более крупное, которое будет давать в результате true или false. Новое выражение равно true, только если оба объединяемых выражения равны true, в противном случае оно равно false. Действительно, в русском языке союз «и» означает «оба». Чтобы составное выражение было равно true, оба составляющих его выражения должны быть равны true. Вот конкретный пример из программы Designers Network:

```
if (username == "S.Meier" &&password == "civilization")
```

Выражение username == "S.Meier" &&password == "civilization" будет равно true лишь в том случае, если оба выражения: username == "S.Meier" и password == "civilization" — будут равны true. Механизм работает отлично: я допускаю Сида Майера в сеть лишь в случае, если он знает и свой логин, и свой пароль. Если он знает только одно или другое, доступ закрыт.

Еще один способ разобраться в работе оператора && — рассмотреть все возможные комбинации истинности и ложности (табл. 2.3).

Таблица 2.3. Возможные комбинации пользовательского имени и пароля, использующие логический оператор И

username == "S.Meier"	password == " civilization "	username == "S.Meier" && password == "civilization"
true	true	true
true	false	false
false	true	false
false	false	false

Разумеется, программа Designers Network работает и с другими пользовательскими записями, а не только с аккаунтом Сида Майера. С помощью ряда инструкций `if` с условиями `else` и оператором `&&` программа проверяет различные пары значений `username` и `password`. Если пользователь вводит известную программе пару значений, то получает персональное приветствие.

Использование логического оператора «ИЛИ»

Логический оператор «ИЛИ», `||`, позволяет объединять два выражения в одно более крупное, которое может давать в результате `true` или `false`. Новое выражение будет равно `true`, если первое *или* второе значение окажется равно `true`, в противном случае оно равно `false`. В русском языке слово «или» также означает «одно из двух». Если первое или второе выражение дает в результате `true`, то новое выражение также дает в результате `true`. Если оба «слагаемых» выражения равны `true`, то само выражение также равно `true`. Вот конкретный пример из программы Designers Network:

```
else if (username == "guest" || password == "guest")
```

Выражение `username == "guest" || password == "guest"` равно `true`, если `username == "guest"` равно `true` или `password == "guest"` равно `true`. Механизм работает: действительно, я хотел предоставлять пользователю гостевой доступ, если он введет слово `guest` в качестве имени или пароля. Если он укажет это слово и для имени, и для пароля, тоже сойдет.

Другой способ, позволяющий понять работу оператора `||`, — рассмотреть все возможные комбинации истинности и ложности (табл. 2.4).

Таблица 2.4. Возможные комбинации пользовательского имени и пароля, использующие логический оператор ИЛИ

Username == "guest"	Password == "guest"	username == "guest" || password == "guest"
true	true	true
true	false	true
false	true	true
false	false	false

Использование логического оператора «НЕ»

Логический оператор «НЕ», !, позволяет «переключать» результат выражения с истинного на ложный и наоборот. Новое выражение равно true, если исходное равно false, и наоборот, новое выражение равно false, если исходное равно true. Действительно, в русском языке слово «не» может означать противоположность. Новое выражение по значению противоположно оригиналу.

Я использую оператор «НЕ» в булевом выражении цикла do:

```
} while (!success);
```

Выражение !success равно true, если success равно false. Механизм работает отлично, так как success равно false лишь при неудачной попытке входа в систему. В таком случае вновь выполняется блок кода, ассоциированный с циклом, и мы снова запрашиваем у пользователя его имя и пароль.

Выражение !success равно false, если success равно true. Это тоже замечательно, так как если success равно true, то пользователь успешно входит в систему и цикл завершается.

Другой способ, позволяющий понять работу оператора !, — рассмотреть все возможные комбинации истинности и ложности (табл. 2.5).

Таблица 2.5. Возможные комбинации пользовательского имени и пароля, использующие логический оператор «НЕ»

security	!security
true	false
false	true

Понятие о порядке операций

Как и арифметические операторы, все логические операторы имеют уровни приоритета, влияющие на порядок, в котором интерпретируется (вычисляется) выражение. Логическое «НЕ», !, имеет более высокий приоритет, чем логическое «И», &&, а логическое «И», в свою очередь, — более высокий приоритет, чем логическое «ИЛИ», ||.

Как и в случае с арифметическими операторами, если вы хотите, чтобы сначала была выполнена операция с более низким приоритетом, заключите ее в круглые скобки. Можно составлять сложные выражения, в которых одновременно задействуются арифметические, реляционные и логические операторы. Приоритет операторов определяет точный порядок, в котором интерпретируются элементы выражения. Но лучше писать простые и ясные выражения, а не создавать тайнопись, для расшифровки которой требуется в совершенстве разбираться в приоритете операторов.

Полный список операторов C++ с указанием их значений приоритета дан в приложении 2 «Приоритет операторов языка C++».

СОВЕТ

Итак, в большом выражении можно использовать круглые скобки, чтобы изменять порядок интерпретации его частей, но также в таком выражении можно ставить избыточные скобки, которые не меняют значение выражения, а просто приводят выражение в более удобочитаемый вид. Приведу пример. Рассмотрим следующее выражение из программы Designers Network:

```
(username == "S.Meier" &&password == "civilization")
```

А вот аналогичное выражение с избыточными скобками:

```
( (username == "S.Meier") && (password == "civilization") )
```

Дополнительные скобки не меняют значение выражения, зато отлично подчеркивают, что здесь имеется два небольших выражения, объединенных оператором &&.

Использование избыточных скобок — это своеобразная красивость. Полезны ли они или действительно просто избыточны? Решать программисту, то есть вам.

Генерирование случайных чисел

Если в игре есть элемент непредсказуемости, она становится только интереснее. Например, компьютерный соперник в игре типа RTS (стратегия в реальном времени) неожиданно изменит стратегию либо в игре типа FPS (шутер от первого лица) из неприметной двери выскочит страшное чудовище. Игроки обожают подобные сюрпризы. Один из способов их реализации в программе связан с генерированием случайных чисел.

Знакомство с программой Die Roller

Программа Die Roller имитирует бросок шестигранной игральной кости. Компьютер получает выпавшее значение, генерируя случайное число. Результат выполнения этой программы показан на рис. 2.12.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `die_roller.cpp`.

```
// Программа Die Roller
// Демонстрирует генерирование случайных чисел
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand(static_cast<unsigned int>(time(0)));
    // запускаем генератор случайных чисел
    int randomNumber = rand(); // генерируем случайное число
```

```
    int die = (randomNumber % 6) + 1; // получаем число между 1 и 6
    cout << "You rolled a " << die << endl;
    return 0;
}
```

Рис. 2.12. Результат броска кости зависит от того, какое случайное число генерирует программа (опубликовано с разрешения компании Microsoft)

Вызов функции rand()

В самом начале программы я включаю в нее новый файл:

```
#include<cstdlib>
```

Файл `cstdlib` среди прочего содержит функции, отвечающие за генерирование случайных чисел. Поскольку я включил этот файл в код, то могу свободно использовать все функции из него, в частности функцию `rand()`, — именно это я и делаю в функции `main()`:

```
int randomNumber = rand(); // генерируем случайное число
```

В главе 1 мы изучили, что функция — это блок кода, который срабатывает и возвращает значение. Для вызова функции пишется ее имя, за которым следует пара круглых скобок. Если функция вернет значение, то это значение можно присвоить переменной. Именно это я и делаю с помощью оператора присваивания. Значение, возвращенное функцией `rand()` (то есть случайное число), я присваиваю переменной `randomNumber`.

СОВЕТ

Функция rand() генерирует случайное число в диапазоне от 0 до как минимум 32 767. Точный верхний предел зависит от реализации языка C. Значение верхнего предела хранится в константе RAND_MAX, определяемой в библиотеке cstdlib. Итак, если вы хотите узнать, какое максимальное случайное число может сгенерировать функция rand(), просто пошлите константу RAND_MAX в cout.

Функции могут принимать и такие значения, которые затем будут использовать в ходе работы. Вы указываете эти значения в круглых скобках после имени функции, разделяя значения запятыми. Эти значения называются *аргументами* функции. Указывая их, вы *передаете (сообщаете)* их функции. Я не передаю никакого значения функции `rand()`, так как она не принимает аргументов.

Посев генератора случайных чисел

Компьютеры генерируют *псевдослучайные числа*, то есть числа, не являющиеся абсолютно случайными, руководствуясь при этом специальной формулой. Можно сказать, что компьютер читает значения в огромной книге заранее подготовленных чисел. При этом создается впечатление, что компьютер выдает последовательность абсолютно случайных чисел.

Однако здесь возникает проблема: дело в том, что компьютер всегда начинает читать эту книгу с начала. Поэтому он выдает в программе все время одну и ту же серию «случайных» чисел. В играх нас это не устраивает. Например, мы бы не хотели получать одинаковые последовательности бросков кости в игре «Крепс».

Чтобы устранить эту проблему, запрограммируем компьютер так: пусть он начинает «читать» книгу чисел с произвольного места. Указание этого места называется *посевом* генератора случайных чисел. Разработчики игр дают генератору случайных чисел исходное число, называемое *зерном*. Зерно соответствует начальной позиции в последовательности псевдослучайных чисел.

Следующий код выполняет посев генератора случайных чисел:

```
srand(static_cast<unsigned int>(time(0)));
// посев генератора случайных чисел
```

Да, строка выглядит довольно мудрено, но ничего сложного тут не происходит. Строка выполняет посев генератора случайных чисел, исходя из актуальных даты и времени. Это нам подходит, поскольку при каждом запуске программы значения даты и времени будут различаться.

Вот что происходит на уровне кода: функция `srand()` сеет генератор случайных чисел, вы должны просто передать ей число типа `unsigned int` в качестве зерна. Здесь таким зерном для этой функции является возвращаемое значение функции `time(0)` — число, зависящее от текущих значений даты и времени. Код `static_cast<unsigned int>` просто преобразует *(приводит)* это значение к типу `unsigned int`. Пока вы, вероятно, не понимаете всех нюансов этой строки. Остановимся на том, что если мы хотим генерировать при каждом прогоне программы новую последовательность случайных чисел, то программа должна выполнять этот код прежде, чем вызывать функцию `rand()`.

СОВЕТ

Подробное описание процедур приведения значений одних типов к другим выходит за рамки этой книги.

Расчет числа в заданном диапазоне

Сгенерировав случайное число, мы записываем в переменную randomNumber числовое значение из диапазона от 0 до 32 767 (именно таковы верхний и нижний пределы диапазона целых чисел в той реализации языка C++, с которой я работаю). Но мне требуется число в диапазоне от 1 до 6, поэтому далее я применяю оператор остатка по модулю, чтобы получить число именно из этого диапазона:

```
int die = (randomNumber % 6) + 1;
// получаем число в диапазоне от 1 до 6
```

Любое положительное число, деленное на 6, даст остаток в диапазоне от 0 до 5. В приведенной строке кода я беру этот остаток и прибавляю к нему 1. В результате получаю положительное число в диапазоне от 1 до 6 — именно этого мы и добивались. Можете пользоваться таким приемом для преобразования случайного числа в число из нужного вам диапазона.

ОСТОРОЖНО!

Применяя оператор деления по модулю для преобразования случайного числа в число из заданного диапазона, мы не всегда будем получать равномерный разброс значений. Некоторые числа из диапазона могут встречаться чаще, нежели другие. Правда, в простых играх данная проблема несущественна.

Понятие об игровом цикле

Игровой цикл — это обобщенное представление потока событий, происходящих в игре. В сумме эти события повторяются, именно поэтому мы и говорим здесь о цикле. Хотя реализация цикла от игры к игре может существенно различаться, фундаментальная структура цикла является практически одинаковой во всех играх любых жанров. Идет ли речь о тривиальном космическом шутере или о сложной ролевой игре (RPG), игровой процесс обычно можно разбить на одинаковые повторяющиеся компоненты, образующие игровой цикл. На рис. 2.13 дано наглядное представление такого цикла.

Вот описание всех этапов игрового цикла.

- **Установка параметров.** На данном этапе игра принимает исходные параметры или загружает игровые ресурсы, например звуки, музыку и графику. Кроме того, пользователю могут быть изложены сюжет игры и ее цели.
- **Получение пользовательского ввода.** Программа принимает пользовательский ввод через то или иное устройство ввода: клавиатуру, мышь, джойстик, трекбол т. п.
- **Обновление внутреннего состояния игры.** В игровом мире срабатывают игровая логика и правила, при этом учитывается полученный ранее пользователь-

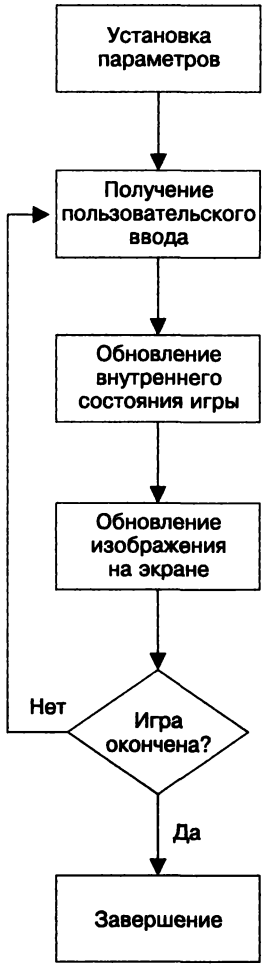


Рис. 2.13. Игровой цикл в упрощенном виде описывает поток событий, которые разворачиваются практически в любой игре

ский ввод. В частности, речь может идти о системе игровой физики, определяющей правила взаимодействия объектов на экране, либо о вычислениях, связанных с искусственным интеллектом виртуального врага.

- **Обновление изображения на экране.** В большинстве компьютерных игр именно этот процесс является наиболее затратным для аппаратного обеспечения, поскольку связан с отрисовкой графики. Однако этот процесс может быть и тривиальным — сводиться к отображению текста.
- **Проверка того, не окончена ли игра.** Если игра не окончена (персонаж еще жив или пользователь пока не справился с задачей), процесс управления вновь откатывается на этап пользовательского ввода. Если игра окончена, цикл переходит к завершающей стадии.

- **Завершение.** На этом этапе игра окончена. Зачастую пользователю выдается какая-нибудь итоговая информация, например набранные им очки. При необходимости программа высвобождает все ресурсы и завершается.

Знакомство с игрой Guess My Number

Заключительный проект этой главы называется Guess My Number. Это классическая игра по угадыванию чисел. Для тех, у кого не было в детстве такой забавы, поясняю: компьютер выбирает случайное число, допустим, в диапазоне от 1 до 100, а игрок пытается угадать это число с наименьшего количества попыток. Всякий раз, когда пользователь вводит вариант, компьютер сообщает ему, насколько близко это число расположено к загаданному: слишком много, слишком мало или «почти-почти». Как только пользователь угадает число, игра закончится. На рис. 2.14 игра Guess My Number показана в действии. Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 2, имя файла — `guess_my_number.cpp`.

Рис. 2.14. Я угадал число всего с трех попыток
(опубликовано с разрешения компании Microsoft)

Применение игрового цикла

Даже такую простую игру можно рассмотреть через призму игрового цикла. На рис. 2.15 показано, как красиво парадигма игрового цикла обрамляет ход игры.

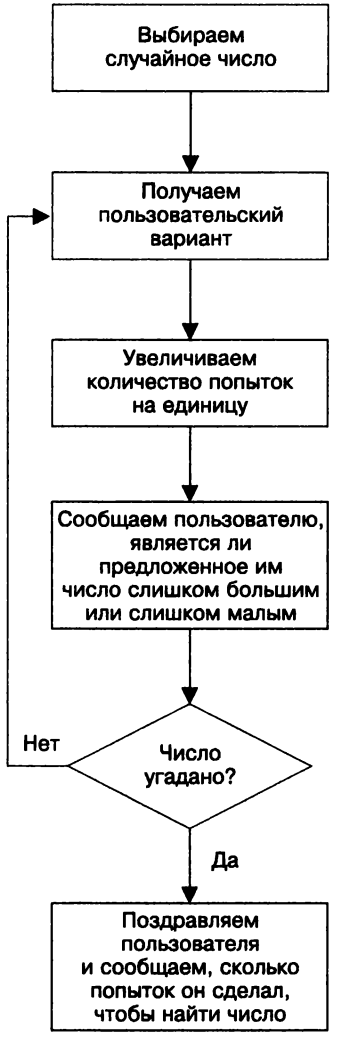


Рис. 2.15. Игровой цикл, реализуемый в игре Guess My Number

Установка игровых параметров

Как всегда, я начинаю код с пары комментариев и включаю необходимые файлы:

```
// Игра Guess My Number
// Классическая игра в угадывание чисел
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
```

Я включаю файл cstdlib, так как собираюсь сгенерировать случайное число. Включаю файл ctime, так как хочу посеять генератор случайных чисел, использовав в качестве зерна текущее значение времени.

Далее я запускаю функцию main(), выбирая случайное число, устанавливая количество использованных попыток в 0 и создавая переменную для хранения пользовательских вариантов:

```
int main()
{
    srand(static_cast<unsigned int>(time(0)));  // запускаем генератор случайных чисел
    intsecretNumber = rand() % 100 + 1;  // случайное число в диапазоне от 1 до 100
    int tries = 0;
    int guess;
    cout << "\tWelcome to Guess My Number\n\n";
}
```

Создание игрового цикла

Далее напишу игровой цикл:

```
    do
    {
        cout << "Enter a guess: ";
        cin >> guess;
        ++tries;
        if (guess > secretNumber)
        {
            cout << "Too high!\n\n";
        }
        else if (guess < secretNumber)
        {
            cout << "Too low!\n\n";
        }
        else
        {
            cout << "\nThat's it! You got it in " << tries << " guesses!\n";
        }
    } while (guess != secretNumber);
```

Я получаю пользовательский вариант, увеличиваю на единицу количество сделанных попыток, затем сообщаю игроку, был ли его вариант слишком мал, слишком велик или почти точен. Если пользователь угадал число, игра прекращается. Обратите внимание: инструкции if вложены в цикл while.

Завершение игры

Как только пользователь найдет загаданное число, цикл и сама игра завершаются. Остается закончить программу.

```
    return 0;
}
```

Резюме

Из этой главы вы должны усвоить следующее.

- Как пользоваться истинностью и ложностью выражения для ветвления программы или пропуска отдельных блоков кода.
- Для обозначения истинности и ложности используются ключевые слова `true` и `false`.
- Можно интерпретировать любое значение или выражение на предмет его истинности или ложности.
- Любое ненулевое значение может быть интерпретировано как `true`, а 0 — как `false`.
- Для того чтобы выражение давало в результате `true` или `false`, как правило, требуется сравнить значения с помощью реляционных операторов.
- Инструкция `if` проверяет выражение и выполняет блок кода лишь в том случае, если выражение дает в результате `true`.
- Условие `else` в инструкции `if` содержит код, который должен быть выполнен лишь в том случае, когда выражение, проверенное инструкцией `if`, дает в результате `false`.
- Инструкция `switch` проверяет значение, которое может быть приведено к типу `int`, и выполняет блок кода, ассоциированный с соответствующим значением.
- В инструкции `switch` может использоваться ключевое слово `default`. С помощью этого ключевого слова указывается код, который следует выполнить, если значение, проверенное в инструкции `switch`, не совпадает ни с одной из перечисленных величин.
- Цикл `while` выполняет блок кода, если выражение равно `true`, и повторяет этот код до тех пор, пока выражение остается равным `true`.
- Цикл `do` выполняет блок кода, а если после этого выражение дает в результате `true`, то повторно выполняет этот блок кода и т. д.
- Инструкция `break` применяется в цикле для того, чтобы программа могла немедленно выйти из цикла.
- Инструкция `continue`, используемая в цикле, автоматически возвращает выполнение кода к началу цикла.
- Оператор `&&` («И») объединяет два простых выражения в более сложное, которое считается истинным, лишь если оба выражения в его составе дают в результате `true`.
- Оператор `||` («ИЛИ») объединяет два простых выражения в более сложное, которое считается истинным, если любое выражение в его составе дает в результате `true`.
- Оператор `!` («НЕ») создает новое выражение, которое является противоположным по признаку истинности/ложности исходному выражению.
- Игровой цикл — это обобщенное представление потока событий, происходящих в игре, причем основная часть этих событий повторяется.

- В файле `cstdlib` содержатся функции, предназначенные для генерирования случайных чисел.
- Функция `srand()`, определенная в файле `cstdlib`, выполняет посев генератора случайных чисел.
- Функция `rand()`, определенная в файле `cstdlib`, возвращает случайное число.

Вопросы и ответы

1. *Обязательно ли мне использовать ключевые слова true и false?*

   Нет, но лучше использовать. До того как были изобретены ключевые слова `true` и `false`, программисты обычно использовали 1 для представления истинности и 0 для представления лжи. Но теперь, когда есть ключевые слова `true` и `false`, удобнее работать именно с ними, а не с 0 и 1.

2. *Можно ли присвоить переменной типа bool какое-либо значение, кроме true и false?*

   Да. Переменной типа `bool` можно присвоить выражение, в таком случае в этой переменной будет храниться информация об истинности или ложности данного выражения.

3. *Можно ли использовать инструкцию switch для проверки каких-либо выражений, кроме целочисленных?*

   Нет. Инструкции `switch` работают только с такими значениями, которые могут быть приведены к целочисленным (таковы, в частности, значения типа `char`).

4. *Как проверить конкретное нецелочисленное значение относительно множества других значений, если инструкция switch в данном случае неприменима?*

   Можно использовать серию инструкций `if`.

5. *Что такое бесконечный цикл?*

   Это цикл, который по определению не прекращается, независимо от пользовательского ввода.

6. *Почему бесконечные циклы считаются порочной практикой?*

   Потому что программа, попавшая в бесконечный цикл, не может закончиться без вмешательства человека. Ее приходится завершать на уровне операционной системы. В худшем случае для этого придется выключить компьютер.

7. *Может ли компилятор идентифицировать бесконечный цикл и просигнализировать о нем как об ошибке?*

   Нет. Бесконечный цикл — это логическая ошибка, отследить которую должен сам программист.

8. *Если бесконечные циклы — порочная практика, почему не возбраняется писать циклы while (true)?*

   Если программист пишет цикл `while (true)`, то также должен предусмотреть способ завершения этого цикла (обычно с помощью инструкции `break`).

9. *Почему некоторые специалисты считают, что выход из цикла с помощью инструкции break — это неграмотное решение?*

   Потому что при злоупотреблении инструкциями `break` сложнее понять, при каких условиях завершается цикл. Тем не менее в некоторых случаях цикл `while (true)` с применением `break` может быть более внятной конструкцией, чем классический вариант написания подобного цикла.

10. *Что такое псевдослучайное число?*

    Как правило, псевдослучайное число генерируется по формуле. Поэтому серия псевдослучайных чисел уже не является по-настоящему случайной, но на практике этим обычно можно пренебречь.

11. *Что такое посев генератора случайных чисел?*

    Это операция, при которой генератор случайных чисел получает исходное число (так называемое зерно), например целое число, от которого затем отталкивается при генерировании новых чисел. Если не посеять генератор случайных чисел, то он при каждом запуске программы будет выдавать одинаковые серии значений.

12. *Всегда ли нужно выполнять посев генератора случайных чисел перед тем, как его использовать?*

    Нет. Возможно, вам как раз понадобится, чтобы программа при каждом запуске выдавала одну и ту же числовую последовательность — например, при тестировании.

13. *Как генерировать по-настоящему случайные числа?*

    Существуют сторонние библиотеки, позволяющие генерировать более произвольные случайные числа, нежели те библиотеки, которые обычно поставляются с компиляторами C++.

14. *Во всех ли играх используется игровой цикл?*

    Игровой цикл — это один из способов представления событий, протекающих в игре. Действительно, такая парадигма может работать в конкретной игре, но не любая игра обязательно реализуется как некая сумма кода, организованного в виде цикла.

Вопросы для обсуждения

1. Какие задачи было бы сложно решать в программе, не содержащей циклов?
2. Каковы достоинства и недостатки инструкции `switch` по сравнению с серией инструкций `if`?
3. Когда можно опустить инструкцию `break` в конце условия `case` в рамках инструкции `switch`?
4. Когда следует использовать именно цикл `while`, а не цикл `do`?
5. Опишите вашу любимую компьютерную игру как пример игрового цикла. Насколько выражены в ней такие циклические черты?

Упражнения

1. Перепишите программу Menu Chooser из этой главы, представив различные уровни сложности в виде перечисления. При этом продолжайте работать с переменной типа `int`.
2. Найдите ошибку в следующем цикле:

```
int x = 0;
while (x)
{
  ++x;
  cout << x << endl;
}
```

3. Напишите новую версию программы Guess My Number, где компьютер и игрок меняются ролями: игрок выбирает число, а компьютер должен его угадать.

3 Циклы for, строки и массивы. Игра «Словомеска»

В предыдущей главе мы научились работать с отдельными значениями, а в этой узнаем, как обращаться с последовательностями данных. Вы подробнее познакомитесь со строками — последовательностями символов. Кроме того, научитесь работать с последовательностями любых типов. В этой главе также будет рассмотрена еще одна разновидность циклов, идеально подходящая для работы с такими последовательностями. В частности, мы обсудим:

- применение циклов `for` для перебора последовательностей;
- использование объектов, содержащих как данные, так и функции;
- использование объектов `string` и их функций-членов экземпляра для работы с последовательностями символов;
- использование массивов для хранения последовательностей любого типа, обращения к этим последовательностям и манипулирования ими;
- использование многомерных массивов для более точного представления некоторых коллекций данных.

Использование циклов for

В главе 2 мы уже познакомились с циклом `while`. Теперь давайте рассмотрим другой цикл, который называется `for`. Подобно `while`, цикл `for` позволяет повторить блок кода, однако циклы `for` специально «заточены» для подсчета элементов последовательности с одновременным передвижением по этой последовательности. Примером такой последовательности элементов может быть содержимое рюкзака персонажа в ролевой игре.

Вот упрощенное представление цикла `for`:

```
for (initialization; test; action)
    statement;
```

`initialization` — это инструкция, задающая начальное условие для цикла. Например, она может устанавливать переменную счетчика в 0. Выражение `test` проверяется перед каждым прогоном цикла, так же как в цикле `while`. Если `test`

равно false, то программа переходит к первому выражению, следующему после цикла. Если test равно true, то программа выполняет statement. Далее совершается action (зачастую это действие сводится к увеличению значения переменной на единицу). Цикл повторяется до тех пор, пока test не станет равно false, в результате чего цикл завершится.

Знакомство с программой Counter

Программа Counter может считать вперед, назад и пятерками. Она даже может высчитать целую таблицу со строками и столбцами. Все это она делает с помощью циклов for. На рис. 3.1 эта программа показана в действии.

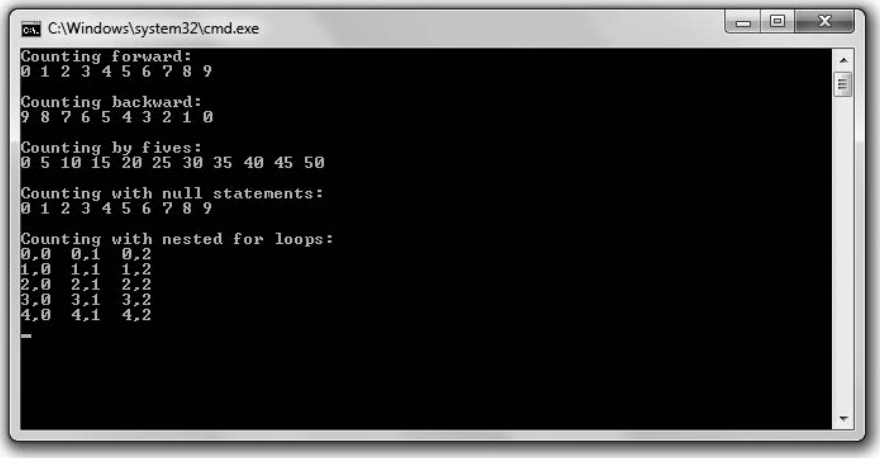


Рис. 3.1. Циклы for выполняют все подсчеты, а пара вложенных циклов for выводит сетку таблицы (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 3, имя файла — counter.cpp.

```
// Программа Counter
// Демонстрирует работу с циклами for
#include <iostream>
using namespace std;
int main()
{
    cout << "Counting forward:\n";
    for (int i = 0; i < 10; ++i)
    {
        cout << i << " ";
    }
    cout << "\n\nCounting backward:\n";
    for (int i = 9; i >= 0; --i)
    {
        cout<< i << " ";
```

```
    }
    cout << "\n\nCounting by fives:\n";
    for (int i = 0; i <= 50; i += 5)
    {
        cout << i << " ";
    }
    cout << "\n\nCounting with null statements:\n";
    int count = 0;
    for ( ; count < 10; )
    {
        cout << count << " ";
        ++count;
    }
    cout << "\n\nCounting with nested for loops:\n";
    const int ROWS = 5;
    const int COLUMNS = 3;
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << i << "," << j << " ";
        }
        cout << endl;
    }
    return 0;
}
```

ОСТОРОЖНО!

Если вы работаете со старым компилятором, который не полностью реализует современный стандарт C++, то при попытке скомпилировать эту программу можете получить ошибку примерно следующего содержания: error: 'i' : redefinition; multiple initialization.

Наилучшее решение подобной проблемы — работать с современным компилятором. Если вы работаете с Windows, то вам повезло: можно скачать популярную (бесплатную!) среду разработки Microsoft Visual Studio Express 2013 для Windows ПК, в которой есть современный компилятор. Чтобы скачать эту программу, перейдите по ссылке www.visualstudio.com/downloads/download-visual-studio-vs.

Если вы вынуждены работать со старым компилятором, то любые переменные счетчика в цикле for нужно объявлять в программе всего один раз, сразу для всех переменных, находящихся в области видимости. Концепция области видимости будет рассматриваться в главе 5.

Подсчеты с помощью циклов for

Первый цикл `for` считает от 0 до 9. Вот как начинается этот цикл:

```
for (int i = 0; i < 10; ++i)
```

В инструкции инициализации `int i = 0` объявляется переменная `i`, инициализируемая со значением 0. Выражение `i < 10` означает, что цикл будет продолжаться до тех пор, пока значение `i` не достигнет 10. Наконец, оператор действия `++i` указывает, что по завершении каждого прогона цикла значение `i` должно увеличиваться

на единицу. В результате имеем 10 *итераций* цикла — по одной для каждого значения от 0 до 9. В ходе каждой итерации тело цикла выводит на экран новое значение `i`.

Следующий цикл `for` считает от 9 до 0. Вот как начинается этот цикл:

```
for (int i = 9; i>= 0; --i)
```

Здесь переменная I инициализируется в значении 9, цикл продолжается до тех пор, пока значение I остается неотрицательным. После каждого выполнения цикла значение `i` уменьшается на единицу. В результате цикл отображает на экране значения от 9 до 0.

Следующий цикл считает пятерками от 0 до 50. Цикл начинается так:

```
for (int i = 0; I <= 50; i += 5)
```

Здесь переменная `i` инициализируется в значении `0`, цикл продолжается до тех пор, пока значение `i` остается меньшим или равным 50. Однако обратите внимание на оператор действия `i += 5`. Эта инструкция увеличивает значение `i` на 5 после каждого прогона цикла. Таким образом, цикл выводит на экран значения 0, 5, 10, 15 и т. д. Выражение `i <= 50` означает, что тело цикла должно выполняться до тех пор, пока значение `i` меньше или равно 50.

Можно инициализировать переменную счетчика, создать тестовое условие и обновлять переменную счетчика любыми значениями, какими хотите. Но, как правило, счетчик начинает работу с 0 и увеличивает его на 1 после каждой итерации цикла.

Наконец, не забывайте о подводных камнях, связанных с бесконечными циклами. Об этих проблемах мы уже говорили применительно к циклам `while`, однако они не менее актуальны и при работе с циклами `for`. Пишите такие циклы, которые легко завершаются, иначе очень огорчите многих геймеров.

Использование пустых инструкций в циклах for

При создании цикла `for` можно использовать пустые инструкции, как в следующей строке:

```
for ( ; count < 10; )
```

Инструкция инициализации и оператор действия здесь оставлены пустыми, это вполне допустимо. Я объявил и инициализировал переменную `count` до начала цикла, а уже в теле цикла увеличил ее значение на единицу. Здесь отображается та же последовательность чисел, что и в самом первом цикле программы. Хотя этот цикл на первый взгляд может показаться странным, он абсолютно корректен.

СОВЕТ

Программисты игр придерживаются разных традиций. В прошлой главе мы убедились, что можно написать такой цикл, который продолжает работу, пока не достигнет инструкции о выходе. Такова, например, инструкция break в цикле while(true). Некоторые программисты предпочитают писать циклы for, которые начинаются с инструкции for (;;). Поскольку тестовое выражение в таком цикле пустое, он будет продолжать работать до тех пор, пока не встретит инструкцию о выходе.

Вложение циклов for

Можно вкладывать циклы for друг в друга. Именно это я и делаю в следующем коде, который подсчитывает элементы таблицы. Внешний цикл начинается так:

```
for (int i = 0; i < ROWS; ++i)
```

Его тело (ROWS) просто выполняется пять раз. Но оказывается, что внутри этого цикла for есть еще один цикл for, который начинается так:

```
for (int j = 0; j < COLUMNS; ++j)
```

Таким образом, второй цикл успевает полностью выполниться в каждой из итераций первого цикла. Это означает, что внутренний цикл выполняет код COLUMNS по три раза на каждый прогон внешнего цикла ROWS, выполняемого пять раз. Итого цикл COLUMNS выполняется 15 раз. Вот что именно здесь происходит.

1. Внешний цикл for объявляет переменную i и инициализирует ее в значении 0. Поскольку i меньше значения ROWS (5), программа входит в тело внешнего цикла.
2. Внутренний цикл объявляет переменную j и инициализирует ее в значении 0. Поскольку j меньше значения COLUMNS (3), программа входит в тело внутреннего цикла, посылая значения I и j в cout, в результате чего через стандартный вывод на экран попадают цифры 0, 0.
3. Программа достигает конца внутреннего цикла и увеличивает значение j до 1. Поскольку j по-прежнему меньше значения COLUMNS (3), программа вновь выполняет тело внутреннего цикла, отображая на экране 0, 1.
4. Программа достигает конца внутреннего цикла и увеличивает значение j до 2. Поскольку j по-прежнему меньше значения COLUMNS (3), программа вновь выполняет тело внутреннего цикла, отображая на экране 0, 2.
5. Программа достигает конца тела внутреннего цикла и увеличивает значение j до 3.
6. Программа завершает первую итерацию внешнего цикла, посылая endl в cout и заканчивая обработку первой строки таблицы.
7. Программа достигает конца тела внешнего цикла и увеличивает значение переменной I на единицу. Поскольку i меньше, чем ROWS(5), программа вновь переходит к телу внешнего цикла.
8. Программа достигает внутреннего цикла, который вновь выполняется с самого начала, а именно с объявления переменной j и инициализации ее в значении 0. Программа повторяет все действия, описанные на этапах 2–7, отображая вторую строку таблицы. Далее этот процесс продолжается до тех пор, пока на экран не будут выведены все строки таблицы.

Еще раз напоминаю о важной детали: внутренний цикл выполняется целиком в каждой итерации внешнего цикла.

Понятие об объектах

До сих пор мы говорили о том, как сохранять отдельные информационные объекты в переменных и как манипулировать этими переменными с помощью операторов и функций. Но большинство сущностей, которые вам придется представлять в компьютерной игре, — скажем, инопланетный космический корабль — являются объектами. Объекты — это цельные неоднородные сущности, обладающие определенными свойствами (например, уровнем энергии) и возможностями (допустим, могут стрелять из пушек). Зачастую при обсуждении объекта целесообразно рассматривать его свойства отдельно от его возможностей.

К счастью, в большинстве современных языков программирования можно работать с программными объектами (которые зачастую называются просто *объектами*), в которых комбинируются данные и функции. Элемент данных в составе объекта называется *«член данных»*, а функция объекта называется *«функция-член»*. Давайте в качестве конкретного примера рассмотрим инопланетный космический корабль. Этот объект может относиться к новому типу `Spacecraft`, определяемому программистом, работающим над данной игрой. В объекте такого типа могут быть член данных, соответствующий уровню энергии, а также функция-член, реализующая стрельбу из пушек. На практике уровень энергии корабля может сохраняться в члене данных `energy` как переменная типа `int`, а возможность стрельбы из пушек может быть определена в функции-члене `fireWeapons()`.

Все объекты, относящиеся к одному и тому же базовому типу, обладают схожей структурой. Иными словами, все объекты одного типа будут содержать схожие наборы членов данных и функций-членов. Правда, конкретные значения членов данных будут различаться от объекта к объекту. Так, если у вас есть эскадра инопланетных космических кораблей, то каждое такое судно будет обладать своим уровнем энергии. Например, у одного корабля это значение может равняться 75, а у другого — всего 10 и т. д. Даже если у двух кораблей уровень энергии одинаков, каждое из этих одинаковых значений будет относиться к своему уникальному кораблю. Кроме того, каждое судно может палить из собственных бортовых пушек, вызывая свою функцию `fireWeapons()`. На рис. 3.2 проиллюстрирована модель инопланетного космического корабля.

Самая классная черта объектов заключается в том, что работать с ними вы можете, не зная деталей их реализации. Так, вполне можно водить автомобиль, не разбираясь в тонкостях его конструкции. Необходимо знать лишь члены данных объекта и его функции-члены. Продолжая пример с автомобилем: чтобы управлять машиной, достаточно знать, где у нее находится руль, педаль газа и педаль тормоза.

Объекты можно хранить в переменных так же, как встроенные типы. Следовательно, объект, соответствующий космическому кораблю пришельцев, можно сохранить в переменной типа `Spacecraft`. Можно обращаться к членам данных и функциям-членам с помощью оператора доступа к члену (`.`). Этот оператор ставится после имени той переменной, которая соответствует объекту. Итак, если вы

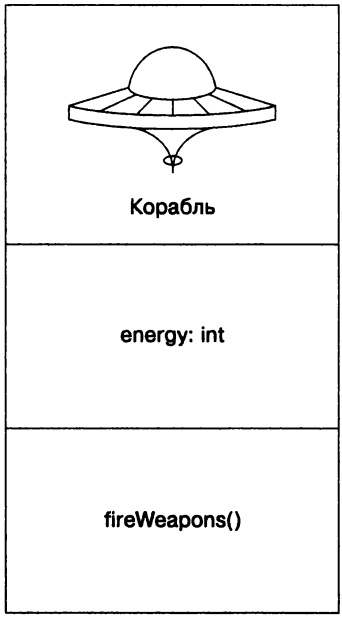


Рис. 3.2. В данной модели представления инопланетного космического корабля указано, что объект будет иметь член данных energy и функцию-член fireWeapons()

хотите, чтобы инопланетный космический корабль `ship` палил из пушек, только если его уровень энергии превышает 10, то можно написать такой код:

```
// ship – это объект типа Spacecraft
if (ship.energy > 10)
{
    ship.fireWeapons()
}
```

Код `ship.energy` обращается к члену данных `energy` этого объекта, а метод `ship.fireWeapons()` вызывает функцию-член `fireWeapons()` этого объекта.

Хотя на данном этапе вы еще не умеете создавать собственные новые типы (например, тип данных для инопланетного космического корабля), вполне можно работать с заранее определенными типами объектов. Именно этим мы сейчас и займемся.

Работа со строковыми объектами

В главе 1 мы уже вкратце упоминали об объектах `string`, которые исключительно удобны при работе с последовательностями символов независимо от того, пишете ли вы полнофункциональную игру-головоломку или просто сохраняете имя пользователя. Строка `string` — это, в сущности, объект. Он предоставляет собственный

набор функций-членов. Эти функции позволяют выполнять над объектом `string` ряд операций, от обычного получения длины конкретной строки до выполнения сложной подстановки символов. Кроме того, строки определяются именно так, что с ними можно легко и интуитивно понятно использовать операторы, с которыми мы уже успели познакомиться.

Знакомство с программой String Tester

Программа String Tester использует объект `string`, равный `"Game Over!!!"`, и сообщает вам длину строки, индекс (номер позиции) каждого символа, а также указывает, содержатся ли в рассматриваемой строке те или иные подстроки. Кроме того, эта программа стирает фрагменты объекта `string`. На рис. 3.3 показан результат выполнения этой программы.

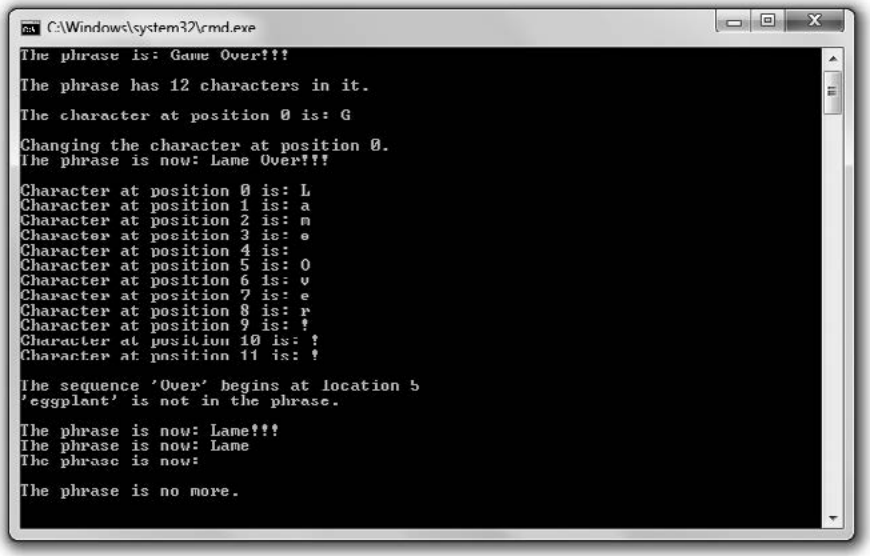


Рис. 3.3. Объекты string комбинируются, изменяются и стираются с помощью уже знакомых нам операторов и функций-членов объекта string (опубликовано с разрешения компании Microsoft)

Код программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 3, имя файла — `string_tester.cpp`.

```
// Программа String Tester
// Демонстрирует работу со строковыми объектами
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string word1 = "Game";
    string word2("Over");
```

```
    string word3(3, '!');
    string phrase = word1 + " " + word2 + word3;
    cout << "The phrase is: " << phrase << "\n\n";
    cout << "The phrase has " << phrase.size() << " characters in it.\n\n";
    cout << "The character at position 0 is: " << phrase[0] << "\n\n";
    cout << "Changing the character at position 0.\n";
    phrase[0] = 'L';
    cout << "The phrase is now: " <<phrase << "\n\n";
    for (unsigned int i = 0; i < phrase.size(); ++i)
    {
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
    }
    cout << "\nThe sequence 'Over' begins at location ";
    cout << phrase.find("Over") << endl;
    if (phrase.find("eggplant") == string::npos)
    {
        cout << "'eggplant' is not in the phrase.\n\n";
    }
    phrase.erase(4, 5);
    cout << "The phrase is now: " << phrase << endl;
    phrase.erase(4);
    cout << "The phrase is now: " << phrase << endl;
    phrase.erase();
    cout << "The phrase is now: " << phrase << endl;
    if (phrase.empty())
    {
        cout << "\nThe phrase is no more.\n";
    }
    return 0;
}
```

Создание строковых объектов

В функции `main()` я первым делом создаю три строки тремя разными способами:

```
string word1 = "Game";
string word2("Over");
stringword3(3, '!');
```

В первой из этих трех строк я просто создаю строковый объект `word1` с помощью оператора присваивания. Оператор присваивания здесь используется точно так же, как с любыми другими переменными. В результате объект `word1` становится равен `"Game"`.

Далее я создаю объект `word2`, ставя в круглых скобках тот строковый объект, который хочу задать в качестве значения моей переменной. В результате объект `word2` становится равен `"Over"`.

Наконец, я создаю объект `word3`, записывая в круглых скобках число и один символ. В результате получается строковый объект, состоящий из нескольких экземпляров указанного в скобках символа, причем количество экземпляров равно числу, указанному в этих же скобках. В результате объект `word3` становится равен `"!!!"`.

Конкатенация строковых объектов

Далее я создаю новый строковый объект, который получаю путем конкатенации (сцепления) трех приведенных ранее строковых объектов:

```
string phrase = word1 + " " + word2 + word3;
```

В результате получаю фразу «Game Over!!!».

Обратите внимание: оператор +, который мы ранее использовали только с числами, работает и с объектами string. Дело в том, что оператор + является *перегруженным*. Возможно, на первый взгляд термин «перегруженный» покажется вам негативно окрашенным, как будто такой оператор вот-вот лопнет! Однако на самом деле все нормально. При перегрузке обычный оператор переопределяется так, что в специфическом контексте, который ранее не имел специального определения, он начинает работать нетипичным для себя образом. В данном случае я использую оператор + не для сложения чисел, а для сцепления объектов string. Я могу совершить здесь такую операцию лишь потому, что тип string специально предусматривает именно такую перегрузку оператора + в следующей формулировке: при использовании со строками оператор + означает конкатенацию объектов string.

Использование функции-члена size()

Итак, давайте рассмотрим функцию-член объекта string. Далее приведен пример работы с функцией-членом size():

```
cout << "The phrase has " << phrase.size() << " characters in it.\n\n";
```

Код phrase.size() вызывает функцию-член size() строкового объекта phrase с помощью оператора доступа к члену . (точки). Функция-член size() просто возвращает беззнаковое целочисленное значение, соответствующее размеру объекта string, то есть количеству символов, содержащихся в этой строке. Поскольку объект string равен "Game Over!!!", функция-член возвращает 12 (учитываются все символы, в том числе пробелы). Разумеется, вызвав функцию size() для другого строкового объекта, можно получить иной результат, так как итоговое число зависит от количества символов в объекте string.

СОВЕТ

Строковые объекты также имеют функцию-член length(). Она, как и функция size(), возвращает количество символов в объекте string.

Индексация объекта string

В объекте string хранится последовательность значений char. Можно обратиться к любому отдельно взятому объекту char, указав его индексный номер с помощью оператора индексации ([]). Именно это я и делаю далее:

```
cout << "The character at position 0 is: " << phrase[0] << "\n\n";
```

Первый элемент в последовательности находится на позиции 0. В предыдущей инструкции код `phrase[0]` соответствует символу G. Поскольку отсчет начинается с 0, последний символ объекта `string` обозначается как `phrase[11]`, хотя строка и состоит из 12 символов.

ОСТОРОЖНО!

Многие забывают, что в таких случаях отсчет начинается с 0. Не забывайте, что, если объект string состоит из n символов, можно проиндексировать от 0 до позиции n – 1.

Оператор индексации не только позволяет обращаться к отдельным символам в объекте `string`, но и обеспечивает присваивание новых значений этим символам взамен уже имеющихся. Такая операция происходит в следующем коде:

```
phrase[0] = 'L';
```

Я меняю первый символ в объекте `phrase` на L, поэтому фраза приобретает вид «`Lame Over!!!`».

ОСТОРОЖНО!

Компиляторы C++ не выполняют проверку границ, когда работают с объектами string и оператором индексации. Таким образом, компилятор не проверяет, не пытаетесь ли вы обратиться к несуществующему элементу. Попытка запросить несуществующий элемент последовательности может привести к катастрофическим результатам, поскольку в таком случае можно затереть критически важные данные в памяти компьютера. В результате вся программа может аварийно завершиться, поэтому будьте осторожны при работе с оператором индексации.

Перебор символов в объектах String

Учитывая, что мы уже знаем о циклах `for` и объектах `string`, нам не составит труда перебрать символы, входящие в состав строки. Именно это я делаю далее:

```
for (unsigned int i = 0; i < phrase.size(); ++i)
{
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
}
```

Этот цикл перебирает все существующие позиции объекта `phrase`. Он начинает работу с 0 и доходит до 11. При каждой итерации на экран выводится один из символов, входящих в строку, это делается с помощью кода `phrase[i]`. Обратите внимание: переменная цикла `i` у меня является беззнаковым целым (относится к типу `unsigned int`), поскольку функция `size()` возвращает значения именно такого типа.

НА ПРАКТИКЕ

Перебор последовательностей — мощный прием, очень распространенный в программировании игр. Например, можно перебрать сотни отдельных юнитов в стратегической игре, обновить их статус и порядок. Можно также перебрать список вершин в трехмерной модели и применить к ней какую-либо геометрическую трансформацию.

Использование функции-члена find()

Далее я использую функцию-член `find()`, чтобы проверить, содержится ли в объекте `phrase` любой из двух строковых литералов. Сначала я ищу строковый литерал `"Over"`:

```
cout << "\nThe sequence 'Over' begins at location ";
cout << phrase.find("Over") << endl;
```

Функция-член `find()` ищет в вызвавшем ее объекте `string` ту строку, которая была указана в качестве аргумента данной функции. Функция-член возвращает номер той позиции, на которой в исследуемой строке начинается первый экземпляр искомой подстроки. Таким образом, код `phrase.find("Over")` возвращает номер позиции, с которой во фразе `phrase` начинается первый экземпляр `"Over"`. Поскольку мы работаем с фразой `"Lame Over!!!"`, функция `find()` возвращает 5. Как вы помните, отсчет символов начинается с нуля, поэтому 5 означает шестой символ.

Но что делать, если искомая подстрока отсутствует в строке, вызвавшей функцию? Такая ситуация рассмотрена далее:

```
if (phrase.find("eggplant") == string::npos)
{
    cout << "'eggplant' is not in the phrase.\n\n";
}
```

Поскольку в объекте `phrase` отсутствует строка `"eggplant"`, функция `find()` возвращает особую константу, определенную в файле `string`. Я обращаюсь к этой константе с помощью кода `string::npos`. В результате на экран выводится сообщение `'eggplant' is not in the phrase`.

Константа, к которой я обращаюсь с помощью `string::npos`, представляет собой максимальный возможный размер объекта `string`. Таким образом, это значение превышает любое возможное допустимое значение позиции в объекте `string`. В сущности, здесь используется номер позиции, которого не может быть. Подобное возвращаемое значение отлично демонстрирует, что одну строку не удается найти в другой.

СОВЕТ

При использовании функции find() ее можно снабдить необязательным аргументом, указывающим номер того символа, с которого программа должна начинать поиск подстроки. Так, в следующем коде мы начинаем искать строковый литерал "eggplant", начиная с позиции 5 в строковом объекте phrase.

```
location = phrase.find("eggplant", 5);
```

Использование функции-члена erase()

Функция-член `erase()` удаляет указанную подстроку из объекта `string`. Один из вариантов вызова такой функции-члена — задать начальную позицию на удаление и длину подстроки, что я и делаю в следующем коде:

```
phrase.erase(4, 5);
```

В предыдущей строке удаляется пятисимвольная подстрока, начинающаяся с позиции 4. Поскольку объект phrase равен "Lame Over!!!", данная функция-член удаляет подстроку Over и вся фраза принимает вид "Lame!!!".

Мы также можем вызвать функцию erase(), указав только начальную позицию подстроки. В результате удалим все символы, начиная с этой позиции вплоть до конца объекта string. Вот как это делается:

```
phrase.erase(4);
```

Эта строка удаляет все символы объекта string, начиная с позиции 4. Поскольку объект phrase равен "Lame!!!", функция-член удаляет подстроку !!!, после чего фраза превращается в "Lame".

Наконец, функцию erase() можно вызывать просто без всяких аргументов, вот так:

```
phrase.erase();
```

Этот код стирает все символы в объекте phrase. Таким образом, phrase превращается в пустую строку, равную "".

Использование функции-члена empty()

Функция-член empty() возвращает значение типа bool. Оно равно true, если объект string пуст, в противном случае возвращает false. Функция-член empty() используется в следующем коде:

```
if (phrase.empty())
{
    cout << "\nThe phrase is no more.\n";
}
```

Поскольку объект phrase равен пустой строке, метод phrase().empty возвращает true и на экран выводится сообщение The phrase is no more.

Работа с массивами

Тогда как объекты string очень удобны для работы с последовательностями символов, массивы позволяют оперировать элементами любых типов. Таким образом, массив вполне можно использовать, скажем, для хранения последовательности целых чисел, образующих список игровых рекордов. Но в массиве можно хранить и такие элементы, типы которых определяет сам программист, например набор артефактов, которые носит с собой персонаж ролевой игры.

Знакомство с программой Hero's Inventory

Программа Hero's Inventory ведет учет снаряжения героя из типичной ролевой игры. По законам жанра наш герой родился в маленькой затерянной деревушке, а его отец погиб от руки злобного воителя. В самом деле, какая сага без такой завязки? Теперь герой возмужал, ему пришло время отправиться в путь и отомстить.

В данной программе элементы снаряжения нашего героя организованы в виде массива. Массив — это последовательность объектов string, по одному такому объекту на каждый артефакт, которым владеет герой. Герой может покупать и даже находить новые вещи. На рис. 3.4 эта программа показана в действии.

Рис. 3.4. Снаряжение героя — это последовательность объектов string, сохраненных в массиве (опубликовано с разрешения компании Microsoft)

Код программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 3, имя файла — heros_inventory.cpp.

```
// Программа Hero's Inventory
// Демонстрирует работу с массивами
#include <iostream>
#include <string>
using namespace std;
int main()
{
    const int MAX_ITEMS = 10;
    string inventory[MAX_ITEMS];
    int numItems = 0;
    inventory[numItems++] = "sword";
    inventory[numItems++] = "armor";
    inventory[numItems++] = "shield";
    cout << "Your items:\n";
    for (int i = 0; i < numItems; ++i)
    {
    cout << inventory[i] << endl;
    }
    cout << "\nYou trade your sword for a battle axe.";
    inventory[0] = "battle axe";
    cout << "\nYour items:\n";
    for (int i = 0; i < numItems; ++i)
    {
        cout << inventory[i] << endl;
    }
```

```
    cout << "\nThe item name '" << inventory[0] << "' has ";
    cout << inventory[0].size() << " letters in it.\n";
    cout << "\nYou find a healing potion.";
    if (numItems < MAX_ITEMS)
    {
        inventory[numItems++] = "healing potion";
    }
    else
    {
        cout << "You have too many items and can't carry another.";
    }
    cout << "\nYour items:\n";
    for (int i = 0; i < numItems; ++i)
    {
        cout <<inventory[i] << endl;
    }
    return 0;
}
```

Создание массивов

Зачастую целесообразно определить константу, соответствующую количеству элементов в массиве. Именно такую роль в моей программе играет величина MAX_ITEMS, соответствующая максимальному количеству вещей, которые может нести с собой герой:

```
const int MAX_ITEMS = 10;
```

Массив определяется примерно так же, как и любые переменные, рассмотренные нами ранее. Указываем тип, за которым следует имя. Кроме того, компилятор должен знать размер массива, чтобы можно было зарезервировать в памяти необходимое место для его хранения. Эту информацию можно сообщить сразу после имени массива, заключив ее в квадратные скобки. Вот как я определяю массив для хранения артефактов героя:

```
string inventory[MAX_ITEMS];
```

Этот код определяет массив inventory, состоящий из объектов string, которых в нем может насчитываться не более MAX_ITEMS. Поскольку константа MAX_ITEMS равна 10, в нашем распоряжении будет не более 10 объектов string.

ПРИЕМ

Можно инициализировать массив прямо со значениями, если при его объявлении вы сразу указываете список инициализаторов. Это последовательность элементов, заключенная в фигурные скобки, причем сами элементы разделяются запятыми. Вот пример:

```
stringinventory[MAX_ITEMS] = {"sword", "armor", "shield"};
```

В этом коде объявляется массив inventory, состоящий из объектов string и имеющий размер MAX_ITEMS. Первые три элемента массива инициализируются в значениях "sword", "armor" и "shield".

Если не указывать количество элементов в списке инициализаторов, то размер созданного массива будет равен количеству элементов, уже имеющихся в списке, например:

```
string inventory[] = {"sword", "armor", "shield"};
```

Поскольку в списке инициализаторов всего три элемента, в предыдущей строке создается массив inventory размером три элемента. Он состоит из элементов "sword", "armor" и "shield".

Индексация массивов

Индексация массивов во многом напоминает индексацию строковых объектов. Можно обратиться к любому отдельно взятому элементу массива, указав его индексный номер с помощью оператора индексации ([]).

Далее с помощью оператора индексации я добавляю элементы в снаряжение героя:

```
int numItems = 0;
inventory[numItems++] = "sword";
inventory[numItems++] = "armor";
inventory[numItems++] = "shield";
```

Сначала я определяю переменную numItems, где будет содержаться количество вещей, которые герой в настоящее время носит с собой. Далее присваиваю позиции 0 в данном массиве значение "sword". Поскольку я использую постфиксный оператор инкремента, значение переменной numItems увеличивается на единицу уже после того, как в массиве произойдет операция присваивания. В следующих двух строках в массив добавляются значения "armor" и "shield", поэтому, когда выполнение кода завершается, переменная numItems имеет верное значение 3.

Итак, когда у героя появилось какое-то снаряжение, я отображаю эту информацию:

```
cout << "Your items:\n";
for (int i = 0; i < numItems; ++i)
{
    cout<<inventory[i] <<endl;
}
```

Наверняка этот код напоминает вам индексацию строк. С помощью цикла код обрабатывает первые три элемента массива inventory, выводя все строковые значения по порядку.

Далее герой меняет свой меч на боевой топор. Эта операция реализуется с помощью следующего кода:

```
inventory[0] = "battleaxe";
```

В данной строке кода мы присваиваем новое значение элементу, находящемуся на позиции 0 в массиве inventory. Теперь на этой позиции находится строковый объект "battleaxe". Итак, первые три элемента в обновленном массиве — это "battleaxe", "armor" и "shield".

ОСТОРОЖНО!

Как и в случае со строковыми объектами, индексация массивов начинается с 0. Таким образом, следующая строка кода определяет массив, состоящий из пяти элементов:

```
inthighScores[5];
```

В этом массиве присутствуют позиции с номерами от 0 до 4 включительно. Элемента highScores[5] здесь нет! Попытка обратиться к элементу highScores[5] может привести к катастрофическим результатам, в том числе к аварийному завершению программы.

Обращение к функциям-членам элемента массива

Для доступа к функциям-членам элемента массива можно записать этот элемент массива, а после него — оператор доступа к члену и имя функции. Звучит немного сложно, но на самом деле здесь нет ничего сверхъестественного. Вот пример:

```
cout<<inventory[0].size() << " lettersinit.\n";
```

Код `inventory[0].size()` означает, что программа должна вызвать функцию-член `size()` элемента `inventory[0]`. Поскольку в данном случае `inventory[0]` равно `"battleaxe"`, вызов вернет значение `10` — количество символов в строковом объекте.

Следите за границами массива

Как вы уже знаете, при индексации массива необходимо действовать осторожно. Поскольку массив обладает фиксированным размером, вы создаете целочисленную константу для сохранения этой величины. Именно это я и сделал в самом начале программы:

```
constintMAX_ITEMS = 10;
```

В следующих строках я пользуюсь константой `MAX_ITEMS` как ограничителем всякий раз, когда добавляю новый элемент к снаряжению героя:

```
if (numItems < MAX_ITEMS)
{
    inventory[numItems++] = "healing potion";
}
else
{
    cout << "You have too many items and can't carry another.";
}
```

В этом коде я сначала проверил, не превышает ли значение переменной `numItems` константу `MAX_ITEMS`. Если оно действительно меньше константы, то я могу смело использовать это значение `numItems` в качестве индекса и добавить в массив новый строковый объект.

В таком случае переменная `numItems` равна 3, поэтому я присваиваю элемент `"healingpotion"` позиции 3 в данном массиве. Если бы значение превысило константу, то я отобразил бы сообщение «Ваш рюкзак полон, вы не можете взять эту вещь с собой».

Итак, что же произойдет, если вы попытаетесь обратиться к элементу, находящемуся за пределами массива? Зависит от обстоятельств, поскольку в такой ситуации вы обращаетесь к какой-то неизвестной области компьютерной памяти. В худшем случае программа может отреагировать непредсказуемым образом и даже аварийно завершиться.

Проверяя, соответствует ли значение индекса одной из допустимых позиций в пределах массива, мы выполняем так называемый *контроль границ*. Контроль границ просто необходим, если существует вероятность обращения к недопустимому значению индекса.

Си-строки

Еще до появления объектов string программисты, работавшие с C++, представляли строки как массивы символов, завершавшиеся нулем. Такие символьные массивы теперь называются си-строками, поскольку они впервые появились в программах, написанных на языке C. Си-строку можно объявить и инициализировать как самый обычный массив:

```
charphrase[] = "GameOver!!!";
```

Все си-строки заканчиваются символом, называемым *нулевым символом*. Он записывается так: '\0'. Мне не требовалось использовать этот символ в коде, рассмотренном ранее, так как он автоматически сохраняется в конце строки. Итак, строго говоря, объект phrase состоит из 13 элементов. Правда, если функция ориентирована на работу с си-строками, она выдаст для объекта phrase длину 12 — это тоже верно и полностью согласуется с механизмами использования строковых объектов.

При работе с си-строками, как и с любыми другими массивами, можно указать размер массива уже на этапе его определения. Вот еще один способ объявления си-строки:

```
char phrase[81] = "GameOver!!!";
```

В предыдущем коде создается си-строка, которая может содержать до 80 символов, выводимых на печать (а также заключительный нулевой символ).

В си-строках не бывает функций-членов. Но в файле cstring, входящем в состав стандартной библиотеки, вы найдете разнообразные функции, предназначенные для работы с си-строками.

Важное достоинство объектов string заключается в том, что они специально разработаны для безупречного взаимодействия с си-строками. Например, далее приведены разнообразные варианты использования си-строк с объектами string, и все они абсолютно корректны:

```
string word1 = "Game";
char word2[] = " Over";
string phrase = word1 + word2;
if (word1 != word2)
{
    cout << "word1 and word2 are not equal.\n";
}
if (phrase.find(word2) != string::npos)
{
    cout << "word2 is contained in phrase.\n";
}
```

Допускается конкатенация си-строк и объектов string, но в результате всегда получается объект string. Поэтому код char phrase2[] = word1 + word2; приведет к ошибке. Можно сравнивать объекты string и си-строки с помощью реляционных

операторов. Более того, мы даже можем использовать си-строки как аргументы для функций-членов объектов `string`.

Си-строки имеют те же недостатки, что и массивы. Один из самых существенных недостатков заключается в том, что длина си-строки является фиксированной. Вывод: по возможности старайтесь использовать объекты `string`, но при необходимости будьте готовы работать и с си-строками.

Использование многомерных массивов

Как вы уже убедились, последовательности незаменимы при программировании игр. Последовательность-строка позволяет, к примеру, сохранить имя пользователя, а массив — список предметов в ролевой игре. Но в некоторых играх требуются не линейные, а более сложные последовательности. Иногда игра действительно должна быть многомерной. Например, если требуется представить шахматную доску в виде массива из 64 элементов, будет гораздо логичнее построить ее как двухмерную сущность размером 8 × 8 элементов. К счастью, если это требуется для вашей игры, вы можете создавать массивы, содержащие два, три и даже больше измерений.

Знакомство с программой Tic-Tac-Toe Board

Программа Tic-Tac-Toe Board представляет собой поле для игры в крестики-нолики. Программа отображает поле и объявляет, что крестики победили. Хотя эту программу и можно было бы написать в виде одномерного массива, здесь мы реализуем поле именно в виде двухмерного массива. Программа проиллюстрирована на рис. 3.5.

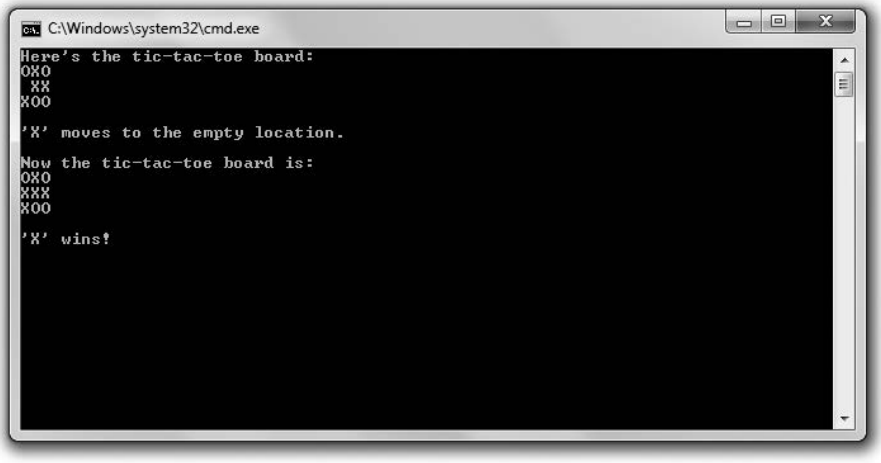


Рис. 3.5. Поле для игры в крестики-нолики представлено в виде двухмерного массива (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 3, имя файла — tic-tac-toe_board.cpp.

```
// Программа Tic-Tac-Toe Board
// Демонстрирует работу с многомерными массивами
#include <iostream>
using namespace std;
int main()
{
    const int ROWS = 3;
    const int COLUMNS = 3;
    char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},
                                  {' ', 'X', 'X'},
                                  {'X', 'O', 'O'} };
    cout << "Here's the tic-tac-toe board:\n";
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << board[i][j];
        }
        cout << endl;
    }
    cout << "\n'X' moves to the empty location.\n\n";
    board[1][0] = 'X';
    cout << "Now the tic-tac-toe board is:\n";
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << board[i][j];
        }
        cout << endl;
    }
    cout<< "\n'X' wins!";
    return 0;
}
```

Создание многомерных массивов

На первом этапе выполнения программы я объявляю и инициализирую массив для поля игры в крестики-нолики.

```
char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},
                              {' ', 'X', 'X'},
                              {'X', 'O', 'O'} };
```

В этом коде объявляется двухмерный символьный массив размером 3 × 3 (поскольку значения ROWS и COLUMNS равны 3). Здесь также инициализируются все элементы.

СОВЕТ

Можно просто объявить многомерный массив, не инициализируя его. Вот пример:

```
char chessBoard[8][8];
```

В этом коде объявляется двухмерный символьный массив chessBoard размером 8×8. Кстати, различные измерения в составе многомерного массива не обязательно должны быть равными. Далее приведено совершенно корректное объявление игровой карты, представленной в виде отдельных символов:

```
char map[12][20];
```

Индексация многомерных массивов

Далее я отображаю в программе поле для игры в крестики-нолики. Но прежде, чем подробно рассмотреть этот этап, поговорим о том, как индексируются отдельные элементы массива. При таком индексировании элементов в многомерном массиве требуется указать значение для каждого измерения в массиве. Именно таким образом я записываю в массиве крестик в свободной клетке:

```
board[1][0] = 'X';
```

Этот код присваивает символ элементу, расположенному на позиции board[1][0] (там только что находилась пустая клетка). Затем я отображаю обновленное поле после этого хода, точно так же, как отображал его до этого хода:

```
for (int i = 0; i < ROWS; ++i)
{
    for (int j = 0; j < COLUMNS; ++j)
    {
        cout << board[i][j];
    }
    cout<< endl;
}
```

С помощью двух вложенных циклов for я двигаюсь через двухмерный массив, отображая при этом символьные элементы. Так на экране вырисовывается поле с партией в крестики-нолики.

Знакомство с игрой «Словомеска»

«Словомеска» (Word Jumble) — это игра-головоломка, в которой компьютер загадывает слово, перемешивая его буквы в случайном порядке. Чтобы выиграть партию, пользователь должен угадать слово. Если пользователь затрудняется, то может запросить у программы подсказку. Игра проиллюстрирована на рис. 3.6.

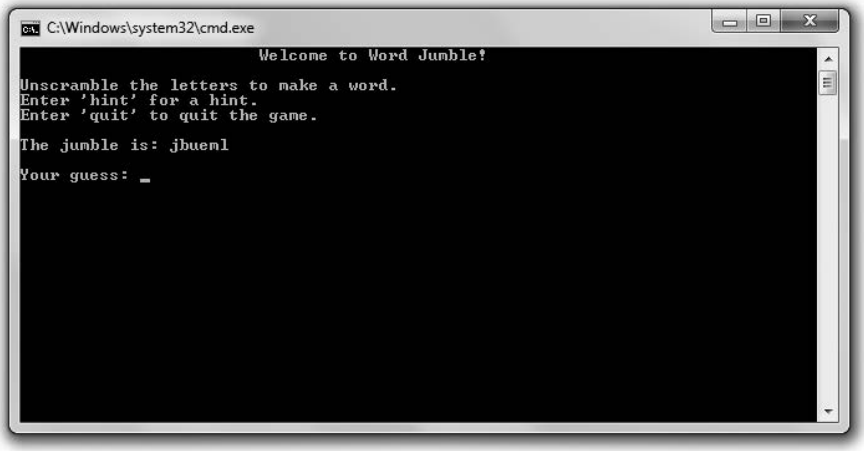


Рис. 3.6. Э-э-э... кажется, здесь написано jumble
(опубликовано с разрешения компании Microsoft)

НА ПРАКТИКЕ

Хотя головоломки обычно не попадают в число топовых игр, крупнейшие компании продолжают их выпускать. Почему? Оказывается, это выгодно. Головоломки не тянут на игровой блокбастер, но стабильно пользуются спросом. Многие геймеры (как играющие эпизодически, так и самые завзятые) не могут оторваться от таинства качественно сработанной головоломки. При этом разработка головоломок обходится гораздо дешевле, чем программирование крупномасштабных игр, для чего требуется нанимать целые команды специалистов и тратить не один год.

Приступаем к разработке программы

Как обычно, я начинаю код с пары комментариев и включаю файлы, которые понадобятся в игре. Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 3, имя файла — `word_jumble.cpp`.

```
// Игра Word Jumble
// Классическая игра-головоломка, в которой пользователь разгадывает слова, с подсказками или без них.
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;
```

Выбор слова для перемешивания

Далее мне нужно выбрать слово для перемешивания — то самое, которое будет загадано игроку. Сначала я составляю список слов и подсказок:

```
int main()
{
```

```
enum fields {WORD, HINT, NUM_FIELDS};
const int NUM_WORDS = 5;
const string WORDS[NUM_WORDS][NUM_FIELDS] =
{
    {"wall", "Do you feel you're banging your head against something?"},
    {"glasses", "These might help you see the answer."},
    {"labored", "Going slowly, is it?"},
    {"persistent", "Keep at it."},
    {"jumble", "It's what the game is all about."}
};
```

Я объявляю и инициализирую двухмерный массив, состоящий из слов и соответствующих им подсказок. В перечислении определяются перечислители для доступа к массиву. Например, объект `WORDS[x][WORD]` всегда будет строковым и представлять одно из слов. Объект `WORDS[x][HINT]` также будет строковым, он представляет соответствующую подсказку.

ПРИЕМ

В последнем перечислителе, входящем в состав перечисления, удобно хранить информацию о количестве элементов. Вот пример:

```
enum difficulty {EASY, MEDIUM, HARD, NUM_DIFF_LEVELS};
cout << "There are " << NUM_DIFF_LEVELS << " difficulty levels.";
```

Здесь константа NUM_DIFF_LEVELS равна 3 — это точное количество уровней сложности, содержащихся в данном перечислении. Поэтому во второй строке выводится сообщение «В этой игре три уровня сложности».

Далее я выбираю из моих вариантов случайное слово:

```
srand(static_cast<unsigned int>(time(0)));
int choice = (rand() % NUM_WORDS);
string theWord = WORDS[choice][WORD]; // слово, которое нужно угадать
string theHint = WORDS[choice][HINT]; // подсказка для слова
```

Я генерирую случайный индекс, исходя из количества слов в массиве. Затем беру две переменные, `theWord` и `theHint`, и записываю в них соответственно слово, расположенное на данной индексной позиции, и сопровождающую его подсказку.

Перемешивание слова

Теперь, когда я выбрал слово, которое загадаю пользователю, мне нужно переставить в нем буквы:

```
String jumble = theWord; // перемешанный вариант слова
intlength = jumble.size();
for (int i = 0; i < length; ++i)
{
    int index1 = (rand() % length);
    int index2 = (rand() % length);
```

```
    char temp = jumble[index1];
    jumble[index1] = jumble[index2];
    jumble[index2] = temp;
}
```

В предыдущем коде я создал копию слова `jumble`, чтобы... перемешать его. Сгенерировал две случайные позиции в объекте `string` и поменял местами символы, стоящие на этих позициях. Я сделаю это несколько раз — столько, сколько букв в слове.

Приглашение игрока

Далее нужно пригласить пользователя поиграть, что я и делаю в следующем коде:

```
cout << "\t\t\tWelcome to Word Jumble!\n\n";
cout << "Unscramble the letters to make a word.\n";
cout << "Enter 'hint' for a hint.\n";
cout << "Enter 'quit' to quit the game.\n\n";
cout << "The jumble is: " << jumble;
string guess;
cout << "\n\nYour guess: ";
cin>>guess;
```

Я объяснил пользователю, как играть, в частности как выйти из игры и попросить подсказку.

СОВЕТ

Даже если ваша игра кажется вам невероятно увлекательной, обязательно предоставляйте пользователю удобный способ выхода из нее.

Начало игрового цикла

Далее начинается игровой цикл:

```
while ((guess != theWord) && (guess != "quit"))
{
    if (guess == "hint")
    {
        cout << theHint;
    }
    else
    {
    cout << "Sorry. that's not it.";
    }
    cout <<"\n\nYour guess: ";
    cin >> guess;
}
```

Цикл предлагает игроку отгадать слово, пока пользователь не угадает слово либо не решит выйти из игры.

Прощание

Когда цикл завершится (пользователь отгадает слово либо решит выйти из игры), с игроком нужно попрощаться:

```
if (guess == theWord)
{
    cout << "\nThat's it! You guessed it!\n";
}
cout << "\nThanks for playing.\n";
return 0;
}
```

Если пользователь угадал слово, я его поздравлю. Затем поблагодарю пользователя за то, что он уделил время моей игре.

Резюме

В этой главе мы изучили следующий материал.

- Циклы `for` позволяют многократно выполнять фрагмент кода. В цикле `for` можно задать инструкцию инициализации, выражение для проверки и действие, которое требуется выполнять после каждой итерации цикла.
- Циклы `for` часто применяются для подсчета или для перебора элементов последовательности.
- Объекты — это целостные сущности, в которых содержатся тесно взаимосвязанные данные *(члены данных)* и функции *(функции-члены)*.
- Объекты `string` (часто именуемые *строками*) определяются в файле `string`, входящем в состав стандартной бибилотеки. В объектах `string` можно хранить последовательности символов, у них также есть функции-члены.
- Объекты `string` определяются таким образом, что они очень удобны для работы с традиционными операторами — в частности, оператором конкатенации и реляционными операторами.
- У всех объектов `string` есть функции-члены, в том числе такие, которые позволяют определить длину объекта `string`, узнать, не пустая ли эта строка, найти в ней подстроки и удалить их.
- В массивах можно хранить последовательности любого типа и обращаться к этим последовательностям.
- Недостаток массивов заключается в том, что они имеют фиксированную длину.
- Можно работать с отдельными элементами строки или массива, для обращения к ним применяется оператор индексации.
- При попытках обратиться к отдельным элементам строк или массивов не выполняется обязательный контроль границ этих объектов. Поэтому обеспечение контроля границ — это задача программиста.

- Си-строки — это символьные массивы, завершаемые нулевым символом. Именно так обычно строятся строки в языке C. Хотя в языке C++ можно без проблем работать с си-строками, при операциях над символьными последовательностями лучше все-таки отдавать предпочтение объектам `string`.
- Многомерные массивы позволяют обращаться к элементам массива сразу по нескольким индексам. Например, шахматную доску можно представить в виде двухмерного массива размером 8 × 8 элементов.

Вопросы и ответы

1. *Какой цикл лучше: while или for?*

   Нельзя сказать, что один из этих циклов безусловно лучше другого. Работайте с тем циклом, который лучше всего подходит для решения стоящей перед вами задачи.

2. *В каких случаях цикл for может быть предпочтительнее, чем цикл while?*

   Можно запрограммировать цикл `for` для выполнения любой задачи, которая обычно решается с помощью цикла `while`. Правда, в некоторых случаях цикл `for` явно более уместен. Типичные случаи такого рода — это подсчет элементов или перебор элементов в последовательности.

3. *Можно ли использовать инструкции break и continue с циклами for?*

   Конечно. Они будут работать так же, как в цикле `while`: `break` разрывает цикл, а `continue` переводит поток выполнения кода обратно к началу цикла.

4. *Почему программисты часто дают имена вроде i, j и k переменным, которые используются в качестве счетчиков в циклах for?*

   Хотите — верьте, хотите — нет, но это просто традиция. Такая практика закрепилась еще в первых версиях языка FORTRAN, где названия целочисленных переменных должны были начинаться с определенных букв, в том числе i, j и k.

5. *Не требуется включать никаких файлов, чтобы использовать типы данных char или int. Почему же требуется включать файл string для работы со строками?*

   Типы `char` и `int` являются встроенными. Они всегда доступны в любой программе, написанной на C++. Тип `string` не является встроенным. Он определяется в файле `string` как часть стандартной библиотеки.

6. *Почему си-строки получили такое название?*

   В языке программирования C принято представлять строки как символьные массивы, завершаемые нулевым символом. Эта практика перешла и в язык C++. После того как в языке C++ появился новый тип данных `string`, программистам потребовался способ для различения «старых» и «новых» строк. С тех пор «старые» строки именуются си-строками.

7. *Почему лучше пользоваться объектами string, а не си-строками?*

   Объекты `string` имеют ряд преимуществ по сравнению с си-строками. Наиболее очевидное таково: размер объекта `string` можно динамически изменять. При создании `string` указывать длину такой строки не требуется.

8. *Следует ли в каких-нибудь случаях пользоваться именно си-строками?*

   Когда это только возможно, работайте с объектами `string`. Если же вы дорабатываете старый проект, реализованный с применением си-строк, вам пригодятся навыки работы с такими строками.

9. *Что такое перегрузка операторов?*

   Перегрузка позволяет использовать традиционные операторы не свойственным им образом, если эти операторы оказываются в заранее обозначенном контексте. В таком случае результат применения оператора будет нетипичным, но предсказуемым. Например, оператор + применяется для сложения чисел, но в перегруженном виде может использоваться для конкатенации строк.

10. *Может ли из-за перегрузки операторов возникнуть какая-либо путаница?*

    Да, при перегрузке оператора его значение меняется. Но новое значение действует только в конкретном новом контексте. Например, очевидно, что в выражении `4 + 6` оператор + означает сложение, тогда как в выражении `myString1 + myString2` оператор `string` объединяет строки.

11. *Можно ли использовать оператор += для конкатенации строк?*

    Да, оператор += перегружен и с его помощью можно объединять строки.

12. *Какую функцию-член следует использовать, чтобы узнать количество символов в объекте string, — length() или size()?*

    Функции `length()` и `size()` вернут одинаковый результат, так что можете использовать любую.

13. *Что такое предикатная функция?*

    Это функция, возвращающая `true` или `false`. Функция-член `empty()`, относящаяся к объекту `string`, является примером предикатной функции.

14. *Что произойдет, если я попытаюсь присвоить значение элементу, находящемуся за пределами массива?*

    Язык C++ допускает такое присваивание. Однако результаты будут непредсказуемыми, вплоть до аварийного завершения программы. Дело в том, что в описанном случае вы изменяете какую-то неизвестную часть памяти компьютера.

15. *Почему следует использовать многомерные массивы?*

    Чтобы работать с группой элементов было понятнее. Например, шахматную доску можно реализовать в виде одномерного массива `chessBoard[64]`, однако гораздо целесообразнее в таком случае прибегнуть к двухмерному массиву, например `chessBoard[8][8]`.

Вопросы для обсуждения

1. Вспомните ваши любимые предметы из какой-нибудь игры, которые можно представить в коде в качестве объектов. Какими членами данных и функциями-членами могли бы обладать такие объекты?
2. Каковы преимущества работы с массивом по сравнению с использованием группы отдельных переменных?

3. Назовите недостатки массивов, связанные с тем, что массивы имеют фиксированный размер.
4. Каковы достоинства и недостатки перегрузки операторов?
5. Какие игры можно создавать, оперируя преимущественно строками, массивами и циклами for?

Упражнения

1. Усовершенствуйте игру «Словомеска», добавив в нее систему подсчета очков. Начисляйте за разгаданное слово столько очков, сколько букв в этом слове. Снимайте очки, если пользователь прибегает к подсказкам.
2. Какие ошибки есть в следующем коде?

```
for (int i = 0; i <= phrase.size(); ++i)
{
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
}
```

3. Какие ошибки есть в следующем коде?

```
const int ROWS = 2;
const int COLUMNS = 3;
char board[COLUMNS][ROWS] = { {'O', 'X', 'O'},
                              {' ', 'X', 'X'} };
```

4 Библиотека стандартных шаблонов. Игра «Виселица»

Ранее мы изучили, как работать с последовательностями значений с помощью массивов. Однако существуют и более изящные способы обращения с коллекциями значений. На самом деле работать с коллекциями приходится так часто, что в стандарте C++ есть целый раздел, посвященный этой проблеме. В этой главе мы подробнее познакомимся с данным разделом, который называется библиотекой стандартных шаблонов. В частности, вы научитесь:

- использовать объекты `vector` для работы с последовательностями значений;
- использовать функции-члены `vector` для манипуляций с элементами последовательностей;
- использовать итераторы для перебора последовательностей;
- использовать библиотечные алгоритмы для работы с группами элементов;
- проектировать будущие программы на псевдокоде.

Знакомство с библиотекой стандартных шаблонов

Хороший программист-игровик знает, что лень — двигатель прогресса. Нет, мы не ленимся трудиться, просто не любим заново выполнять работу, которая когда-то уже была сделана, особенно если она была сделана хорошо. STL *(библиотека стандартных шаблонов)* — это замечательная коллекция готового качественного кода. В ней вы найдете среди прочего группу контейнеров, алгоритмов и итераторов.

Итак, что такое контейнер и как он может пригодиться при написании игр? В контейнере можно хранить коллекции, объединяющие значения одного типа, а также обращаться к этим значениям. Массивы тоже позволяют выполнять такие операции, но контейнеры STL — более гибкие и мощные сущности, чем старый добрый проверенный массив. В STL определяется целый ряд контейнерных типов; контейнеры разных типов работают немного по-разному, поскольку применяются для решения специфических задач.

Алгоритмы, определенные в STL, работают с контейнерами этой библиотеки. *Алгоритмы* — это широко используемые функции, которые программисту-игровику приходится регулярно использовать с группами значений. В частности, существуют алгоритмы для сортировки, поиска, копирования, слияния, вставки и удаления контейнерных элементов. Самое интересное, что один и тот же алгоритм может применяться со многими контейнерными типами.

Итераторы — это объекты, находящие элементы в контейнерах, а также используемые для перехода от элемента к элементу в контейнере. Они отлично подходят для перебора элементов в контейнере. Кроме того, итераторы необходимы для работы с алгоритмами STL.

Весь этот материал станет гораздо понятнее, стоит нам только рассмотреть реализацию одного из контейнерных типов. Этим мы и займемся далее.

Работа с векторами

Класс `vector` определяет один из видов контейнеров, предоставляемых в STL. В принципе, вектор — это *динамический массив*, то есть массив, способный при необходимости увеличиваться и уменьшаться в размере. Кроме того, в классе `vector` определяются функции-члены, предназначенные для манипуляций с элементами вектора. Таким образом, вектор не только не уступает массиву в функциональном отношении, но и превосходит его.

Вероятно, вы уже задаетесь вопросом: «Зачем мне учить эти странные векторы, если я уже могу работать с массивами?» Дело в том, что у векторов по сравнению с массивами есть некоторые преимущества.

- Вектор может расширяться, а массив — нет. Таким образом, если применить вектор для хранения игровых объектов, обозначающих врагов, то размер вектора будет адаптироваться под количество врагов, с которыми приходится сражаться герою. Если бы такая задача решалась с помощью массива, то понадобилось бы создать массив, способный содержать максимальное количество врагов. Но если бы в ходе игры потребовалось увеличить вражескую армию сверх этого размера, то вам пришлось бы туго.
- Векторы можно использовать с алгоритмами STL, а массивы — нельзя. Таким образом, при работе с векторами вы приобретаете сложный встроенный функционал, в частности возможности поиска и сортировки. При работе с массивами пришлось бы писать собственный код для решения всех этих задач.

У векторов по сравнению с массивами есть и некоторые недостатки.

- Для работы с векторами требуется немного больше памяти.
- Когда вектор сильно увеличивается, это может негативно сказаться на производительности программы.
- Некоторые игровые приставки могут не поддерживать работу с векторами.

В целом векторы и стандартную библиотеку шаблонов можно использовать практически в любых проектах.

Знакомство с программой Hero's Inventory 2.0

С точки зрения пользователя программа Hero's Inventory 2.0 очень похожа на программу Hero's Inventory, которую мы рассмотрели в главе 3. В новой версии программы вновь будем хранить и обрабатывать коллекцию объектов `string`, представляющих элементы снаряжения героя. Однако с точки зрения программиста вторая программа значительно отличается от первой. Дело в том, что снаряжение героя представлено в виде вектора, а не в виде массива. На рис. 4.1 показан результат выполнения программы.

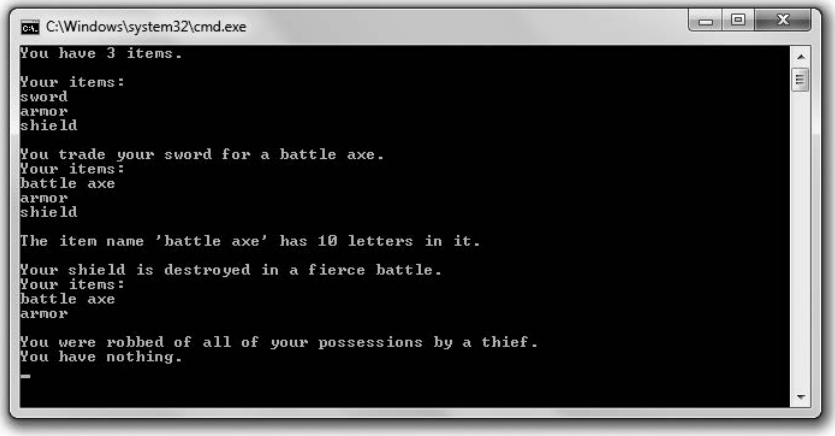


Рис. 4.1. На этот раз снаряжение героя представлено в виде вектора (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 4, имя файла — `heros_inventory2.cpp`.

```
// Программа Hero's Inventory 2.0
// Демонстрирует работу с векторами
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
I    nventory.push_back("armor");
    inventory.push_back("shield");
    cout << "You have " << inventory.size() << " items.\n";
    cout << "\nYour items:\n";
    for (unsigned int i = 0; i < inventory.size(); ++i)
    {
        cout << inventory[i] << endl;
```

```
    }
    cout << "\nYou trade your sword for a battle axe.";
    inventory[0] = "battle axe";
    cout << "\nYour items:\n";
    for (unsigned int i = 0; i < inventory.size(); ++i)
    {
        cout << inventory[i] << endl;
    }
    cout << "\nThe item name '" << inventory[0] << "' has ";
    cout << inventory[0].size() << " letters in it.\n";
    cout << "\nYour shield is destroyed in a fierce battle.";
    inventory.pop_back();
    cout << "\nYour items:\n";
    for (unsigned int i = 0; i < inventory.size(); ++i)
    {
        cout << inventory[i] << endl;
    }
    cout << "\nYou were robbed of all of your possessions by a thief.";
    inventory.clear();
    if (inventory.empty())
    {
        cout << "\nYou have nothing.\n";
    }
    else
    {
        cout << "\nYou have at least one item.\n";
    }
    return 0;
}
```

Подготовка к работе с векторами

Прежде чем я смогу объявить вектор, мне потребуется включить в код файл с определением вектора:

```
#include<vector>
```

Все компоненты библиотеки STL относятся к пространству имен `std`. Поэтому я, как правило, использую для обращения к вектору код, позволяющий обойтись без предварительного указания `std::`.

```
Using namespace std;
```

Объявление вектора

Итак, первым делом мы объявляем в функции `main()` новый вектор:

```
vector<string>inventory;
```

В этой строке объявлен пустой вектор под названием `inventory`, который может содержать объекты типа `string`. Объявлять пустой вектор очень удобно, так как при добавлении в него новых элементов вектор просто увеличивается в размерах.

Чтобы объявить собственный вектор, напишите слово `vector`, а далее — тип объектов, которые собираетесь в нем хранить (тип указывается в угловых скобках <>), затем напишите имя вектора.

СОВЕТ

Вектор можно объявить и другими способами. Например, для вектора можно задать исходный размер, указав соответствующее число в круглых скобках после имени вектора:

```
vector<string> inventory(10);
```

В этом коде объявляется вектор, в котором будут содержаться объекты string. Исходный размер вектора равен 10. Кроме того, можно инициализировать все элементы вектора в одном и том же значении уже при объявлении вектора. Для этого просто задается начальное значение, указываемое после исходного размера вектора, вот так:

```
vector<string> inventory(10, "nothing");
```

В предыдущем коде мы объявили вектор размером 10, а все 10 его элементов инициализировали в значении nothing. Наконец, можно объявить вектор и инициализировать его с содержимым другого вектора:

```
vector<string> inventory(myStuff);
```

В данном коде был создан новый вектор, содержимое которого идентично содержимому вектора myStuff.

Работа с функцией-членом push_back( )

Далее я даю герою те же три исходных артефакта, что и в первой версии программы:

```
inventory.push_back("sword");
inventory.push_back("armor");
inventory.push_back("shield");
```

Функция-член `push_back()` добавляет новый элемент к концу вектора. В предыдущих строках я добавил к вектору `inventory` элементы `"sword"`, `"armor"` и `"shield"`. Теперь элемент `inventory[0]` равен `"sword"`, `inventory[1]` равен `"armor"`, а `inventory[2]` равен `"shield"`.

Работа с функцией-членом size()

Далее я отображаю количество артефактов, которые есть у героя:

```
cout << "You have " << inventory.size() << " items.\n";
```

Чтобы получить размер массива `inventory`, я вызываю функцию `size()` с помощью кода `inventory.size()`. Функция-член `size()` просто возвращает размер вектора, в данном случае — значение 3.

Индексация векторов

Далее я отображаю весь арсенал героя:

```
cout << "\nYour items:\n";
for (unsigned int i = 0; i < inventory.size(); ++i)
{
    cout<<inventory[i] <<endl;
}
```

Точно так же, как и массивы, векторы поддаются индексации с помощью соответствующего оператора `[]`. На самом деле предыдущий код практически идентичен аналогичному фрагменту из исходной программы Hero's Inventory. Вся разница заключается в том, что я воспользовался методом `inventory.size()`, чтобы указать, где должен заканчиваться цикл. Обратите внимание: переменная цикла `i` у меня относится к типу `unsigned int`, так как значения, возвращаемые функцией `size()`, — это беззнаковые целые числа.

Далее я заменяю первый элемент из снаряжения героя:

```
inventory[0] = "battle axe";
```

Опять же, как и в случае с массивами, для присваивания нового значения для позиции, которая уже занята старым элементом, я применяю оператор индексации.

ОСТОРОЖНО!

Хотя векторы и динамические, нельзя увеличить размер вектора с помощью оператора индексации. Например, далее приведен очень опасный код, который не увеличит размер вектора inventory:

```
vector<string> inventory; // создание пустого вектора
inventory[0] = "sword"; // программа может аварийно завершиться!
```

Вызов функций-членов элемента

Далее я отображаю количество букв в названии первого элемента в снаряжении героя.

```
cout << inventory[0].size() << " letters in it.\n";
```

Как и при работе с массивами, доступ к функциям-членам элемента вектора выполняется так: записывается имя элемента, ставится точка (оператор доступа к члену), затем записывается имя функции-члена. Поскольку `inventory[0]` равно `"battle axe"`, `inventory[0].size()` возвращает 10.

Использование функции-члена pop_back()

Удаляю щит героя с помощью следующего кода:

```
inventory.pop_back();
```

Функция `pop_back()` удаляет из вектора последний элемент и уменьшает размер вектора на единицу. Итак, здесь `inventory.pop_back()` удаляет `"shield"` из вектора `inventory`, поскольку это был последний элемент в векторе. Размер `inventory` уменьшается с 3 на 2.

Использование функции-члена clear()

Далее моделируется такая ситуация: вор похищает у героя рюкзак со всеми его вещами:

```
inventory.clear();
```

Функция-член `clear()` удаляет все элементы вектора и устанавливает его значение в 0. После выполнения предыдущей строки кода вектор `inventory` становится пустым.

Использование функции-члена empty()

Наконец, я проверяю, остались ли в снаряжении героя какие-либо предметы:

```
if (inventory.empty())
{
    cout << "\nYou have nothing.\n";
}
else
{
    cout << "\nYou have at least one item.\n";
}
```

Функция-член `empty()` класса `vector` действует примерно так же, как и функция-член `empty()` класса `string`. Она возвращает `true`, если объект `vector` пуст, в противном случае она возвращает `false`. Поскольку вектор `inventory` пуст, программа выводит на экран сообщение `You have nothing` (У вас ничего нет).

Работа с итераторами

Итератор — важнейшая сущность, позволяющая использовать контейнеры на полную мощность. Итераторы позволяют перебирать последовательность элементов, заключенных в контейнере. Кроме того, итераторы требуются для работы с важными компонентами библиотеки STL. Многие контейнерные функции-члены и алгоритмы STL принимают итераторы в качестве аргументов. Если вы хотите в полной мере пользоваться преимуществами этих функций-членов и алгоритмов, то должны научиться работе с итераторами.

Знакомство с программой Hero's Inventory 3.0

Программа Hero's Inventory 3.0 работает примерно так же, как и две предыдущие версии, по крайней мере вначале. Программа выводит на экран список вещей, заменяет первый элемент и отображает количество букв в названии этого элемента. Но затем программа делает кое-что новенькое: вставляет элемент в начало списка, а затем удаляет элемент из середины списка. Это делается с помощью итераторов. Программа в действии показана на рис. 4.2.

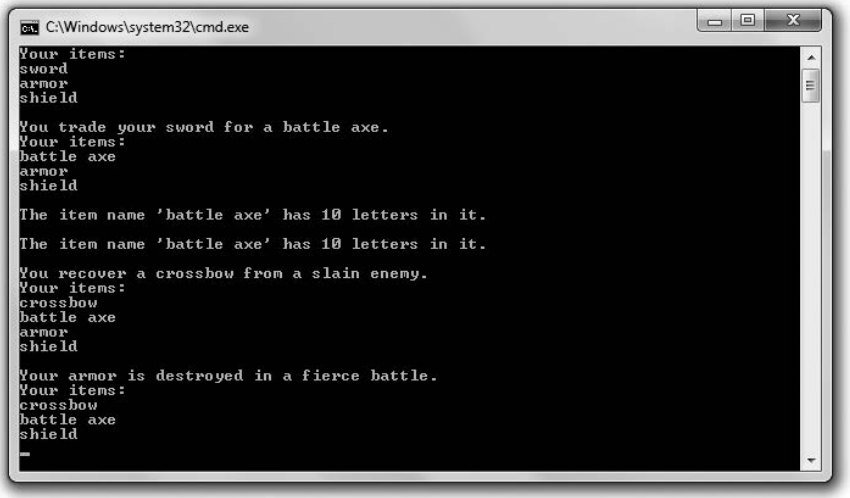


Рис. 4.2. Программа выполняет ряд манипуляций с элементами вектора, для таких операций обычно применяются итераторы (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 4, имя файла — `heros_inventory3.cpp`.

```
// Программа Hero's Inventory 3.0
// Демонстрирует работу с итераторами
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");
    vector<string>::iterator myIterator;
    vector<string>::const_iterator iter;
    cout << "Your items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }
    cout << "\nYou trade your sword for a battle axe.";
    myIterator = inventory.begin();
    *myIterator = "battle axe";
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout <<*iter << endl;
    }
    cout << "\nThe item name '" << *myIterator << "' has ";
```

```
    cout << (*myIterator).size() << " letters in it.\n";
    cout << "\nThe item name '" << *myIterator << "' has ";
    cout << myIterator->size() << " letters in it.\n";
    cout << "\nYou recover a crossbow from a slain enemy.";
    inventory.insert(inventory.begin(), "crossbow");
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }
    cout << "\nYour armor is destroyed in a fierce battle.";
    inventory.erase((inventory.begin() + 2));
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout<< *iter<<endl;
    }
    return 0;
}
```

Объявление итераторов

Когда я объявляю вектор со снаряжением героя и добавляю те самые три строковых объекта, которые уже использовались в предыдущих версиях программы, я также объявляю итератор:

```
vector<string>::iterator myIterator;
```

Здесь я объявил итератор для вектора `myIterator`, содержащего строковые объекты. Чтобы объявить собственный итератор, действуйте по такому же принципу. Напишите тип-контейнер, затем укажите тип объектов, которые будут содержаться в этом контейнере (тип этих объектов указывается в угловых скобках < и >). Далее ставится оператор разрешения области видимости (символ `::`), затем идет слово `iterator` и, наконец, имя вашего нового итератора.

Итак, что же представляют собой *итераторы*? Это значения, идентифицирующие конкретный элемент как контейнер для других элементов. Имея итератор, можно получить доступ к значению элемента. Имея подходящий итератор, можно изменить такое значение. Итераторы также могут переходить между элементами с помощью хорошо знакомых нам арифметических операторов.

Чтобы образно представить себе итераторы, давайте сравним их с этикетками, которые можно наклеивать на конкретные элементы в контейнере. Итератор — это не элемент как таковой, а своеобразная ссылка на элемент. В частности, я могу воспользоваться итератором `myIterator`, чтобы указать на конкретный элемент в векторе `inventory`, словно наклеиваю итератор на этот элемент. Сделав это, я могу не только обратиться к элементу по итератору, но даже изменить элемент таким образом.

Далее я объявляю другой итератор:

```
vector<string>::const_iterator iter;
```

В этой строке кода объявляется константный итератор, называемый `iter` и используемый с вектором, содержащим строковые объекты. *Константный итератор*

подобен обычному итератору, однако с помощью константного итератора нельзя изменять тот элемент, на который он указывает, — элемент должен оставаться постоянным. Можно считать, что константный оператор предоставляет доступ к элементу в режиме «только для чтения». Правда, сам итератор может меняться. Это означает, что вы можете перемещать итератор `iter` по всему вектору `inventory` как посчитаете нужным. Однако итератор `iter` не позволяет изменить значение какого-либо элемента в векторе. Продолжая пример с этикеткой, можно представить себе следующую ситуацию: надпись на этикетке может меняться, но приклеить этот листок можно лишь к строго определенному элементу.

Зачем может потребоваться константный итератор, если он фактически является просто обедненной версией обычного итератора? Во-первых, с помощью константного итератора программист может четче выражать свои мысли. При использовании константного итератора вы недвусмысленно сообщаете, что не собираетесь изменять те элементы, на которые он ссылается. Во-вторых, константный итератор безопаснее. Можно пользоваться константными итераторами, чтобы застраховаться от случайного изменения контейнерного элемента. Если вы попытаетесь изменить элемент с помощью константного итератора, то получите ошибку компиляции.

ОСТОРОЖНО!

Функция push_back() может вывести из строя все итераторы, ссылающиеся на элементы вектора.

Не показались ли вам слишком абстрактными все эти рассуждения об итераторах? Если так — не пугайтесь, вскоре я продемонстрирую настоящий итератор в действии.

Перебор содержимого вектора

Далее я переберу содержимое вектора с помощью цикла и выведу на экран все снаряжение героя:

```
cout << "Your items:\n";
for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    cout<< *iter<<endl;
```

В данном фрагменте кода я применяю цикл `for`, чтобы пройти от первого до последнего элемента в векторе `inventory`. В самом общем смысле можно сказать, что именно так я перебрал артефакты из снаряжения героя в векторе в программе Hero's Inventory 2.0. Но в данном случае я работаю уже не с целочисленными значениями оператора индексации, а с итератором. Можно сказать, что я последовательно переклеил этикетки на каждый из элементов в последовательности, а затем отобразил значения всех элементов, которые в тот или иной момент были снабжены этикетками. В этом небольшом цикле присутствует множество идей, которые мы пока не обсуждали. Остановлюсь на каждой из них по отдельности.

Вызов векторной функции-члена begin()

В инструкции инициализации, используемой в данном цикле, я присваиваю итератору `iter` значение, возвращаемое функцией `inventory.begin()`. Функция-член `begin()` возвращает итератор, указывающий на первый элемент контейнера.

Таким образом, здесь инструкция присваивает такой итератор, который ссылается на первый элемент вектора `inventory` (это строковый объект, равный `"sword"`). На рис. 4.3 схематически представлен итератор, возвращаемый после вызова функции `inventory.begin()`. Обратите внимание: рисунок абстрактен, так как в векторе `inventory` нет строковых литералов `"sword"`, `"armor"` и `"shield"`, есть только объекты `string`.

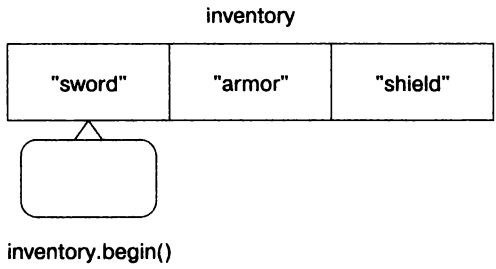


Рис. 4.3. Вызов функции inventory.begin() возвращает итератор, указывающий на первый элемент в векторе

Вызов векторной функции-члена end()

В тестовой инструкции цикла я проверяю возвращаемое значение функции `inventory.end()`: сравниваю это значение с `iter` и убеждаюсь, что два этих значения не равны. Функция `inventory.end()` возвращает итератор, следующий за последним элементом в контейнере. Таким образом, цикл не закончится, пока `iter` не переберет все элементы в векторе `inventory`. На рис. 4.4 дано схематическое представление итератора, возвращаемого при вызове этой функции-члена. Обратите внимание: рисунок абстрактен, так как в векторе `inventory` нет строковых литералов `"sword"`, `"armor"` и `"shield"`, есть только объекты `string`.

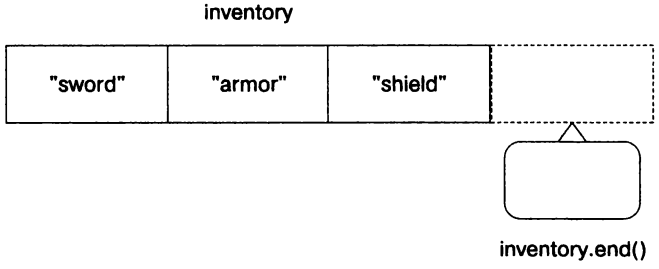


Рис. 4.4. Вызов функции inventory.end() возвращает итератор, указывающий на элемент, стоящий непосредственно после окончания контейнера

ОСТОРОЖНО!

Функция-член end() vector возвращает итератор, указывающий на первый элемент после окончания контейнера, а не на последний элемент в контейнере. Следовательно, вы не сможете получить значение из итератора, возвращаемого функцией end(). Это может показаться нелогичным, но такой принцип отлично подходит для работы с циклами, перебирающими элементы контейнера.

Изменение итератора

Содержащаяся в цикле инструкция действия `++iter` увеличивает на единицу итератор `iter`, который затем переходит к следующему элементу в векторе. В зависимости от итератора отличается и набор арифметических операций, допускаемых этим итератором при перемещении элементов в контейнере. Однако чаще всего требуется выполнять именно инкремент итератора.

Разыменование итератора

В теле цикла я посылаю `*iter` в `cout`. Ставя оператор разыменования (*) перед `iter`, я отображаю значение того элемента, на который ссылается итератор (но не сам итератор). Ставя перед итератором оператор разыменования, вы указываете: «Здесь упоминается элемент, на который указывает итератор, а не сам итератор».

Изменение значения элемента вектора

Далее я изменяю первый элемент, содержащийся в векторе. Вместо строкового объекта, равного `"sword"`, я ставлю строковый элемент, равный `"battleaxe"`. Для начала сделаю так, чтобы `myIterator` ссылался на первый элемент из вектора `inventory`:

```
myIterator = inventory.begin();
```

Затем изменю значение первого элемента:

```
*myIterator = "battleaxe";
```

Учтите, что, разыменовывая `myIterator` оператором *, предыдущая инструкция присваивания гласит: «Присвой значение `"battleaxe"` тому элементу, на который указывает итератор `myIterator`». Сам `myIterator` при этом не изменяется. После инструкции присваивания `myIterator` по-прежнему указывает на первый элемент в векторе.

Просто чтобы убедиться в том, что операция присваивания сработала, я отображаю все элементы, входящие в состав `inventory`.

Доступ к функциям-членам элемента вектора

Далее я отображаю количество символов, которое насчитывается в названии первого элемента, входящего в снаряжение героя:

```
cout << "\nThe item name '" << *myIterator << "' has ";
cout << (*myIterator).size() << " letters in it.\n";
```

Код `(*myIterator).size()` означает: «Возьми объект, полученный в результате разыменования `myIterator`, затем вызови функцию-член `size()` этого объекта». Поскольку `myIterator` ссылается на строковый объект, равный `"battleaxe"`, этот код возвращает значение `10`.

СОВЕТ

Всякий раз, когда вы разыменовываете итератор, чтобы обратиться к члену данных или функции-члену, разыменованный итератор нужно ставить в круглые скобки. Таким образом гарантируется, что оператор доступа к члену будет применяться к тому объекту, на который ссылается итератор.

Код `(*myIterator).size()` не слишком красив, поэтому в языке C++ предлагается альтернативный, более логичный способ выражения такой же логики, который я продемонстрирую в двух следующих строках:

```
cout << "\nThe item name '" << *myIterator << "' has ";
cout << myIterator->size() << " letters in it.\n";
```

Данный код выполняет такую же работу, как и первый фрагмент из двух строк, представленный в этом разделе, а именно: отображает количество символов, содержащихся в объекте `"battleaxe"`. Однако обратите внимание на то, что здесь я заменяю `myIterator->size()` на `(*myIterator).size()`. Несложно заметить, что эта версия (с символом `->`) более удобочитаема. Для компьютера два этих фрагмента кода абсолютно равнозначны, но вторая версия легче воспринимается программистом. В принципе, можно использовать оператор косвенного выбора члена `->` для обращения к функциям-членам или членам данных того объекта, на который ссылается итератор.

СОВЕТ

Более красивые альтернативные варианты синтаксиса принято называть «синтаксический сахар». Такие конструкции заменяют сложный синтаксис сравнительно удобоваримым. Например, вместо кода (*myIterator).size() можно воспользоваться синтаксическим сахаром, содержащим оператор ->, вот так: myIterator->size().

Использование векторной функции-члена insert()

Далее я добавляю новый артефакт в снаряжение героя. Но на этот раз не записываю новый элемент в конец последовательности, а вставляю его в самом начале:

```
inventory.insert(inventory.begin(), "crossbow");
```

Одна из разновидностей функции-члена `insert()` позволяет вставить новый элемент в вектор непосредственно перед элементом, на который ссылается конкретный итератор. Для такой версии функции `insert()` мы сообщаем два аргумента: во-первых, итератор, во-вторых, элемент, который требуется вставить. Здесь я вставляю в вектор `inventory` элемент `"crossbow"`, причем записываю его прямо перед первым элементом. В результате все остальные элементы в массиве сдвинутся на одну позицию. Такая версия функции-члена `insert()` возвращает итератор, указывающий на только что вставленный элемент. В данном случае я не присваиваю возвращенный итератор переменной.

ОСТОРОЖНО!

При вызове функции-члена insert() в векторе выводятся из строя все итераторы, ссылающиеся на элементы вектора после точки, в которой произошла вставка, поскольку все элементы в векторе сдвигаются вниз на одну позицию.

Далее я отображаю содержимое вектора, чтобы убедиться, что операция вставки сработала.

Использование векторной функции-члена erase()

Далее я удаляю элемент из снаряжения героя. Но на этот раз вынимаю элемент не из конца последовательности, а из середины:

```
inventory.erase((inventory.begin() + 2));
```

Один из вариантов функции `erase()` удаляет элемент из вектора. Для такой версии функции `erase()` сообщается один аргумент — итератор, ссылающийся на элемент, который вы хотите удалить. В данном случае я сообщаю аргумент `(inventory.begin() + 2)`, соответствующий итератору, ссылающемуся на третий элемент в векторе `inventory`. Так удаляется строковый объект, равный `"armor"`. В результате все следующие элементы сдвинутся один за другим. Такая версия функции-члена `erase()` возвращает итератор, ссылающийся на элемент, следующий непосредственно после удаленного. Здесь я не присваиваю возвращенный итератор переменной.

ОСТОРОЖНО!

При вызове функции-члена erase() в векторе выводятся из строя все итераторы, ссылающиеся на элементы вектора после точки удаления, поскольку все элементы в векторе сдвигаются вниз на одну позицию.

Далее я отображаю содержимое вектора, чтобы убедиться, что операция удаления сработала.

Использование алгоритмов

Библиотека STL определяет группу алгоритмов, которые позволяют манипулировать элементами в контейнере с помощью итераторов. Алгоритмы предназначены для решения распространенных задач, таких как поиск, рандомизация, сортировка. Эти алгоритмы входят в ваш незаменимый арсенал многофункциональных и эффективных орудий труда. Применяя алгоритмы, можно перепоручить STL рутинную работу по манипуляции элементами в контейнере, причем библиотека будет решать эти задачи хорошо проверенными способами, а вы сможете сосредоточиться на написании игры. Самая сильная черта алгоритмов заключается в их универсальности — один и тот же элемент может работать с элементами, содержащимися в контейнерах разных типов.

Знакомство с программой High Scores

Программа High Scores создает вектор с таблицей рекордов. В этой программе используются алгоритмы STL, предназначенные для поиска, перемешивания и сортировки рекордов. Программа проиллюстрирована на рис. 4.5.

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 4, имя файла — `high_scores.cpp`.

```
// Программа High Scores
// Демонстрирует работу с алгоритмами
#include <iostream>
#include <vector>
```

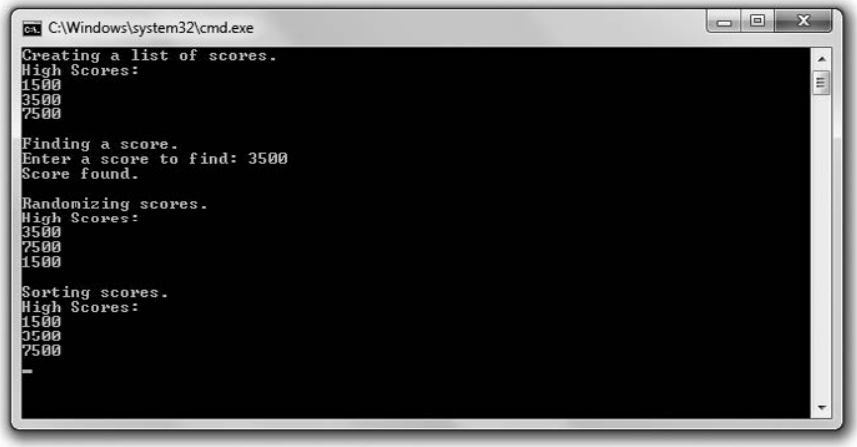


Рис. 4.5. Алгоритмы STL для поиска, перемешивания и сортировки элементов в векторе с таблицей рекордов

```
#include <algorithm>
#include <ctime>
#include <cstdlib>
using namespace std;
int main()
{
    vector<int>::const_iterator iter;
    cout << "Creating a list of scores.";
    vector<int> scores;
    scores.push_back(1500);
    scores.push_back(3500);
    scores.push_back(7500);
    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout << *iter << endl;
    }
    cout << "\nFinding a score.";
    int score;
    cout << "\nEnter a score to find: ";
    cin >> score;
    iter = find(scores.begin(), scores.end(), score);
    if (iter != scores.end())
    {
        cout << "Score found.\n";
    }
    else
    {
        cout << "Score not found.\n";
    }
    cout << "\nRandomizing scores.";
    srand(static_cast<unsigned int>(time(0)));
    random_shuffle(scores.begin(), scores.end());
```

```
    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout << *iter << endl;
    }
    cout << "\nSorting scores.";
    sort(scores.begin(), scores.end());
    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout<< *iter<<endl;
    }
    return 0;
}
```

Подготовка к использованию алгоритмов

Чтобы использовать алгоритмы из библиотект STL, я включаю в программу файл с их определениями:

```
#include<algorithm>
```

Как вы знаете, все компоненты библиотеки STL относятся к пространству имен `std`. С помощью следующего кода (я предпочитаю пользоваться именно им) можно ссылаться на алгоритмы, не ставя перед ними `std::`:

```
Using namespace std;
```

Использование алгоритма find()

Отобразив содержимое вектора `scores`, я получаю от пользователя значение, которое нужно найти и сохранить в переменной `score`. Затем использую алгоритм `find()`, чтобы выполнить поиск данного значения в векторе:

```
iter = find(scores.begin(), scores.end(), score);
```

Алгоритм STL `find()` просматривает указанный диапазон элементов контейнера в поисках заданного значения. Он возвращает итератор, который ссылается на первый элемент, удовлетворяющий условиям поиска. Если совпадений не найдено, то алгоритм возвращает итератор, указывающий на последний элемент рассматриваемого диапазона. Алгоритму нужно сообщить начальную точку поиска как итератор, конечную точку поиска как итератор и искомое значение. Алгоритм выполняет поиск от начального итератора вплоть до конечного итератора, но не захватывая конечный итератор. Здесь я сообщаю в качестве первого и второго аргументов функции `scores.begin()` и `scores.end()`, так как хочу выполнить поиск по всему вектору. В качестве третьего аргумента указываю `score` — таким образом я планирую найти значение, введенное пользователем.

Далее я проверяю, было ли найдено значение `score`:

```
if (iter != scores.end())
{
```

```
    cout << "Score found.\n";
}
else
{
    cout << "Score not found.\n";
}
```

Как вы помните, итератор `iter` будет указывать на первый экземпляр значения `score` в векторе, если это значение будет найдено. Поэтому, если `iter` не равен `scores.end()`, я знаю, что значение было найдено, и вывожу на экран соответствующее сообщение. Если же `iter` окажется равен `scores.end()`, это будет свидетельствовать, что значение `score` найти не удалось.

Использование алгоритма random_shuffle()

Далее я собираюсь рандомизировать рекорды с помощью алгоритма `random_shuffle()`. Точно так же, как и при генерировании отдельного случайного числа, я выполняю посев генератора случайных чисел и лишь потом вызываю алгоритм `random_shuffle()`. Поэтому при каждом запуске программы список рекордов выстраивается в новом порядке:

```
srand(static_cast<unsigned int>(time(0)));
```

Затем я переупорядочиваю список рекордов случайным образом:

```
random_shuffle(scores.begin(), scores.end());
```

Алгоритм `random_shuffle()` рандомизирует элементы в последовательности. Можно указать в виде итераторов начальную и конечную точки той последовательности элементов, которую требуется перемешивать. В данном случае я сообщаю итераторы, возвращаемые функциями `scores.begin()` и `scores.end()`. Два этих итератора указывают, что я хочу перемешать все элементы в векторе `scores`.Таким образом, вектор `scores` будет содержать все те же показатели рекордов, но в случайном порядке.

Затем я отображаю таблицу рекордов, чтобы убедиться, что сортировка работает.

ПРИЕМ

Пусть перемешивание списка рекордов в случайном порядке кажется сомнительным занятием, алгоритм random_shuffle() очень полезен при программировании игр. С его помощью можно выполнять самые разные задачи, от тасования колоды карт до изменения последовательности врагов, с которыми пользователь повстречается на конкретном уровне игры.

Использование алгоритма sort()

Далее я сортирую рекорды:

```
sort(scores.begin(), scores.end());
```

Алгоритм `sort()` сортирует элементы, входящие в последовательность, выстраивая их по возрастанию. В качестве итераторов нужно указать начальный

и конечный элементы той последовательности, которую предполагается отсортировать. Я сообщаю итераторы, возвращаемые функциями `scores.begin()` и `scores.end()`. Два этих итератора указывают, что я хочу отсортировать все элементы, входящие в вектор `scores`. Таким образом, вектор `scores` будет содержать все рекорды, выстроенные в порядке возрастания.

Наконец, я отображаю таблицу рекордов, чтобы убедиться, что сортировка работает.

ПРИЕМ

Одно из самых замечательных свойств, присущих алгоритмам STL, таково: они могут работать с контейнерами, которые были определены вне STL. Эти контейнеры просто должны соответствовать определенным требованиям. Например, хотя объекты string и не входят в состав STL, к ним применимы любые алгоритмы STL. Этот факт проиллюстрирован в следующем коде:

```
stringword = "HighScores";
random_shuffle(word.begin(), word.end());
```

Этот код случайным образом перемешивает символы, содержащиеся в строке word. Как видите, объекты string обладают функциями-членами begin() и end(), которые возвращают итераторы, указывающие соответственно на первый и последний символ в строке. Это одна из причин, по которой алгоритмы STL работают со строками, — просто они для этого приспособлены.

Понятие о производительности векторов

Как и все STL-контейнеры, векторы позволяют программистам игр филигранно работать с информацией. Однако, приобретая такие возможности, приходится частично жертвовать производительностью программы. При этом следует отметить, что любой программист-игровик просто помешан на том, чтобы выжать из своего кода максимальную производительность. Все же не волнуйтесь: и векторы, и другие контейнеры STL невероятно эффективны. Они уже используются во многих играх для ПК и приставок. Правда, у контейнеров есть как достоинства, так и недостатки; разработчик игр должен ориентироваться в производительности контейнеров различных типов и выбирать такие контейнеры, которые оптимально подходят для решения поставленной задачи.

Подробнее о росте векторов

Хотя векторы и могут увеличиваться по мере надобности, каждый из них имеет конкретный размер. Когда в вектор добавляется новый элемент и, соответственно, его размер увеличивается, компилятор перераспределяет память и даже может скопировать все элементы вектора в заново выделенный фрагмент памяти. Порой это негативно сказывается на производительности программы.

Работая над производительностью программы, важнее всего учитывать, так ли она принципиальна. Например, нельзя допускать перераспределение памяти под векторы в критической части вашей программы. Если вы все-таки вынуждены выполнить перераспределение памяти, сделайте это в какой-то другой части программы, не считаясь с потерей производительности. Кроме того, при работе с ма-

ленькими векторами затраты на перераспределение обычно несущественны, поэтому их можно смело проигнорировать. Если же вам действительно требуется более полный контроль над тем, как происходит перераспределение памяти, инструменты для этого имеются.

Использование функции-члена capacity()

Функция-член `capacity()` `vector` возвращает емкость вектора — иными словами, сколько элементов может содержаться в данном векторе при условии, что программа не будет перераспределять под него дополнительную память. Емкость вектора не тождественна его размеру (то есть количеству элементов, которые содержатся в нем в настоящий момент). Далее приведен фрагмент кода, поясняющий этот феномен:

```
cout<< "Creating a 10 element vector to hold scores.\n";
vector<int>scores(10, 0); // инициализирую все 10 элементов в значении 0
cout << "Vector size is :" << scores.size() << endl;
cout <<"Vector capacity is:" << scores.capacity() << endl;
cout << "Adding a score.\n";
scores.push_back(0); // в ответ на увеличение вектора память перераспределяется
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;
```

Сразу после объявления и инициализации вектора этот код сообщает, что и размер, и емкость вектора равны 10. Но после того, как в вектор добавляется еще один элемент, код сообщает, что размер вектора стал равен 11, а емкость — 20. Дело в том, что емкость вектора удваивается всякий раз, когда программа выделяет под него дополнительную память. В данном случае после добавления нового рекорда произошло перераспределение памяти, поэтому емкость вектора удвоилась с 10 до 20.

Использование функции-члена reserve()

Функция-член `reserve()` увеличивает емкость вектора на величину, указываемую в качестве аргумента этой функции. Функция `reserve()` позволяет лучше контролировать расход памяти при операциях ее перераспределения. Вот пример:

```
cout <<"Creating a list of scores.\n";
vector<int> scores(10, 0); // инициализирую все 10 элементов в значении 0
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;
cout << "Reserving more memory.\n";
scores.reserve(20); // резервирую память для 10 дополнительных элементов
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;
```

Сразу после объявления и инициализации вектора этот код сообщает, что и размер, и емкость вектора равны 10. Но после того, как я резервирую память еще для 10 элементов, код сообщает, что размер вектора остался равен 10, а его емкость теперь равна 20.

Пользуясь функцией-членом `reserve()` для обеспечения адекватной емкости вектора, зависящей от стоящих перед вами задач, можно отложить перераспределение памяти на тот момент, когда выполнить его будет удобно.

СОВЕТ

Начинающему программисту-игровику желательно разобраться в том, как работает механизм выделения памяти в векторах, однако не зацикливайтесь на этом. Первые программы, которые вы будете писать, вряд ли серьезно выиграют от таких манипуляций с памятью.

Подробнее о вставке и удалении элементов

Если требуется вставить или удалить элемент в начале или конце вектора, то лучше всего использовать при этом функции-члены `push_back()` или `pop_back()`. Однако если нужно вставить или удалить элемент в любой другой точке вектора (например, с помощью функций `insert()` или `erase()`), то задача серьезно усложнится, так как удаление или вставка повлияют на позиции сразу многих элементов. При работе с небольшими векторами такие издержки обычно несущественны, но если вектор велик (допустим, содержит тысячи элементов), то вставка или затирание элементов в середине такого вектора может значительно ухудшить производительность.

К счастью, в библиотеке STL для хранения последовательностей предоставляется другой тип контейнеров, `list`, обеспечивающий эффективную вставку и удаление элементов независимо от размера конкретной последовательности. Важно помнить, что один контейнер определенно не подходит для решения любой проблемы. Хотя `vector` и является универсальным и наиболее популярным контейнером в STL, в некоторых случаях действительно целесообразнее работать с другими контейнерами.

ОСТОРОЖНО!

Если вам требуется вставлять элементы в середине последовательности или удалять их оттуда, это еще не повод отказываться от вектора! В игре вектор все равно может быть очень удобен. Если последовательность невелика, причем в ней требуется удалить или добавить считаные элементы, то, возможно, лучше все-таки остановиться на векторе.

Исследование других контейнеров библиотеки STL

В библиотеке STL определяются разнообразные контейнерные типы, которые относятся к двум основным типам: последовательные и ассоциативные. Работая с *последовательным контейнером*, можно извлекать значения из него по порядку, тогда как из *ассоциативного контейнера* можно извлекать значение по ключу. Вектор — это пример последовательного контейнера.

Как пользоваться этими разнотипными контейнерами? Допустим, мы программируем онлайновую пошаговую стратегическую игру. Можно применить последо-

вательный контейнер для хранения группы игроков, которые должны действовать по очереди. Но в такой игре можно воспользоваться и ассоциативным контейнером, чтобы извлекать информацию о пользователе случайным образом, по его уникальному идентификатору, например IP-адресу.

Наконец, в STL определяются контейнеры-адаптеры, модифицирующие те или иные последовательные контейнеры. *Контейнеры-адаптеры* — это стандартные структуры данных, изучаемые в курсе информатики. Хотя официально они не считаются контейнерами, адаптеры выглядят и функционируют так же, как контейнеры. В табл. 4.1 перечислены типы контейнеров, предлагаемые в библиотеке STL.

Таблица 4.1. Контейнеры STL

Контейнер	Тип	Описание
deque	Последовательный	Двусвязная очередь
list	Последовательный	Линейный список
map	Ассоциативный	Коллекция пар «ключ/значение», где каждый ключ ассоциирован ровно с одним значением
multimap	Ассоциативный	Коллекция пар «ключ/значение», где каждый ключ может быть ассоциирован более чем с одним значением
multiset	Ассоциативный	Коллекция, в которой каждый элемент не обязательно должен быть уникальным
priority_queue	Адаптер	Очередь с приоритетом
queue	Адаптер	Очередь
set	Ассоциативный	Коллекция, в которой все элементы уникальны
stack	Адаптер	Стек
vector	Последовательный	Динамический массив

Планирование программ

Все программы, рассмотренные нами до сих пор, были довольно простыми. Идея предварительного вычерчивания таких программ на бумаге кажется нелепой. Однако это не так. Планируя все программы (даже самые маленькие), вы практически наверняка экономите время и сберегаете себе нервы.

Программирование во многом напоминает архитектуру. Представьте себе, каким получится дом, построенный без чертежа. Жуть! Чего доброго, в этом доме окажется 12 санузлов, ни одного окна, а входная дверь будет на втором этаже. Кроме того, этот дом может обойтись вам вдесятеро дороже, чем указано в смете. Программирование — точно такая же работа. Без предварительного плана вы, вероятно, будете продираться через сплошные трудности и тратить время. Более того, у вас может получиться программа, которая просто не работает.

Использование псевдокода

Многие программисты выполняют «эскизы» своих программ на *псевдокоде* — естественном языке, формально напоминающем язык программирования. В принципе, псевдокод должен быть понятен любому читателю. Приведу пример. Допустим, я хочу заработать миллион долларов. Достойная цель, но как ее достичь? Нужен план. Я вынашиваю такой план и записываю его на псевдокоде:

```
Если придумать инновационный и очень полезный продукт
То это будет мой продукт
Иначе
Красиво упаковать уже имеющийся продукт и выдать его за свой
Составить о нем рекламно-информационный ролик
Показать ролик по телевизору
Брать 100 долларов за единицу продукта
Продать 10 000 единиц продукта
```

Хотя мой план будет понятен любому читателю, даже не программисту, этот псевдокод немного напоминает компьютерную программу. Первые четыре строки напоминают инструкцию `if` с условием `else`, и это не случайно. Когда пишешь план на псевдокоде, стилистически он должен восприниматься как настоящий код.

Пошаговое усовершенствование программы

Вполне вероятно, что черновой проект вашей программы еще будет дорабатываться. Часто требуется подготовить несколько редакций псевдокода, прежде чем его можно будет реализовать в виде программного кода. *Пошаговое усовершенствование* — это процесс постепенного переписывания псевдокода и подготовки его к реализации в виде программы. Пошаговое усовершенствование — довольно простой процесс. В целом, работа ведется по принципу: «Формулируй проще! Еще проще!» Берется каждый шаг, упомянутый в псевдокоде, и дробится на ряд более простых шагов. В ходе такой детализации план начинает все больше напоминать программный код. Применяя пошаговое усовершенствование, вы разжевываете каждый шаг до тех пор, пока не становится очевидно, что полученный план можно легко переписать на языке программирования. Возьмем для примера один из этапов моего плана о том, как заработать миллион долларов:

```
Составить о нем рекламно-информационный ролик
```

Конечно, эта задача может быть сформулирована точнее. Как делаются такие ролики? С помощью пошагового усовершенствования можно разбить один этап на несколько. В результате имеем:

```
Написать сценарий для информационно-рекламного ролика о моем продукте
На сутки арендовать студию для съемки
Нанять съемочную группу
Нанять оптимистично настроенных зрителей
Отснять ролик
```

Если вы считаете, что эти пять этапов вполне достижимы и не нуждаются в более детальном описании, то эта часть псевдокода достаточно улучшена. Если по поводу этого этапа остаются какие-то неясности, продолжайте пошаговое усовершенствование до тех пор, пока не выработаете полностью готовый план и не заработаете миллион долларов.

Знакомство с программой «Виселица»

В игре «Виселица» (Hangman) компьютер загадывает слово, которое пользователь пытается отгадать буква за буквой. Игрок имеет право на восемь ошибок. Если угадать слово еще не удалось, а восемь ошибок уже допущено, игрока «вешают» и партия завершается поражением. Игра проиллюстрирована на рис. 4.6.

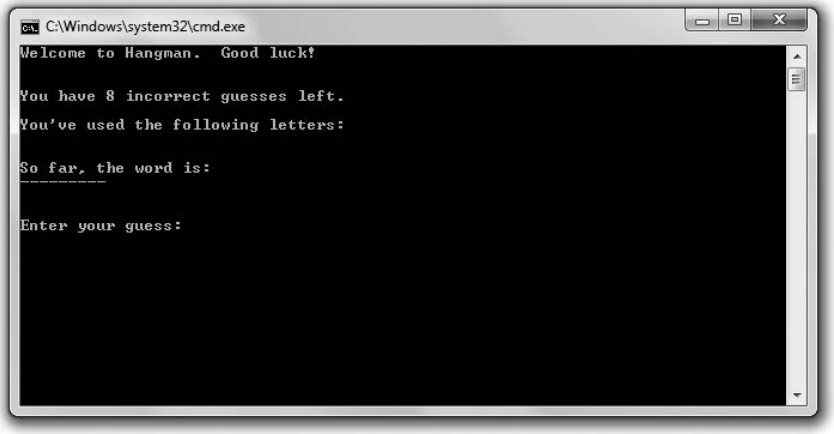


Рис. 4.6. Игра «Виселица» в действии (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 4, имя файла — `hangman.cpp`.

Планирование игры

Прежде чем приступать к написанию программы на C++, я спланирую всю игру на псевдокоде:

```
Создать подборку слов
Выбрать из подборки одно слово, загадать его
Если игрок еще не превысил лимит ошибок, но и не разгадал это слово
    Сообщить игроку, сколько ошибок он еще имеет право допустить
    Показать игроку буквы, которые он уже угадал
    Показать игроку, какую часть загаданного слова он уже открыл
    Получить от игрока следующий вариант буквы
    Если игрок предложит букву, которую он уже угадал
        Получить вариант игрока
```

```
        Добавить новый вариант в множество использованных букв
    Если предложенная буква присутствует в загаданном слове
        Сообщить пользователю, что эта догадка верная
        Добавить в уже имеющийся фрагмент слова угаданную букву
    Иначе
        Сообщить игроку, что предложенный им вариант неверен
        Увеличить на единицу количество ошибочных вариантов, предложенных игроком
    Если игрок допустил слишком много ошибок
        Сообщить игроку, что он повешен
    Иначе
        Поздравить игрока с разгадкой секретного слова
```

Хотя строки этого псевдокода не вполне соответствуют тому порядку строк, который мне предстоит написать на C++, я думаю, что в этом псевдокоде хорошо изложил стоящие передо мной задачи. Далее приступаю к написанию программы.

Подготовка программы

Как обычно, начинаю программу с вводных комментариев и включаю файлы, которые мне понадобятся:

```
// Виселица
// Классическая игра "Виселица"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cctype>
using namespace std;
```

Обратите внимание: здесь я включаю файл нового типа, `cctype`. Он входит в состав стандартной библиотеки и содержит функции для перевода символов в верхний регистр. Это помогает мне сравнивать отдельные символы (поскольку все они записаны в одном регистре).

Инициализация переменных и констант

Далее я начинаю функцию `main()`, в которой инициализирую константы и переменные для игры:

```
Int main()
{
    // подготовка
    const int MAX_WRONG = 8; // максимально допустимое количество ошибок
    vector<string>words; // подборка слов для загадывания
    words.push_back("GUESS");
    words.push_back("HANGMAN");
    words.push_back("DIFFICULT");
    srand(static_cast<unsigned int>(time(0)));
    random_shuffle(words.begin(), words.end());
    const string THE_WORD = words[0]; // слово для отгадывания
```

```
int wrong = 0; // количество ошибочных вариантов
string soFar(THE_WORD.size(), '-'); // часть слова, открытая на данный момент
string used = ""; // уже отгаданные буквы
cout << "Welcome to Hangman. Good luck!\n";
```

Константа MAX_WRONG означает максимальное количество неверных догадок, которые имеет право сделать пользователь. words — это вектор слов, которые могут быть загаданы в игре. Я перемешиваю слова случайным образом с помощью алгоритма random_shuffle(), а затем присваиваю первое слово из вектора константе THE_WORD. Она означает то слово, которое должен отгадать игрок. Переменная wrong — это количество неверных вариантов, уже предложенных игроком. Переменная soFar — это часть слова, уже разгаданная игроком. Переменная soFar изначально выглядит как ряд дефисов — по одному на каждую букву в секретном слове. Когда игрок угадывает букву, имеющуюся в секретном слове, я заменяю соответствующий дефис в переменной этой буквой.

Начало основного цикла

Далее я начинаю основной цикл, который продолжается до тех пор, пока игрок не исчерпает лимит ошибок либо не отгадает слово:

```
// основной цикл
while ((wrong < MAX_WRONG) && (soFar != THE_WORD))
{
    cout << "\n\nYou have " << (MAX_WRONG - wrong);
    cout << " incorrect guesses left.\n";
    cout << "\nYou've used the following letters:\n" << used << endl;
    cout << "\nSo far, the word is:\n" << soFar << endl;
```

Получение вариантов от пользователя

Далее я получаю от пользователя одну из догадок (букву):

```
    char guess;
    cout << "\n\nEnter your guess: ";
    cin>>guess;
    guess = toupper(guess); // перевод в верхний регистр,
    // так как загаданное слово записано в верхнем регистре
    while (used.find(guess) != string::npos)
    {
        cout << "\nYou've already guessed " << guess << endl;
        cout << "Enter your guess: ";
        cin >> guess;
        guess = toupper(guess);
    }
    used += guess;
    if (THE_WORD.find(guess) != string::npos)
    {
        cout << "That's right! " << guess << " is in the word.\n";
        // обновить переменную soFar, включив в нее новую угаданную букву
        for (int i = 0; i < THE_WORD.length(); ++i)
```

```
        {
            if (THE_WORD[i] == guess)
            {
                soFar[i] = guess;
            }
        }
    }
    else
    {
        cout << "Sorry, " << guess << " isn't in the word.\n";
        ++wrong;
    }
}
```

Я перевожу пользовательский вариант в верхний регистр с помощью функции `uppercase()`, определяемой в файле `cctype`. Этот шаг необходим, чтобы можно было сравнивать буквы в верхнем регистре именно с буквами в верхнем регистре, ведь загаданное слово в программе записывается именно в верхнем регистре.

Если игрок предложит букву, которую он уже угадал, я дам ему возможность сделать еще одну догадку. Если вариант пользователя окажется верным, то я обновлю часть слова, открытую на текущий момент. В противном случае сообщу игроку, что предложенный им вариант неверен, и увеличу на единицу количество допущенных ошибок.

Конец игры

Рано или поздно игрок либо угадает слово, либо допустит слишком большое количество ошибок. В обоих случаях игра закончится:

```
// конец игры
if (wrong == MAX_WRONG)
{
    cout << "\nYou've been hanged!";
}
else
{
    cout << "\nYou guessed it!";
}
cout << "\nThe word was " << THE_WORD << endl;
return 0;
}
```

Поздравляю пользователя либо, наоборот, огорчаю его: «Вы повешены». Затем открываю секретное слово.

Резюме

В этой главе мы изучили следующие концепции.

- Стандартная библиотека шаблонов (STL) — это мощная коллекция программного кода, содержащая контейнеры, алгоритмы и итераторы.

- Контейнеры — это объекты, в которых можно хранить коллекции однотипных значений; затем можно обращаться к значениям из такой коллекции.
- Алгоритмы, определенные в библиотеке STL, могут использоваться вместе с контейнерами; алгоритмы предоставляют востребованные функции для работы с группами объектов.
- Итераторы — это объекты, идентифицирующие элементы в контейнеры. С помощью итераторов удобно переходить в контейнере от элемента к элементу.
- Итераторы очень важны для максимально эффективного использования контейнеров. Для работы со многими функциями-членами контейнеров требуются итераторы, итераторы нужны и для операций с алгоритмами.
- Чтобы получить значение, на которое указывает итератор, итератор нужно разыменовать с помощью специального оператора разыменования (*).
- Вектор — это один из последовательных контейнеров, предоставляемых в STL. Он действует как динамический массив.
- Перебор элементов в векторе очень эффективен. Кроме того, в конце вектора очень удобно удалять или вставлять элемент.
- Вставка или удаление элементов в середине вектора могут быть неэффективны, особенно если сам вектор велик.
- Псевдокод — это текст, одновременно напоминающий как естественный язык, так и язык программирования. На псевдокоде удобно планировать программы.
- Пошаговое усовершенствование — это процесс постепенного переписывания псевдокода, при этом псевдокод подготавливается к реализации.

Вопросы и ответы

1. *В чем заключается важность STL?*

   Эта библиотека помогает программисту сэкономить время и силы. В библиотеке STL предоставляются самые востребованные типы контейнеров и алгоритмы.

2. *Быстро ли работает библиотека STL?*

   Весьма. Библиотеку STL оттачивали сотни программистов, что позволяет выжать из нее максимальную производительность на всех поддерживаемых платформах.

3. *В каких случаях следует пользоваться векторами, а не массивами?*

   Практически всегда. Векторы эффективны и удобны. Для работы с ними требуется немного больше памяти, чем для работы с массивами, но такие издержки практически всегда себя оправдывают.

4. *Не уступает ли вектор массиву по скорости работы?*

   Обращение к элементу вектора происходит практически так же быстро, как и к элементу массива. Кроме того, перебор вектора обычно не менее быстр, чем перебор массива.

5. *Я ведь могу использовать оператор индексации при работе с векторами — зачем же вообще мне могут понадобиться итераторы?*

   По нескольким причинам. Во-первых, итераторы нужны для работы со многими функциями-членами объекта `vector` (`insert()` и `erase()` — всего два примера). Во-вторых, итераторы нужны для работы с алгоритмами STL. Наконец, оператор индексации нельзя использовать с большинством контейнеров STL, так что вам все равно рано или поздно придется учиться работать с итераторами.

6. *Каким способом лучше всего обращаться к элементам вектора — с помощью итераторов или с помощью оператора индексации?*

   Зависит от ситуации. Если вам требуется обращаться к элементам в случайном порядке, то для такой задачи лучше подходит оператор индексации. Если приходится работать с алгоритмами STL, то лучше использовать итераторы.

7. *Как лучше перебирать элементы вектора — с помощью оператора индексации или итераторов?*

   Оба способа хороши. Однако преимущество итератора заключается в том, что он позволяет при необходимости заменить вектор другим контейнером STL (например, списком) и для этого почти не потребуется менять код.

8. *Почему в библиотеке STL определяется несколько типов последовательных контейнеров?*

   Различные типы последовательных контейнеров различаются по признаку производительности. Они напоминают инструменты в ящике столяра: каждый инструмент лучше всего подходит для выполнения конкретной работы.

9. *Что такое контейнеры-адаптеры?*

   Контейнеры-адаптеры построены на основе одного из последовательных контейнеров STL, они представляют собой стандартные компьютерные структуры данных. Хотя официально адаптеры не считаются контейнерами, они выглядят и функционируют так же, как контейнеры.

10. *Что такое стек?*

    Это структура данных, элементы которой удаляются в обратном порядке относительно того, в котором они добавлялись. Таким образом, последний добавленный элемент будет удален первым. Стек напоминает обычную стопку книг: если вы берете из стопки верхнюю книгу, то, вероятно, именно ее вы положили в стопку последней.

11. *Что такое очередь?*

    Это структура данных, элементы из которой удаляются в том же порядке, в котором добавлялись. Она напоминает обычную очередь в кассу: чем ближе к началу очереди стоит человек, тем раньше он будет обслужен.

12. *Что такое двусвязная очередь?*

    Это структура данных, допускающая добавление и удаление элементов с любого из двух ее концов.

13. *Что такое очередь с приоритетом?*

    Это структура данных, позволяющая найти в очереди и удалить из нее элемент с наивысшим приоритетом.

14. *Когда следует пользоваться псевдокодом?*

    Всякий раз, когда планируете программу или большой фрагмент кода.

15. *Когда следует пользоваться пошаговым усовершенствованием?*

    В случаях, когда требуется добиться максимальной детализации вашего псевдокода.

Вопросы для обсуждения

1. Почему разработчикам игр следует пользоваться библиотекой STL?
2. Каковы преимущества вектора по сравнению с массивом?
3. Какие типы игровых объектов можно хранить в векторе?
4. Как характеристики, связанные с производительностью контейнера определенного типа, влияют на выбор этого контейнера для решения конкретной задачи?
5. Почему важно планировать программы?

Упражнения

1. Воспользуйтесь векторами и итераторами и напишите программу, позволяющую пользователю вести список любимых игр. В этой программе у пользователя должны быть такие возможности: перечислить заголовки всех игр, добавить заголовок игры, удалить заголовок игры.
2. Допустим, `scores` — это вектор, содержащий объекты типа `int`. В таком случае что неверно в следующем фрагменте кода (код предназначен для увеличения каждого из элементов на единицу)?

```
vector<int>::iterator iter;
// увеличиваем на единицу каждое из значений
for (iter = scores.begin(); iter != scores.end(); ++iter)
{
    iter++;
}
```

3. Напишите псевдокод для игры «Словомеска» из главы 3.

5 Функции. Игра «Безумные библиотекари»

Каждая из программ, которые мы писали ранее, состояла всего из одной функции `main()`. Однако, когда программа достигает определенных размера и уровня сложности, с ней становится неудобно работать в рамках одной функции. К счастью, существуют способы разбивать программу на компактные, более удобоваримые фрагменты кода. В этой главе мы познакомимся с одним из таких способов — научимся создавать новые функции. В частности, мы обсудим:

- как писать новые функции;
- задавать значения для новых функций, сообщая их в виде параметров;
- возвращать информацию от новых функций в виде возвращаемых значений;
- работать с глобальными переменными и константами;
- перегружать функции;
- встраивать функции.

Создание функций

На языке C++ можно писать программы, состоящие из множества функций. Новые функции, создаваемые вами, будут работать точно так же, как и те функции, что входят в стандартный состав языка. Функция приступает к выполнению задачи, справляется с ней, а затем передает управление вашей программе. Писать новые функции удобно как раз потому, что таким образом вы можете разбивать код на небольшие фрагменты, которыми легко управлять. Как и уже изученные ранее функции из стандартной библиотеки, ваши функции должны работать хорошо.

Знакомство с программой Instructions

Результаты программы Instructions незамысловаты: мы видим на экране несколько строк, в которых пользователь получает краткий инструктаж перед игрой. Если судить по внешнему виду, то кажется, что мы могли бы написать программу

Instructions еще в главе 1. Однако в этой программе появляется свежий, пусть и неявный элемент — новая функция. Рассмотрим неброские результаты выполнения кода этой программы (рис. 5.1).

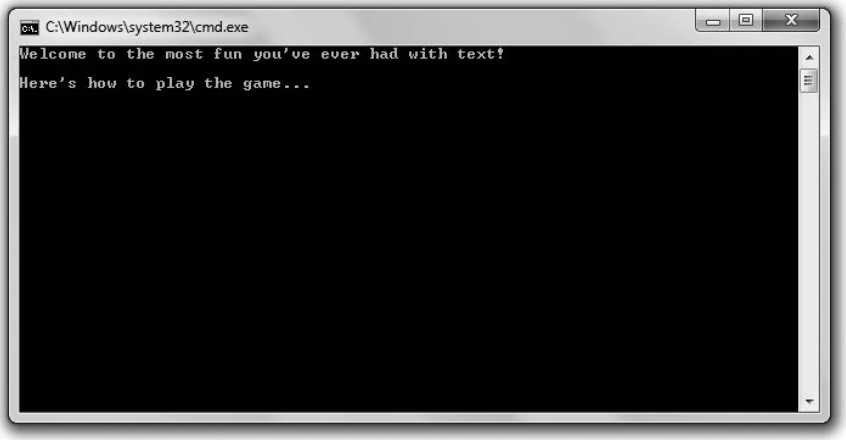


Рис. 5.1. Инструкции выводятся на экран с помощью функции (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `instructions.cpp`.

```
// Программа Instructions
// Демонстрирует написание новых функций
#include <iostream>
using namespace std;
// прототип функции (объявление)
void instructions();
int main()
{
    instructions();
    return 0;
}
// определение функции
void instructions()
{
    cout << "Welcome to the most fun you've ever had with text!\n\n";
    cout << "Here's how to play the game...\n";
}
```

Объявление функций

Прежде чем вы сможете вызвать написанную вами функцию, ее потребуется объявить. Один из способов сделать это — подготовить *прототип функции*, то есть код, который эту функцию описывает. Чтобы написать прототип функции, нужно

указать ее возвращаемое значение (или void, если функция не возвращает значения), затем написать имя функции, затем — список принимаемых ею параметров, заключенный в круглые скобки. *Параметры* получают значения, передаваемые в виде аргументов при вызове функции.

Непосредственно перед функцией main() я пишу прототип функции:

```
void instructions();
```

В данной строке кода я объявил функцию instructions, которая не возвращает значений, — это понятно, поскольку в качестве возвращаемого типа я указал void. Функция также не принимает никаких значений, поэтому не имеет параметров (это также понятно, поскольку в скобках после функции ничего нет).

Прототипы — не единственный вариант объявления функций. Объявление функции также можно совместить с ее определением. Для этого просто нужно поставить определение функции до точки вызова этой функции в программе.

СОВЕТ

Хотя использовать прототипы и не обязательно, они очень полезны, в том числе и потому, что с ними ваш код становится понятнее.

Определение функций

Определить функцию — значит написать весь код, благодаря которому она будет работать. Определяя функцию, вы указываете ее возвращаемое значение (или void, если функция не возвращает значения), затем пишете имя функции, а после этого — список ее параметров, перечисленных в круглых скобках. Процесс очень напоминает создание прототипа функции (за тем исключением, что в конце такой строки кода не ставится точка с запятой). Так получается *заголовок функции*. Затем пишется заключенный в фигурные скобки блок кода, в котором перечисляются все инструкции, которые должны выполняться в ходе работы функции. Этот код называется *телом функции*.

В конце программы Instructions я определяю простую функцию instructions(), выводящую на экран ряд предыгровых подсказок. Поскольку эта функция не возвращает значения, я обхожусь без инструкции return, которой мы пользовались в функции main(). Определение функции завершается обычной закрывающей фигурной скобкой:

```
Void instructions()
{
    cout<< "Welcome to the most fun you've ever had with text!\n\n";
    cout << "Here's how to play the game...\n";
}
```

ОСТОРОЖНО!

У функции и ее прототипа должны совпадать имя и возвращаемый тип, в противном случае вы получите ошибку компиляции.

Вызов функций

Вы вызываете написанные вами функции точно так же, как и любые другие: пишете имя функции, ставите после него круглые скобки, а в скобках указываете валидный список аргументов. В функции `main()` я вызываю мою новую функцию совершенно безыскусно:

```
instructions();
```

Эта строка кода вызывает функцию `instructions()`. Когда вызывается функция, ей передается управление программой. В данном случае управление программой передается функции `instructions()` и программа выполняет код функции, выводящей на экран игровые инструкции. Когда функция завершает свою работу, управление программой возвращается тому коду, который ее вызвал. Здесь управление программой возвращается функции `main()`. Далее выполняется следующая инструкция функции `main()` (`return 0;`), и вся программа завершается.

Понятие об абстрагировании

Работа по написанию и вызову функций называется *абстрагированием*. Благодаря абстрагированию можно держать в уме целостную картину происходящего, не вдаваясь в детали. Так, я могу просто использовать функцию `instructions()`, не задумываясь о том, как именно текст отображается на экране. Мне просто нужно вызвать функцию одной строкой кода, и работа будет выполнена.

Возможно, вы задумаетесь, где еще может применяться абстрагирование, но оказывается, что мы постоянно им пользуемся! Например, представьте себе двоих сотрудников, работающих в ресторане быстрого питания. Если один из них говорит другому, что только что «сделал номер 3» и «оформил его», то его коллега понимает: первый сотрудник принял у клиента заказ, пошел к инфракрасным лампам, взял бургер, пошел к фритюрнице, насыпал большую порцию картофеля фри, далее пошел к сифону с газированной водой, налил большой стакан газировки, отдал все это клиенту, взял у него деньги и отсчитал ему сдачу. При таком уровне детализации разговор между двумя сотрудниками ресторана получится не просто очень скучным, но и ненужным. Оба работника и так понимают, что такое «сделать номер 3» и «оформить его». Им не приходится вдаваться во все эти детали, поскольку они их абстрагировали.

Использование параметров и возвращаемых значений

Как было показано на примерах с функциями из стандартной библиотеки, в программе можно задать значение функции и в ответ получить от функции другое значение. Например, при работе с функцией `toupper()` вы задаете для нее символ,

а функция возвращает вам этот символ, преобразованный в верхний регистр. Ваши собственные функции также могут получать и возвращать значения. Таким образом они могут обмениваться информацией с остальной программой.

Знакомство с программой Yes or No

Программа Yes or No задает игроку простые вопросы, на которые он должен отвечать. Сначала пользователь будет просто выбирать один из двух ответов: «Да» или «Нет». Затем программа конкретизируется, и пользователь должен ответить, хочет ли он сохранить игру. Результат выполнения программы непримечателен, в ней интересна именно реализация. Каждый вопрос задается новой функцией, которая обменивается информацией с функцией `main()`. Прогон программы продемонстрирован на рис. 5.2.

Рис. 5.2. Каждый вопрос задается отдельной функцией. Эти функции и функция main() обмениваются информацией (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `yes_or_no.cpp`.

```
// Программа Yes or No
// Демонстрирует работу с возвращаемыми значениями и параметрами
#include <iostream>
#include <string>
using namespace std;
char askYesNo1();
char askYesNo2(string question);
int main()
{
    char answer1 = askYesNo1();
    cout << "Thanks for answering: " << answer1 << "\n\n";
    char answer2 = askYesNo2("Do you wish to save your game?");
    cout<< "Thanks for answering: " << answer2 << "\n";
```

```
    return 0;
}
char askYesNo1()
{
    char response1;
    do
    {
        cout << "Please enter 'y' or 'n': ";
        cin >> response1;
    } while (response1 != 'y' && response1 != 'n');
    return response1;
}
char askYesNo2(string question)
{
    char response2;
    do
    {
        cout << question << " (y/n): ";
        cin >> response2;
    } while (response2 != 'y' && response2 != 'n');
    return response2;
}
```

Возврат значения

Можно возвращать значение от функции, чтобы передавать нужную информацию тому коду, который вызвал эту функцию. Для возврата значения необходимо задать возвращаемый тип, а затем возвратить от этой функции значение данного типа.

Указание возвращаемого типа

Первая функция, которую я объявляю, называется `askYesNo1()`, она возвращает значение `char`. Это понятно из прототипа функции, идущего перед функцией `main()`:

```
char askYesNo1();
```

Кроме того, это следует из определения функции, идущего после `main()`:

```
char askYesNo1()
```

Использование инструкции return

Функция `askYesNo1()` предлагает пользователю ввести символ `y` или `n`, причем продолжает предлагать это, пока пользователь не напишет один из двух этих символов. Как только пользователь введет `y` или `n`, функция завершится следующей строкой кода, которая вернет значение `response1`:

```
return response1;
```

Обратите внимание: значение `response1` относится к типу `char`. Оно должно относиться именно к этому типу, так как я обещал вернуть значение типа `char` и в прототипе функции, и в ее определении.

Функция завершается, как только встречает инструкцию return. Вполне допустимо, чтобы в функции было несколько инструкций return — это означает, что функция предусматривает несколько вариантов завершения.

ПРИЕМ

Вы не обязаны возвращать значение в инструкции return. Эта инструкция может использоваться сама по себе в функции, не возвращающей значения (то есть в функции с возвращаемым типом void), просто для завершения функции.

Использование возвращаемого значения

В функции main() я вызываю функцию с помощью следующей строки, присваивающей переменной answer1 возвращаемое значение данной функции:

```
char answer1 = askYesNo1();
```

Таким образом, переменная answer1 получает значение y или n в зависимости от того, какой символ пользователь введет в ответ на приглашение функции askYesNo1().

Далее в функции main() я вывожу значение answer1 на всеобщее обозрение.

Запись значений в параметры

Те значения, которые принимает функция, можно сообщить ей в виде параметров. Это наиболее распространенный способ записи информации в функцию.

Указание параметров

Вторая определяемая мной функция, askYesNo2(), принимает значение в виде параметра. Именно эта функция принимает значение типа string. Об этом можно узнать из прототипа функции, идущего еще до функции main():

```
char askYesNo2(string question);
```

СОВЕТ

Вы не обязаны использовать в прототипе имена параметров — достаточно будет включить только типы параметров. Например, далее приведен корректный прототип, объявляющий askYesNo2(), функцию с одним параметром string, возвращающую значение типа char:

```
char askYesNo2(string);
```

Пусть использовать имена прототипов в параметрах и не обязательно, лучше этим не пренебрегать. При наличии имен параметров код становится гораздо понятнее, так что затраченные на это небольшие усилия не пропадут зря.

Из заголовка функции askYesNo2() понятно, что эта функция принимает в качестве параметра объект string, имя этого параметра — question:

```
char askYesNo2(string question)
```

В определении функции имена параметров указывать необходимо — этим определение функции отличается от прототипа. Имя параметра внутри функции применяется для получения доступа к значению параметра.

ОСТОРОЖНО!

Типы параметров, указанные в прототипе функции, должны совпадать с типами параметров, перечисленными в ее определении. В противном случае вы получите гнусную ошибку компиляции.

Передача значений параметрам

Функция `askYesNo2()` — это усовершенствованный вариант `askYesNo1()`. Новая функция позволяет вам задать собственный персонализированный вопрос, передав в функцию строковое приглашение для ответа. В функции `main()` я вызываю `askYesNo2()` с помощью следующего кода:

```
char answer2 = askYesNo2("Do you wish to save your game?");
```

Эта инструкция вызывает функцию `askYesNo2()` и передает ей в качестве аргумента строковый литерал `"Do you wish to save your game?"`.

Использование значений параметров

Строковый литерал `"Do you wish to save your game?"` записывается в параметр `question` функции `askYesNo2()`; этот параметр действует, как любая переменная, применяемая в функции. Фактически я отображаю `question` с помощью следующей строки:

```
cout << question << " (y/n): ";
```

СОВЕТ

Следует отметить, что на внутрисистемном уровне здесь происходит и еще кое-что. Когда строковый литерал "Do you wish to save your game?" записывается в параметр question, создается объект string, равный этому строковому литералу, после чего этот объект string присваивается параметру question.

Как и `askYesNo1()`, функция `askYesNo2()` продолжает предлагать пользователю ввести символ до тех пор, пока пользователь не напишет `y` или `n`. Затем функция возвращает это значение и завершается.

Когда программа вновь переходит к функции `main()`, возвращенное значение `char` присваивается переменной `answer2`, после чего я вывожу результат на экран.

Понятие об инкапсуляции

Возможно, при работе с собственными функциями вы предпочтете обходиться без возвращаемых значений. Почему бы просто не задействовать переменные `response1` и `response2` после того, как мы вернемся в функцию `main()`? В том-то и дело, что это невозможно: переменные `response1` и `response2` не существуют вне той функции, в которой были определены. Фактически ни к одной переменной, создаваемой в функции, в частности ни к одному параметру функции, нельзя обратиться извне этой функции. Это очень положительный феномен, он называется *инкапсуляцией*. Инкапсуляция обеспечивает подлинную независимость одного самостоятельного фрагмента кода от другого, скрывая детали. Именно поэтому мы работаем с параметрами и возвращаемыми значениями — чтобы передавать только ту информацию, которой действительно необходимо обмениваться. Кроме того, вам не приходится отслеживать создаваемые вами переменные по всей программе, так как они

существуют лишь в пределах той функции, где были определены. Чем больше программа, тем очевиднее польза инкапсуляции.

Возможно, у вас сложилось впечатление, что инкапсуляция во многом похожа на абстрагирование. Действительно, у двух этих феноменов много общего. Инкапсуляция — это принцип абстрагирования. Абстрагирование помогает программисту не вдаваться в детали, тогда как инкапсуляция скрывает от вас детали. В качестве примера рассмотрим телевизионный пульт, на котором есть кнопки для увеличения и уменьшения громкости. Настраивая громкость передачи с помощью пульта, вы применяете абстрагирование, так как вам не требуется знать, что при этом происходит внутри телевизора. Теперь предположим, что на нашем телевизионном пульте есть 10 уровней громкости. С помощью пульта вы можете выбрать любой уровень громкости, «пролистав» по порядку все предыдущие, но не можете напрямую обратиться к тому или иному уровню громкости. Это означает, что вы не можете непосредственно задать конкретный уровень громкости. У вас в распоряжении всего две кнопки — «тише» и «громче», с их помощью вы рано или поздно выберете желаемый уровень громкости. При этом все конкретные значения громкости инкапсулированы, вы не можете выбирать их напрямую.

Понятие о переиспользовании ПО

Функции можно переиспользовать (брать и использовать снова) в других программах. Например, поскольку во многих играх вам может понадобиться получить от пользователя ответ «да» или «нет» на поставленный вопрос, можно написать функцию `askYesNo()` и задействовать ее во всех программах, которые вы напишете в дальнейшем. Поэтому, если вы напишете качественный код для текущего проекта, то сэкономите время и силы при создании не только данной игры, но и многих будущих разработок.

НА ПРАКТИКЕ

Изобретая велосипед, мы впустую тратим время, поэтому переиспользование ПО — применение имеющегося кода и других элементов в новых проектах — это обязательное умение для сотрудников всех компаний, занимающихся программированием игр. Среди достоинств переиспользования ПО можно назвать следующие.

- Повышение производительности труда. Переиспользуя код и другие уже имеющиеся элементы, например графический движок, можно выполнять проекты быстрее, затрачивая на них меньше усилий.
- Повышение качества ПО. Если игровая компания уже располагает хорошо протестированным блоком кода, например сетевым модулем, смело может его переиспользовать, так как этот код гарантированно будет исправен.
- Повышение производительности ПО. Если игровая компания располагает высокопроизводительным блоком кода, то, переиспользуя его, компания не только избавляется от необходимости изобретать велосипед, но и не рискует, что новый велосипед будет уступать по эффективности старому.

Чтобы переиспользовать написанный вами код, достаточно скопировать его из одной программы и вставить в другую. Однако есть и более оптимальный способ — большой игровой проект можно разделить на несколько файлов. Этот подход будет изучен в главе 10.

Работа с областями видимости

Область видимости переменной определяет, в каких частях вашей программы будет видна эта переменная. Применяя области видимости, можно ограничить доступ к переменным. Область видимости — ключевая концепция, связанная с инкапсуляцией. С помощью областей видимости удобно отграничивать друг от друга различные компоненты вашей программы, например функции.

Знакомство с программой Scoping

Программа Scoping демонстрирует обращение с областями видимости. В ней мы создадим три одноименные переменные, каждая из которых имеет свою область видимости. Программа отображает значения этих переменных, и вы можете убедиться, что, хотя эти переменные и имеют одинаковые имена, они независимы друг от друга. Результат выполнения программы показан на рис. 5.3.

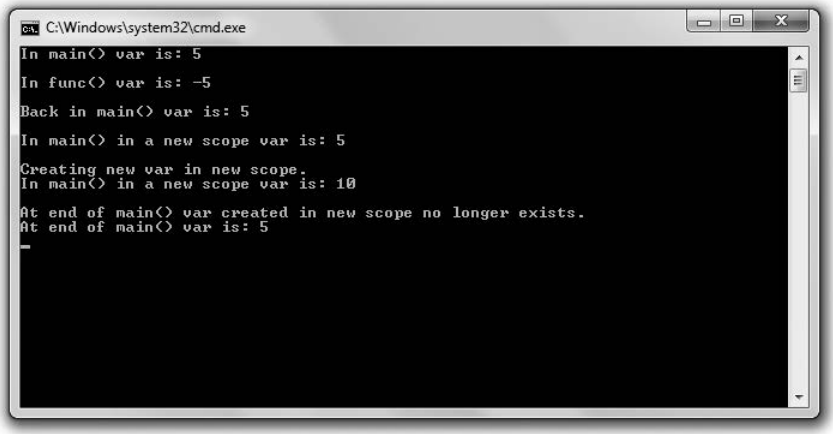


Рис. 5.3. Хотя три переменные в этой программе и являются одноименными, каждая из них уникальна и существует в собственной области видимости (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `scoping.cpp`.

```
// Программа Scoping
// Демонстрирует работу с областями видимости
#include <iostream>
using namespace std;
void func();
int main()
{
    int var = 5; // локальная переменная в функции main()
    cout << "In main() var is: " << var << "\n\n";
    func();
    cout <<"Back in main() var is: " << var << "\n\n";
```

```
    {
        cout << "In main() in a new scope var is: " << var << "\n\n";
        cout << "Creating new var in new scope.\n";
        int var = 10; // переменная в новой области видимости
                      // скрывает другую переменную, называемую var
        cout << "In main() in a new scope var is: " << var << "\n\n";
    }
    cout << "At end of main() var created in new scope no longer exists.\n";
    cout << "At end of main() var is: " << var << "\n";
    return 0;
}
void func()
{
    int var = -5; // локальная переменная в функции func()
    cout << "In func() var is: " << var << "\n\n";
}
```

Работа с отдельными областями видимости

Всякий раз, создавая с помощью фигурных скобок блок кода, вы создаете новую область видимости. Переменные, объявляемые внутри этой области, невидимы за ее пределами. Таким образом, переменные, определенные в функции, невидимы за пределами этой функции.

Переменные, определяемые внутри функции, называются *локальными переменными*, то есть они локализованы в своей функции. Именно таким образом функции инкапсулируются.

Мы уже рассмотрели на практике множество локальных переменных. В следующей строке кода я определю еще одну переменную, которая будет локальна в функции `main()`:

```
int var = 5; // локальная переменная в функции main()
```

В этой строке кода объявляется и инициализируется локальная переменная под названием `var`. В следующей строке кода я посылаю эту переменную в `cout`:

```
cout << "In main() var is: " << var << "\n\n";
```

Результат ожидаемый — на экране отображается число 5.

Далее я вызываю функцию `func()`. Войдя в эту функцию, я оказываюсь в отдельной области видимости, вне той области видимости, которая принадлежит функции `main()`. Отсюда я не могу получить доступ к переменной `var`, определенной в функции `main()`. Соответственно, когда в следующей строке кода я определяю переменную `var`, локальную для функции `func()`, эта переменная абсолютно автономна от переменной `var` из функции `main()`:

```
int var = -5; // локальная переменная в функции func()
```

Две одноименные переменные никак не влияют друг на друга — в этом и заключается красота областей видимости. Когда вы пишете функцию, вы можете не задумываться о том, встречаются ли в других функциях программы переменные с такими же именами, как в этой.

С помощью приведенной далее строки кода я отображаю значение переменной var из функции func(). Компьютер выводит на экран число -5:

```
cout << "In func() var is: " << var << "\n\n";
```

Дело в том, что компьютер находит в данной области видимости всего одну переменную с именем var — локальную переменную, определенную именно в этой функции.

Когда область видимости прекращается, все объявленные в ней переменные перестают существовать. Принято говорить, что они *выходят за пределы области видимости*. Итак, когда заканчивается функция func(), заканчивается и ее область видимости. Все переменные, определенные в функции func(), при этом уничтожаются. Это происходит и с переменной var, имеющей значение -5, которую я объявил в функции func().

После окончания функции func() управление программой возвращается к функции main(), причем функция main() продолжает выполняться ровно с того места, на котором остановилась. Далее выполняется следующая строка, отправляющая var в cout:

```
cout << "Back in main() var is: " << var << "\n\n";
```

На экране вновь отображается число 5 — это значение той переменной var, которая локальна для функции main().

Может возникнуть вопрос: а что было с переменной var из функции main(), пока мы работали с функцией func()? Да, эта переменная не была уничтожена, так как функция main() еще не закончилась. Код программы просто сделал небольшой крюк, заглянув в функцию func(). Когда программа ненадолго выходит из одной функции, чтобы выполнить другую, компьютер сохраняет состояние первой функции, запоминая то место, на котором остановилось выполнение, а также значения всех локальных переменных этой функции. Как только программа возвращается к первой функции, вся эта информация восстанавливается.

СОВЕТ

Параметры функций действуют точно так же, как локальные переменные.

Работа с вложенными областями видимости

Можно создать в имеющейся области видимости новую, вложенную область видимости. Вложенная область видимости заключается в фигурные скобки. Именно это я делаю далее в функции main():

```
{
    cout<< "Inmain() inanewscopevaris: " <<var<< "\n\n";
    cout << "Creating new var in new scope.\n";
    int var = 10; // переменная в новой области видимости
                  // скрывает другую переменную с именем var
    cout << "In main() in a new scope var is: " << var << "\n\n";
}
```

Эта новая область видимости относится к функции main() и является вложенной. В этой вложенной области видимости я первым делом отображаю переменную

var. Если та или иная переменная не объявлена в данной области видимости, то компьютер просматривает вложенные области видимости, расположенные на разных уровнях (по одному уровню за раз) в поисках запрошенной вами переменной. В нашем случае, поскольку переменная var не была объявлена в текущей области видимости, компьютер ищет в области видимости, расположенной на уровень выше (в той, где была определена функция main()), и находит переменную var. В результате программа отображает значение, принадлежащее именно этой переменной, то есть 5.

Вот какую операцию я выполняю в этой вложенной области видимости следующей: объявляю переменную var и инициализирую ее в значении 10. Теперь, когда я посылаю var в cout, на экране отображается 10. На этот раз компьютеру не приходится просматривать многоуровневую иерархию областей видимости, чтобы найти переменную var, — в данной области видимости существует своя локальная переменная var. Не волнуйтесь, та переменная var, которую я в самом начале объявил в функции main(), по-прежнему существует; просто в этой новой области видимости она скрыта локальной переменной var.

ОСТОРОЖНО!

Хотя и можно объявлять одноименные переменные в серии вложенных областей видимости, лучше так не делать — запутаетесь.

Далее, когда вложенная область видимости заканчивается, та переменная var, которая была равна 10, выходит из области видимости и прекращает существовать. А вот первая переменная var по-прежнему существует, поэтому, когда я в последний раз вывожу на экран переменную var в функции main(), программа выдает значение 5:

```
cout << "At end of main() var is: " << var << "\n";
```

СОВЕТ

Переменные, определенные в циклах for и while, инструкциях if и switch, не существуют вне «родных» структур. Функционально они напоминают переменные, объявленные во вложенной области видимости. Например, в следующем коде переменная i не существует вне цикла:

```
for(int i = 0; i < 10; ++i)
{
cout << i;
}
// переменная i не существует вне цикла
```

Однако будьте осторожны — некоторые старые компиляторы неправильно реализуют эту возможность стандарта C++. Рекомендую использовать при разработке IDE с современным компилятором, например Microsoft Visual Studio Express 2013 для Windows ПК. Пошаговые инструкции о том, как создать первый проект в этой IDE, приведены в приложении 1 «Создание первой программы на языке C++».

Использование глобальных переменных

Благодаря волшебству инкапсуляции все функции, рассмотренные нами ранее, надежно изолированы и независимы друг от друга. Единственный способ записать в них информацию — сделать это через параметры, а получить от них информацию

можно только через возвращаемые значения. Хотя нет, этим варианты не ограничиваются. Чтобы совместно использовать одну и ту же информацию в разных частях вашей программы, можно работать также с *глобальными переменными* (они доступны из любой части программы).

Знакомство с программой Global Reach

В программе Global Reach демонстрируется работа с глобальными переменными. Здесь будет показано, как обратиться к глобальной переменной из любой части вашей программы. Здесь вы также узнаете, как скрыть глобальную переменную в области видимости. Наконец, в этом разделе мы научимся изменять глобальную переменную из любой части программы. Результат выполнения программы показан на рис. 5.4.

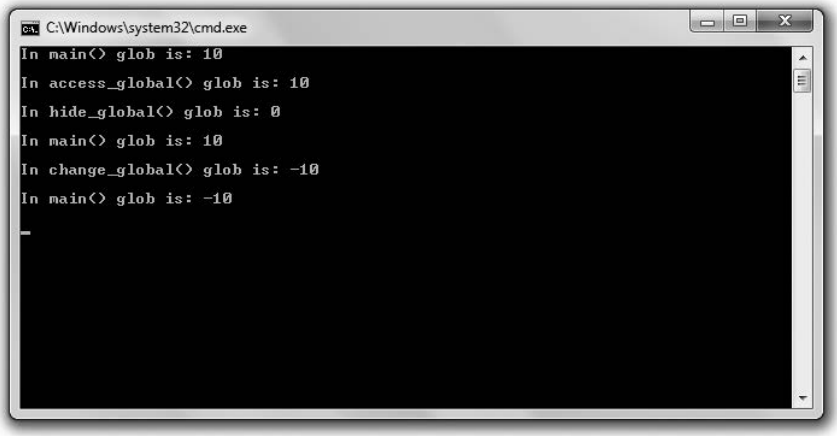


Рис. 5.4. Обращаться к глобальным переменным и изменять их можно из любой части программы, но такие переменные также могут быть скрыты в области видимости (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `global_reach.cpp`.

```
// Программа Global Reach
// Демонстрирует работу с глобальными переменными
#include <iostream>
using namespace std;
int glob = 10; // глобальная переменная
void access_global();
void hide_global();
void change_global();
int main()
{
    cout << "In main() glob is: " << glob << "\n\n";
    access_global();
    hide_global();
    cout << "In main() glob is: " << glob << "\n\n";
```

```
    change_global();
    cout << "In main() glob is: " << glob << "\n\n";
    return 0;
}
void access_global()
{
    cout << "In access_global() glob is: " << glob << "\n\n";
}
void hide_global()
{
    int glob = 0; // скрываем глобальную переменную glob
    cout << "In hide_global() glob is: " << glob << "\n\n";
}
void change_global()
{
    glob = -10; // изменяем глобальную переменную glob
    cout << "In change_global() glob is: " << glob << "\n\n";
}
```

Объявление глобальных переменных

Глобальные переменные определяются в файле программы вне каких-либо функций. Именно это я делаю в следующей строке, которая создает глобальную переменную `glob` и инициализирует ее в значении `10`:

```
int glob = 10; // глобальная переменная
```

Доступ к глобальным переменным

К глобальной переменной можно обратиться из любой точки программы. Чтобы в этом убедиться, отобразим переменную `glob` в функции `main()` с помощью следующей строки кода:

```
cout << "In main() glob is: " << glob << "\n\n";
```

Программа выводит на экран значение `10`, так как `glob`, будучи глобальной переменной, доступна в любой части программы. Чтобы дополнительно это продемонстрировать, далее я вызываю функцию `access_global()`, и компилятор выполняет в этой функции следующий код:

```
cout << "In access_global() glob is: " << glob << "\n\n";
```

Вновь видим на экране `10`. Это верно, поскольку в каждой функции я работаю с одной и той же переменной `glob`.

Скрывание глобальных переменных

Глобальную переменную, как и любую другую, можно скрыть в области видимости. Для этого просто нужно объявить в конкретной области видимости переменную с таким же именем, как и у глобальной. Именно это я и делаю далее, когда вызываю

функцию hide_global(). Основная строка в этой функции не изменяет переменную glob, она просто создает новую переменную glob, локальную для функции hide_global(). Локальная переменная glob скрывает одноименную глобальную переменную:

```
Int glob = 0; // скрываем глобальную переменную glob
```

В результате, когда в следующей строке функции hide_global() я посылаю переменную glob в cout, на экране отображается 0:

```
cout << "In hide_global() glob is: " << glob << "\n\n";
```

Глобальная переменная glob остается скрытой в области видимости функции hide_global(), пока эта функция не заканчивается.

Чтобы убедиться, что глобальная переменная всего лишь скрыта, но не изменена, далее я вновь отображаю glob в функции main() с помощью следующей строки:

```
cout << "In main() glob is: " << glob << "\n\n";
```

На экране опять видим 10.

ОСТОРОЖНО!

Хотя и можно объявлять в функции локальную переменную, одноименную глобальной переменной, действующей во всей программе, лучше так не делать — запутаетесь.

Изменение глобальных переменных

Можно не только обратиться к глобальной переменной из любой точки программы, но и изменить ее из любой части программы. Именно это я делаю далее, вызывая функцию change_global(). Основная строка функции присваивает значение -10 глобальной переменной glob:

```
glob = -10; // изменение глобальной переменной glob
```

Чтобы убедиться, что все сработало, я отображаю переменную в функции change_global() с помощью следующей строки:

```
cout << "In change_global() glob is: " << glob << "\n\n";
```

Далее, возвращаясь к функции main(), я посылаю glob в cout с помощью строки:

```
cout << "In main() glob is: " << glob << "\n\n";
```

Поскольку глобальная переменная glob изменилась, на экране видим значение -10.

Минимизация использования глобальных переменных

То, что можно что-то сделать, еще не означает, что так действительно стоит поступать. Это важное кредо программистов. Иногда технически операция возможна, но к ней лучше не прибегать. Классический пример такого случая — использование

глобальных переменных. Вообще с глобальными переменными программа становится более запутанной, так как зачастую бывает сложно отслеживать все изменяющиеся в ней значения. Поэтому следует максимально ограничить любое использование глобальных переменных.

Использование глобальных констант

В отличие от глобальных переменных, которые могут просто запутать вашу программу, *глобальные константы*, то есть константы, доступные из любой части программы, зачастую делают программу яснее. Глобальная константа объявляется примерно так же, как и глобальная переменная, — вне каких-либо функций. Поскольку вы объявляете константу, нужно использовать ключевое слово `const`. Например, в следующей строке кода определяется глобальная константа (предполагается, что такое объявление находится вне каких-либо функций). Глобальная константа называется `MAX_ENEMIES` и имеет значение `10`. К ней можно обратиться из любой части программы:

```
const int MAX_ENEMIES = 10;
```

ОСТОРОЖНО!

Точно так же, как и при работе с глобальными переменными, можно скрыть глобальную константу, объявив одноименную ей локальную константу. Правда, так лучше не делать, поскольку это может запутать.

Как именно глобальные константы помогают прояснить игровой код? Предположим, вы пишете игру-экшен, в которой хотите ограничить количество врагов, способных одновременно палить в беднягу игрока. Чтобы в таких случаях не приходилось повсюду использовать цифровой литерал, скажем, 10, можно определить глобальную константу `MAX_ENEMIES`, равную `10`. Теперь, видя в коде глобальную константу, вы точно знаете, что она означает.

Здесь существует один подводный камень: глобальные константы следует применять лишь в тех случаях, когда константное значение требуется вам в нескольких частях программы. Если константное значение нужно лишь в конкретной области видимости (например, в функции), то лучше использовать локальную, а не глобальную константу.

Использование аргументов, задаваемых по умолчанию

Если вы пишете функцию, один из параметров которой практически всегда получает одно и то же значение, можно избавить вызывающий компонент от необходимости постоянно указывать это значение. В таких случаях применяется *аргумент по умолчанию* — значение, присваиваемое параметру, если не указано иное. Рассмотрим конкретный пример. Допустим, у вас есть функция, определяющая вывод графики. Один из ее параметров может быть равен `bool fullScreen`, он сообщает программе, должна ли игра отображаться в полноэкранном или оконном

режиме. Если вы считаете, что такая функция обычно будет вызываться со значением `true` для параметра `fullScreen`, то можно сообщить этому параметру аргумент `true` как применяемый по умолчанию. Тогда нам не придется сообщать параметру `fullScreen` значение `true` при каждом вызове функции отображения в полноэкранном режиме.

Знакомство с программой Give Me a Number

Программа Give Me a Number запрашивает у пользователя два числа, относящихся к двум разным диапазонам. Эта функция вызывается всякий раз, когда мы предлагаем пользователю ввести число. Правда, при каждом вызове этой функции используется разное количество аргументов, поскольку для нижнего предела она имеет аргумент, задаваемый по умолчанию. Таким образом, вызывающий компонент может опустить аргумент, соответствующий нижнему пределу, тогда функция автоматически воспользуется значением, задаваемым по умолчанию. Результат выполнения этой программы показан на рис. 5.5.

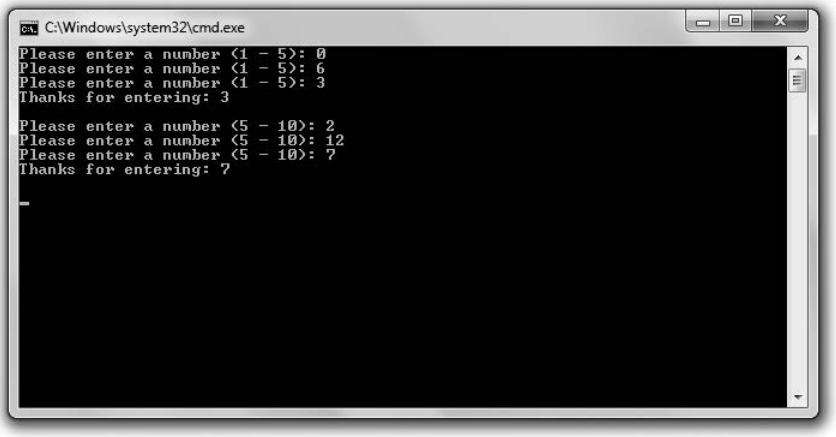


Рис. 5.5. В первый же раз, когда пользователю предлагается ввести число, для нижнего предела числового диапазона используется аргумент, задаваемый по умолчанию (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `give_me_a_number.cpp`.

```
// Программа Give Me a Number
// Демонстрирует работу с аргументами функций, задаваемыми по умолчанию
#include <iostream>
#include <string>
using namespace std;
int askNumber(int high, int low = 1);
int main()
{
    int number = askNumber(5);
    cout << "Thanks for entering: " << number << "\n\n";
```

```
    number = askNumber(10, 5);
    cout << "Thanks for entering: " << number << "\n\n";
    return 0;
}
int askNumber(int high, int low)
{
    int num;
    do
    {
        cout << "Please enter a number" << " (" << low << " - " << high << "): ";
        cin >> num;
    } while (num > high || num < low);
    return num;
}
```

Задание аргументов, применяемых по умолчанию

Функция `askNumber()` имеет два параметра: `high` и `low`. Это понятно из прототипа функции:

```
int askNumber(int high, int low = 1);
```

Обратите внимание: второй параметр `low` выглядит так, как будто ему присвоено значение. В каком-то смысле так и есть: `1` — это аргумент, задаваемый по умолчанию, то есть если при вызове функции параметру `low` не сообщается никакого значения, то он принимает значение 1. Чтобы задать для параметра аргумент по умолчанию, нужно поставить после параметра знак = и сообщаемое ему значение.

ОСТОРОЖНО!

Указав в списке параметров хотя бы один аргумент, задаваемый по умолчанию, потребуется указать такие аргументы и для всех остальных параметров, имеющихся в списке. Таким образом, следующий прототип функции допустим:

```
void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);
```

а вот этот — нет:

```
void setDisplay(int width, int height, int depth = 32, bool fullScreen);
```

Кстати, задаваемый по умолчанию аргумент не повторяется в определении функции, что хорошо видно в определении функции `askNumber()`:

```
int askNumber(int high, int low)
```

Присваивание параметрам аргументов, задаваемых по умолчанию

Функция `askNumber()` требует от пользователя указать число в диапазоне от верхнего до нижнего предела. Функция повторяет это требование до тех пор, пока пользователь не введет число, относящееся к этому диапазону, а затем возвращает это число. В первый раз я вызываю эту функцию из функции `main()`, вот так:

```
int number = askNumber(5);
```

В результате выполнения этого кода параметру high в функции askNumber() присваивается значение 5. Поскольку я не указываю никакого значения для второго параметра, low, он получает задаваемое по умолчанию значение 1. Таким образом, функция предлагает пользователю указать число в диапазоне от 1 до 5.

ОСТОРОЖНО!

Когда вы вызываете функцию с аргументами, задаваемыми по умолчанию, в случае пропуска одного аргумента необходимо опустить и аргументы для всех остальных параметров. Например, для функции с прототипом:

```
void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);
```

допустимый вызов записывается так:

```
setDisplay(1680, 1050);
```

а недопустимый, например, так:

```
setDisplay(1680, 1050, false);
```

Как только пользователь введет число из допустимого диапазона, функция askNumber() возвращает это значение и завершается. Программа возвращается к функции main(), где это значение присваивается переменной number и отображается.

Переопределение аргументов, задаваемых по умолчанию

Далее я вновь вызываю функцию askNumber() с помощью следующего кода:

```
number = askNumber(10, 5);
```

На этот раз я сообщаю значение параметру low — 5. Это совершенно нормально: можно сообщить аргумент любому параметру, уже имеющему аргумент, заданный по умолчанию, причем значение, сообщаемое вами при этом, переопределит (заменит собой) заданное по умолчанию. В данном случае параметр low получает значение 5.

Таким образом, пользователю предлагается ввести число в диапазоне от 5 до 10. Как только пользователь введет подходящее число, функция askNumber() возвращает это значение и завершается. Программа возвращается к функции main(), где это значение присваивается переменной number и отображается.

Перегрузка функций

Итак, мы рассмотрели, как нужно указывать список параметров и единственный возвращаемый тип для каждой из функций, которую вы пишете. Но что делать, если вам требуется более гибкая функция — такая, которая может принимать различные наборы аргументов? Например, нужно написать функцию, которая применяет трехмерные трансформации к множеству вершин, представляемых в виде чисел с плавающей запятой (float). При этом вы также хотите, чтобы эта функция работала и с целыми числами (int). Чтобы не писать две отдельные функции

с разными именами специально для первой и специально для второй задачи, можно применить перегрузку функции, то есть приспособить отдельно взятую функцию к обработке разных списков параметров. Таким образом, вы сможете вызывать одну и ту же функцию, а вершины сообщать ей либо как `float`, либо как `int`.

Знакомство с программой Triple

Программа Triple утраивает значение 5 и последовательность символов `gamer`. Программа решает обе эти задачи с помощью единственной функции, которая перегружена для работы с двумя разными типами аргументов: `int` и `string`. На рис. 5.6 показан примерный прогон программы.

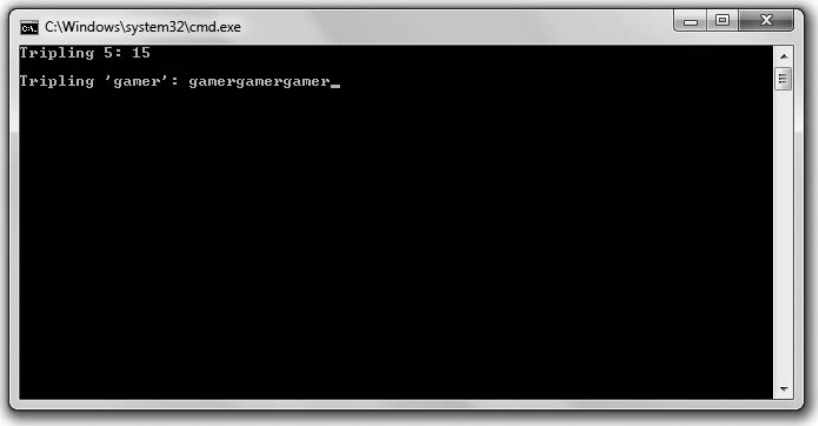


Рис. 5.6. Перегрузка позволяет утраивать значения двух разных типов с помощью одной и той же функции (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `triple.cpp`.

```
// Программа Triple
// Демонстрирует перегрузку функций
#include <iostream>
#include <string>
using namespace std;
int triple(int number);
string triple(string text);
int main()
{
    cout << "Tripling 5: " << triple(5) << "\n\n";
    cout << "Tripling 'gamer': " << triple("gamer");
    return 0;
}
int triple(int number)
{
    return (number * 3);
}
```

```
string triple(string text)
{
    return (text + text + text);
}
```

Создание перегруженных функций

Для создания перегруженной функции просто нужно написать несколько определений конкретной функции, указав в них одно и то же имя и разные списки параметров. В программе Triple я пишу два определения для функции `triple()`, в каждом из которых в качестве единственного аргумента указывается свой тип. Вот прототипы функций:

```
int triple(int number);
string triple(string text);
```

Первый принимает аргумент `int` и возвращает `int`. Второй принимает объект `string` и возвращает объект `string`.

Вы можете убедиться, что в обоих определениях функций я на выходе утраиваю то значение, которое было на входе. В первом случае я возвращаю целое число `int`, умноженное на 3. Во втором случае возвращаю отправленную строку, записанную трижды.

ОСТОРОЖНО!

Чтобы реализовать перегрузку функций, необходимо написать несколько определений одной и той же функции с разными списками параметров. Обратите внимание: здесь я вообще не упоминаю возвращаемые типы. Дело в том, что если вы напишете два определения функции, различающиеся лишь возвращаемым типом, то получите ошибку компиляции. Например, в программе нельзя поставить сразу два следующих прототипа:

```
int Bonus(int);
float Bonus(int);
```

Вызов перегруженных функций

Перегруженные функции можно вызывать точно таким же образом, как и любые другие: по имени, указывая при этом ряд допустимых аргументов. Но при работе с перегруженными функциями компилятор (на основе значений аргументов) решает, какое из определений вызвать. Например, когда я вызываю функцию `triple()` с помощью следующей строки кода, используя в качестве аргумента число `int`, компилятор вызовет именно то определение, в котором указан тип `int`. В результате функция вернет `int` 15:

```
cout << "Tripling 5: " <<triple(5) << "\n\n";
```

Снова вызываю функцию `triple()` следующей строкой:

```
cout << "Tripling 'gamer': " << triple("gamer");
```

Поскольку во втором случае в качестве аргумента используется строковый литерал, компилятор вызывает именно то определение функции, в котором содержится объект `string`. Функция возвращает объект `string`, равный `gamergamergamer`.

Подстановка вызова функций

Вызов функции чреват небольшими издержками для производительности. Как правило, ими можно пренебречь, так как они действительно очень невелики. Однако в случае с крошечными функциями (например, состоящими из одной или двух строк) иногда можно ускорить работу программы, замещая вызовы таких функций (выполняя их подстановку). При *подстановке* вы приказываете компилятору делать копию функции везде, где он ее встречает. В результате при каждом вызове функции программе не приходится перепрыгивать к уже пройденному месту в коде.

Знакомство с программой Taking Damage

Программа Taking Damage имитирует урон, наносимый здоровью персонажа, когда он подвергается радиационному облучению. В каждом раунде персонаж теряет половину здоровья. К счастью, в программе предусмотрено всего три прогона, поэтому наш несчастный персонаж выживает. В программе подставляется маленькая функция, заново вычисляющая уровень здоровья персонажа. Результат ее выполнения показан на рис. 5.7.

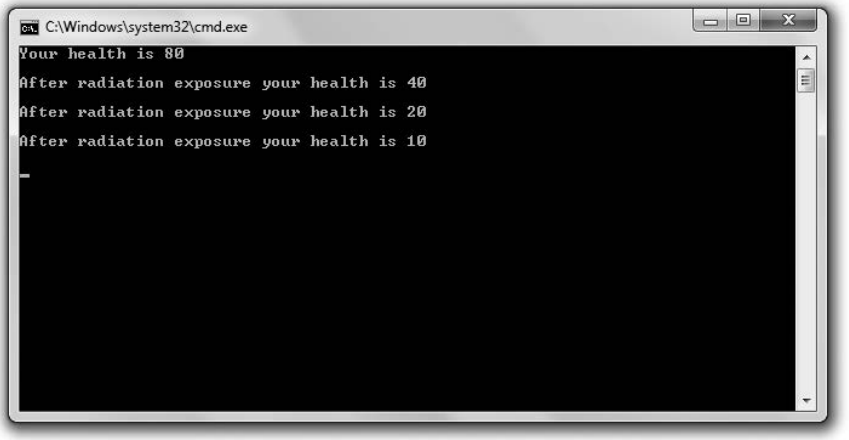


Рис. 5.7. Персонаж медленно, но верно угасает по мере того, как его здоровье ухудшается под действием подставляемой функции (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `taking_damage.cpp`.

```
// Программа Taking Damage
// Демонстрирует подстановку функций
#include <iostream>
int radiation(int health);
using namespace std;
int main()
```

```
{
    int health = 80;
    cout << "Your health is " << health << "\n\n";
    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";
    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";
    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";
   return 0;
}
inline int radiation(int health)
{
    return (health / 2);
}
```

Задание функций для подстановки

Чтобы пометить функцию как предназначенную для подстановки, просто поставьте перед ее определением ключевое слово `inline`. Именно так я поступаю в следующей функции:

```
inline int radiation(int health)
```

Обратите внимание на то, что при объявлении функции ключевое слово `inline` не используется:

```
int radiation(int health);
```

Помечая функцию ключевым словом `inline`, вы приказываете компилятору скопировать функцию непосредственно в вызывающий код. Экономится ресурс, который пришлось бы потратить на вызов функции, поскольку программе не приходится переключаться на другой участок вашего кода. При работе с небольшими функциями такой прием позволяет повысить производительность.

Однако подстановка функций — не панацея для обеспечения производительности. На самом деле при сплошной подстановке функций производительность программы может даже ухудшиться, поскольку в таком случае возникает множество копий функции и очень существенно возрастает потребление памяти.

ПОДСКАЗКА

Подставляя функцию, вы фактически запрашиваете у компилятора право на подстановку, и именно машина принимает окончательное решение о подстановке. Если машина сочтет, что подстановка негативно скажется на производительности, то не будет ее выполнять.

Вызов подставленных функций

Вызов подставленной функции не отличается от вызова самой обычной функции. Рассмотрим вызов функции `radiation()`, упоминавшийся ранее:

```
health = radiation(health);
```

Эта строка присваивает переменной `health` половину ее исходного значения.

Если предположить, что компилятор примет мой запрос на подстановку, то этот код не приведет к вызову функции. Компилятор просто поставит код, уполовинивающий значение переменной `health`, прямо в этой точке программы. На самом деле именно так компилятор и поступает при всех трех вызовах функции.

НА ПРАКТИКЕ

Конечно, программиста-игровика хлебом не корми — дай поволноваться о производительности игры. Но существует реальная опасность чрезмерно увлечься разгоном программы. На самом деле программисту следует сначала добиться, чтобы игра работала хорошо, а уже затем совершенствовать ее, повышая производительность. На этапе такой доработки программист будет профилировать программу, пользуясь при этом специальной утилитой — профилировщиком. Профилировщик анализирует, на что именно программа тратит время во время работы. Если программист замечает узкие места, то может подумать и об оптимизации, например о подстановке функций.

Знакомство с программой «Безумные библиотекари»

В игре «Безумные библиотекари» пользователь сочиняет историю. Игрок указывает имя персонажа, существительное во множественном числе, число, часть тела и глагол. Программа принимает всю эту информацию и составляет из нее персонализованную историю. На рис. 5.8 показан примерный прогон программы.

Рис. 5.8. После того как пользователь предоставит всю необходимую информацию, программа напишет настоящий шедевр (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 5, имя файла — `mad_lib.cpp`.

Подготовка программы

Как обычно, я начинаю программу с пары комментариев, а также включаю в нее нужные файлы:

```
// Программа Mad-Lib
// Формирует краткую историю на основе пользовательского ввода
#include <iostream>
#include <string>
using namespace std;
string askText(string prompt);
int askNumber(string prompt);
void tellStory(string name, string noun, int number, string bodyPart, string verb);
```

По прототипам функций несложно догадаться, что, кроме функции `main()`, у меня здесь будет еще три функции: `askText()`, `askNumber()` и `tellStory()`.

Функция main()

Функция `main()` вызывает все остальные функции. Так, она вызывает функцию `askText()`, чтобы получить от пользователя имя, существительное во множественном числе, часть тела и глагол. Она вызывает функцию `askNumber()`, чтобы получить от пользователя число. Наконец, вызываемая ею функция `tellStory()` принимает всю сообщенную пользователем информацию и генерирует на ее основе сюжет:

```
int main()
{
    cout << "Welcome to Mad Lib.\n\n";
    cout << "Answer the following questions to help create a new story.\n";
    string name = askText("Please enter a name: ");
    string noun = askText("Please enter a plural noun: ");
    int number = askNumber("Please enter a number: ");
    string bodyPart = askText("Please enter a body part: ");
    string verb = askText("Please enter a verb: ");
    tellStory(name, noun, number, bodyPart, verb);
    return 0;
}
```

Функция askText()

Функция `askText()` получает от пользователя последовательность символов. Эта функция универсальна, принимает параметр типа `string`, выдаваемый пользователю в качестве приглашения. Поэтому я могу вызывать всего одну эту функцию, запрашивая у пользователя самые разные информационные фрагменты, в частности имя, существительное во множественном числе, часть тела и глагол:

```
string askText(string prompt)
{
    string text;
    cout << prompt;
    cin >> text;
    return text;
}
```

ОСТОРОЖНО!

Учтите, что такой простой вариант использования cin работает только со сплошными строками (не содержащими знаков табуляции или пробелов). Поэтому, когда пользователю предлагается указать часть тела, он может ввести bellybutton (пуп), но medulla oblongata (продолговатый мозг) может вызвать в программе проблему.

С такой проблемой можно справиться, но для этого потребовалось бы обсудить тему потоков (streams), которая выходит за рамки нашей книги. Итак, пользуйтесь cin, как указано ранее, но помните об ограничениях этого объекта.

Функция askNumber()

Функция `askNumber()` получает от пользователя целое число. Хотя я могу использовать ее в программе всего один раз, она мне пригодится, так как принимает параметр типа `string` и выдает пользователю соответствующее приглашение:

```
int askNumber(string prompt)
{
    int num;
    cout << prompt;
    cin >> num;
    return num;
}
```

Функция tellStory()

Функция `tellStory()` принимает всю информацию, предоставленную пользователем, и на ее основе формирует персонализированный сюжет:

```
void tellStory(string name, string noun, int number, string bodyPart, string verb)
{
    cout << "\nHere's your story:\n";
    cout << "The famous explorer ";
    cout << name;
    cout << " had nearly given up a life-long quest to find\n";
    cout << "The Lost City of ";
    cout << noun;
    cout << " when one day, the ";
    cout << noun;
    cout << " found the explorer.\n";
    cout << "Surrounded by ";
    cout << number;
    cout << " " << noun;
    cout << ", a tear came to ";
    cout << name << "'s ";
    cout << bodyPart << ".\n";
    cout << "After all this time, the quest was finally over. ";
    cout << "And then, the ";
    cout << noun << "\n";
    cout << "promptly devoured ";
```

```
    cout << name << ". ";
    cout << "The moral of the story? Be careful what you ";
    cout << verb;
    cout << " for.";
}
```

Резюме

В этой главе мы изучили следующий материал.

- Функции позволяют разбивать программу на удобные в управлении фрагменты.
- Чтобы объявить функцию, можно, в частности, написать ее прототип — код, в котором перечисляются имя функции, ее возвращаемое значение, а также типы параметров, принимаемых этой функцией.
- Определить функцию означает написать весь код, который обеспечивает ее работу.
- Инструкция `return` может использоваться для возврата значения от функции. Кроме того, инструкция `return` может использоваться для завершения функции, если данная функция имеет возвращаемый тип `void`.
- Область видимости переменной определяет, в каких частях вашей программы будет видна данная переменная.
- Глобальные переменные доступны из любой части программы. Не следует увлекаться использованием глобальных переменных.
- Глобальные константы доступны из любой части программы. Благодаря использованию глобальных констант код программы становится чище.
- Аргументы, задаваемые по умолчанию, присваиваются параметру, если при вызове функции для этого параметра не было указано никакого значения.
- Перегрузка функции заключается в создании нескольких определений для одной и той же функции, причем при каждом конкретном определении функция принимает отдельный набор параметров.
- Подстановка функции — это процесс, при котором мы приказываем компилятору скопировать функцию и вставить ее копии во все точки кода, где она вызывается. При подстановке очень небольших функций зачастую удается повысить производительность программы.

Вопросы и ответы

1. *Зачем писать функции?*

   Функции позволяют разбивать программу на фрагменты, обладающие внутренней логикой. Как правило, такие фрагменты невелики и лучше управляемы, чем цельная монолитная программа.

2. *Что такое инкапсуляция?*

   По сути, инкапсуляция предназначена для разграничения сущностей. Так, благодаря инкапсуляции функций объявляемые в них переменные недоступны извне этих функций.

3. *В чем заключается разница между аргументом и параметром?*

   Аргумент — это информационная единица, передаваемая функции при вызове. Параметр используется при определении функции для приема значений, передаваемых этой функции.

4. *Может ли функция содержать более одной инструкции return?*

   Конечно. Несколько инструкций `return` в функции могут понадобиться для того, чтобы указать различные точки, в которых данная функция может завершиться.

5. *Что такое локальная переменная?*

   Это переменная, определенная в конкретной области видимости. Все переменные, определяемые в функциях, являются локальными (в пределах своих функций).

6. *Что значит скрыть переменную?*

   Мы скрываем переменную, если объявляем во внутренней области видимости переменную с таким именем, как и у другой переменной, находящейся во внешней области видимости. В результате, будучи во внутренней области видимости, вы не сможете обратиться к одноименной «внешней» переменной по ее имени.

7. *Когда переменная выходит из области видимости?*

   Переменная выходит из области видимости, когда область видимости, в которой она была создана, заканчивается.

8. *Что происходит, когда переменная выходит из области видимости?*

   Переменная прекращает существовать.

9. *Что такое вложенная область видимости?*

   Это область видимости, созданная в пределах другой, уже имеющейся области видимости.

10. *Должен ли аргумент иметь такое же имя, как и параметр, которому он сообщается?*

    Нет. Можете использовать любые удобные имена. От вызова функции к самой функции передается только значение.

11. *Можно ли написать одну функцию, которая вызывает другую?*

    Конечно. На самом деле, когда вы пишете любую функцию, вызываемую из `main()`, вы делаете именно это. Можно писать и другие функции (кроме `main()`), которые вызывают иные функции.

12. *Что такое профилирование кода?*

    Это процесс, в ходе которого определяется, сколько процессорного времени тратится на использование различных компонентов программы.

13. *Зачем профилировать код?*

    Чтобы выявить в программе узкие места. Иногда бывает целесообразно пересмотреть некоторые фрагменты вашего кода и попытаться оптимизировать их.

14. *Когда программисты профилируют код?*

    Обычно это делается на завершающем этапе игрового проекта.

15. *Что такое преждевременная оптимизация?*

    Это попытка приступить к оптимизации кода на слишком раннем этапе разработки. Как правило, оптимизировать код целесообразно лишь ближе к концу работы над проектом.

Вопросы для обсуждения

1. Как инкапсуляция функций способствует повышению качества программ?
2. Как глобальные переменные могут запутывать код?
3. Как глобальные константы помогают сделать код чище?
4. Каковы достоинства и недостатки оптимизации кода?
5. Какую пользу приносит игровой индустрии переиспользование программного кода?

Упражнения

1. Найдите ошибку в следующем прототипе:

```
int askNumber(int low = 1, int high);
```

2. Перепишите программу «Виселица» из главы 4, на этот раз с использованием функций. Сделайте функцию, которая будет принимать от пользователя его вариант, а также другую функцию, которая будет определять, соответствует ли пользовательский вариант загаданному слову.
3. С помощью аргументов, задаваемых по умолчанию, напишите функцию, которая запрашивает у пользователя число и возвращает это число. Функция должна принимать строковое приглашение от вызывающего кода. Если вызывающий код не предоставляет такого приглашения, то функция должна использовать обобщенное приглашение. Затем с помощью перегрузки напишите функцию, обеспечивающую такой же результат.

6 Ссылки. Игра «Крестики-нолики»

В ссылках нет ничего сложного, однако значение их огромно. В этой главе мы подробно поговорим о ссылках и о том, как они помогают писать более эффективный игровой код. В частности, вы научитесь:

- создавать ссылки;
- обращаться к значениям, на которые указывают ссылки, и изменять эти значения;
- передавать ссылки функциям, чтобы изменять значения аргументов или повышать эффективность;
- возвращать ссылки от функций, чтобы повышать эффективность или изменять значения.

Использование ссылок

Ссылка является синонимом для переменной, на которую она ссылается. Все операции, которые вы совершаете со ссылкой, применяются и к переменной, на которую она указывает. Ссылку можно сравнить с прозвищем переменной, ее вторым именем. В первой программе из этой главы будет рассмотрено, как создавать ссылки. Далее в нескольких следующих программах я покажу целесообразность использования ссылок и то, каким образом они помогают улучшить игровые программы.

Знакомство с программой Referencing

Программа Referencing демонстрирует работу со ссылками. Программа объявляет и инициализирует переменную, в которой записывается счет, а затем создает ссылку, указывающую на эту переменную. Программа отображает счет, используя переменную и ссылку и тем самым демонстрируя, что в обоих случаях происходит обращение к одному и тому же значению. Программа проиллюстрирована на рис. 6.1.

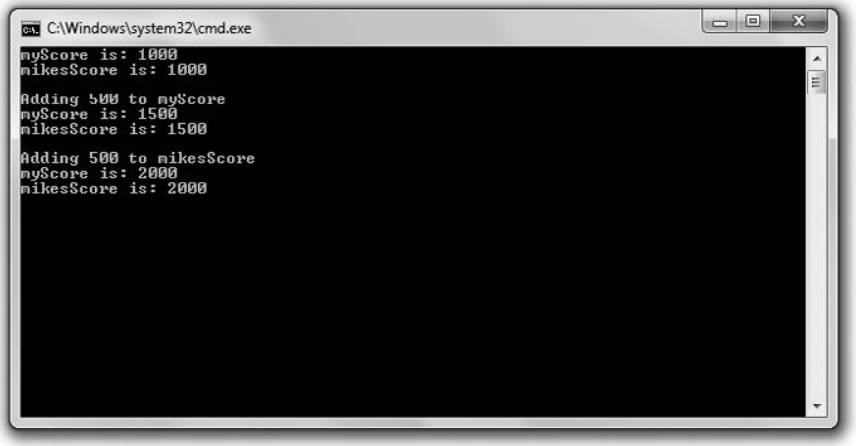


Рис. 6.1. Переменная myScore и ссылка mikesScore — это два названия одного и того же значения (набранных очков) (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 6, имя файла — `referencing.cpp`.

```
// Программа Referencing
// Демонстрирует работу со ссылками
#include <iostream>
using namespace std;
int main()
{
    int myScore = 1000;
    int& mikesScore = myScore; // создаем ссылку
    cout << "myScore is: " << myScore << "\n";
    cout << "mikesScore is: " <<mikesScore << "\n\n";
    cout << "Adding 500 to myScore\n";
    myScore += 500;
    cout << "myScore is: " << myScore << "\n";
    cout << "mikesScore is: " << mikesScore << "\n\n";
    cout << "Adding 500 to mikesScore\n";
    mikesScore += 500;
    cout << "myScore is: " << myScore << "\n";
    cout << "mikesScore is: " << mikesScore << "\n\n";
    return 0;
}
```

Создание ссылок

Первым делом в функции `main()` я создаю переменную, в которой буду вести счет:

```
int myScore = 1000;
```

Затем создаю ссылку, указывающую на переменную `myScore`:

```
int& mikesScore = myScore; // создание ссылки
```

В этой строке создается и инициализируется ссылка mikesScore, указывающая на переменную myScore. mikesScore — это псевдоним myScore. Ссылка mikesScore не содержит собственного целочисленного значения int, она — просто еще один способ доступа к тому значению int, которое записано в переменной myScore.

Чтобы объявить и инициализировать ссылку, для начала укажем тип значения, на которое будет указывать ссылка. После типа ставится ссылочный оператор (&), за ним идут имя ссылки, знак = и, наконец, та переменная, на которую указывает ссылка.

ПРИЕМ

Иногда программисты ставят перед именем ссылки букву r, чтобы не забывать, что это ссылка (reference). Так, программист может включить в код следующие строки:

```
int playerScore = 1000;
int& rScore = playerScore;
```

Чтобы лучше представлять себе ссылки, можно сравнить их с прозвищами или кличками. Допустим, у вас есть друг Юджин, который (кто бы сомневался) просит обращаться к нему по прозвищу Гибби (конечно, то еще прозвище, но если Юджин так хочет...). Итак, вы с другом приходите на вечеринку и можете называть его как Юджином, так и Гибби. Ваш друг — это всего один человек, но это не мешает вам обращаться к нему по-разному. Именно так взаимосвязаны переменная и ссылка на эту переменную. Можно получить значение, хранимое в переменной, применив либо имя этой переменной, либо имя ссылки, указывающей на данную переменную. Однако, как бы то ни было, старайтесь не называть переменную Eugene — так будет лучше.

ОСТОРОЖНО!

Поскольку ссылка всегда должна указывать на какое-то значение, ссылку нужно инициализировать прямо при объявлении. В противном случае вы получите ошибку компиляции. Например, в следующей строке содержится грубая ошибка:

```
int& mikesScore; // никогда так не делайте!
```

Доступ к значениям по ссылкам

Далее я отправляю в cout и myScore, и mikesScore:

```
cout << "myScore is: " << myScore << "\n";
cout << "mikesScore is: " << mikesScore << "\n\n";
```

Обе строки кода дают значение 1000, поскольку и переменная, и ссылка обращаются к одному и тому же фрагменту памяти, в котором хранится число 1000. Подчеркиваю: значение всего одно и оно хранится в переменной myScore. Ссылка mikesScore — просто еще один способ обратиться к этому значению.

Изменение значений через указывающие на них ссылки

Далее я увеличиваю значение переменной myScore на 500:

```
myScore += 500;
```

Когда я посылаю myScore в cout, на экран выводится значение 1500, как и ожидалось. Когда я посылаю mikesScore в cout, на экран также выводится значение 1500. Опять же дело в том, что mikesScore — просто другое название переменной myScore. В сущности, я дважды посылаю в cout одну и ту же переменную.

Далее я увеличиваю значение mikesScore на 500:

```
mikesScore += 500;
```

Поскольку mikesScore — просто другое название переменной myScore, данная строка кода увеличивает значение myScore на 500. Поэтому, когда я посылаю myScore в cout, на экран выводится значение 2000. Когда я посылаю mikesScore в cout, на экран также выводится значение 2000.

ОСТОРОЖНО!

Ссылка всегда указывает на ту переменную, вместе с которой она была инициализирована. Перенаправить ссылку на другую переменную нельзя, поэтому, в частности, следующий код может дать непредвиденный результат:

```
int myScore = 1000;
int& mikesScore = myScore;
int larrysScore = 2500;
mikesScore = larrysScore; // возможен не тот результат, на который вы рассчитывали!
```

Строка mikesScore = larrysScore; не переприсваивает ссылку mikesScore;, чтобы она относилась к larrysScore, так как переприсвоить ссылку нельзя. Однако, как мы уже знаем, mikesScore — это фактически псевдоним myScore. Поэтому код mikesScore = larrysScore; эквивалентен myScore = larrysScore;. Таким образом, переменной myScore присваивается значение 2500. После всех этих манипуляций переменная myScore становится равна 2500, а ссылка mikesScore по-прежнему указывает на myScore.

Передача ссылок для изменения аргументов

Теперь, рассмотрев, как работают ссылки, можно задать вопрос: а зачем они вообще могут потребоваться? Дело в том, что переменные очень удобны при передаче переменных функциям: когда вы передаете переменную функции, функция получает копию переменной. Это означает, что оригинал переданной вами переменной (он называется *переменной-аргументом*) не может быть изменен. Иногда нам как раз это и нужно, поскольку в данной ситуации переменная-аргумент не может быть изменена, следовательно, о ней можно не беспокоиться. Но в других случаях бывает необходимо изменить переменную-аргумент изнутри той функции, которой она была передана. Это можно сделать с помощью ссылок.

Знакомство с программой Swap

В программе Swap определяются две переменные. В одной переменной будет содержаться мой неказистый результат, а в другой — поразительно высокий ваш. После отображения этих результатов программа вызывает функцию, которая меняет наши очки местами. Но, поскольку в функцию посылаются лишь копии набранных результатов, сами переменные-аргументы, в которых записаны эти очки, не изменяются. Далее программа вызывает еще одну функцию перестановки. На этот раз с помощью ссылок мы успешно меняем местами значения переменных-аргументов — мне достается большой результат, а вам — маленький. На рис. 6.2 программа показана в действии.

Рис. 6.2. Благодаря передаче ссылок функция goodSwap() может менять переменные-аргументы (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 6, имя файла — `swap.cpp`.

```
// Программа Swap
// Демонстрирует передачу ссылок для изменения переменных-аргументов
#include <iostream>
using namespace std;
void badSwap(int x, int y);
void goodSwap(int& x, int& y);
int main()
{
    int myScore = 150;
    int yourScore = 1000;
    cout << "Original values\n";
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";
    cout << "Calling badSwap()\n";
    badSwap(myScore, yourScore);
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";
```

```
    cout << "Calling goodSwap()\n";
    goodSwap(myScore, yourScore);
    cout<< "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n";
    return 0;
}
void badSwap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
void goodSwap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Передача по значению

Объявив и инициализировав переменные myScore и yourScore, я посылаю их в cout. Как и следовало ожидать, на экране отображаются значения 150 и 1000. Далее я вызываю функцию badSwap().

При указании параметров таким образом, как это делалось ранее (как обычные переменные, а не как ссылки), вы сообщаете программе, что аргумент для данного параметра будет *передаваться по значению*. Это означает, что параметр получит *копию* переменной-аргумента и у него не будет доступа к самой переменной-аргументу (оригиналу). Просмотрев заголовок функции badSwap(), мы сразу видим, что при вызове этой функции оба ее аргумента передаются по значению.

```
void badSwap(int x, int y)
```

Таким образом, когда я вызываю функцию badSwap() с помощью следующей строки кода, копии переменных myScore и yourScore передаются параметрам x и y соответственно:

```
badSwap(myScore, yourScore);
```

Здесь переменная x получает значение 150, а переменная y — значение 1000. Поэтому, что бы я ни делал с x и y в функции badSwap(), мои действия никак не скажутся на переменных myScore и yourScore.

Когда выполнится внутренняя логика функции badSwap(), параметры x и y поменяются значениями: x станет равен 1000, а y — 150. Однако, когда эта функция завершится, оба параметра, x и y, выйдут из области видимости и прекратят существование. Затем управление программой вернется к функции main(), где переменные myScore и yourScore не менялись. Когда после этого я посылаю переменные myScore и yourScore в cout, на экран вновь выводятся значения 150 и 1000. Как ни жаль, у меня снова стало мало очков, зато у вас — много.

Передача по ссылке

Функции можно предоставить доступ к переменной-аргументу, передав этой функции в качестве параметра ссылку на переменную-аргумент. В результате все действия, которые мы совершаем с этим параметром, будут применяться и к переменной-аргументу. Для *передачи по ссылке* параметр сначала нужно объявить как ссылку.

При вызове функции `goodSwap()` оба ее аргумента передаются по ссылке — в этом можно убедиться, заглянув в заголовок данной функции:

```
void goodSwap(int& x, int& y)
```

Таким образом, когда я вызываю функцию `goodSwap()` в следующей строке кода, параметр `x` будет относиться к `myScore`, а параметр `y` — к `yourScore`:

```
goodSwap(myScore, yourScore);
```

Итак, `x` — это просто второе имя `myScore`, а `y` — второе имя `yourScore`. Когда функция `goodSwap()` выполнится и параметры `x` и `y` поменяются значениями, на самом деле произойдет обмен значениями между переменными `myScore` и `yourScore`.

Когда эта функция завершится, управление программой вновь перейдет к функции `main()`, где я посылаю переменные `myScore` и `yourScore` в `cout`. На этот раз на экран выводятся значения `1000` и `150`. Переменные поменялись значениями. Я взял себе большой результат, а вам достался маленький. Наконец-то я вас обставил!

Передача ссылок для обеспечения эффективности

Передача переменной по значению сопряжена с некоторыми издержками, так как вам придется скопировать переменную, прежде чем вы присвоите ей параметр. Когда речь идет о переменных простых встроенных типов — скажем, `int` или `float`, — такие издержки являются пренебрежимо маленькими. Но копирование большого объекта — например, представляющего целый трехмерный игровой мир — бывает дорогим удовольствием. В то же время передача по ссылке гораздо эффективнее, так как при этом не приходится копировать переменную-аргумент. Вместо этого вы просто предоставляете через ссылку доступ к уже существующему объекту.

Знакомство с программой Inventory Displayer

Программа Inventory Displayer создает вектор строк, представляющий собой снаряжение героя. Затем она вызывает функцию, отображающую это снаряжение. Программа передает отображающей функции вектор элементов как ссылку, поэтому данный вызов эффективен (сам вектор не копируется). Правда, здесь возникает новая заминка. Программа передает вектор как особую ссылку, и этот прием не позволяет отображающей функции изменить вектор. Наша программа продемонстрирована на рис. 6.3.

Рис. 6.3. Вектор inventory быстрым и безопасным способом передается функции, отображающей снаряжение героя (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 6, имя файла — `inventory_displayer.cpp`.

```
// Программа InventoryDisplayer
// Демонстрирует работу с константными ссылками
#include <iostream>
#include <string>
#include <vector>
usingnamespacestd;
// параметр vec – это константная ссылка на вектор строк
void display(const vector<string>& inventory);
int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");
    display(inventory);
    return 0;
}
// параметр vec – это константная ссылка на вектор строк
void display(const vector<string>& vec)
{
    cout << "Your items:\n";
    for (vector<string>::const_iterator iter = vec.begin();
        iter != vec.end(); ++iter)
    {
        cout << *iter << endl;
    }
}
```

Несколько слов о подводных камнях, связанных с передачей ссылок

Один из приемов, позволяющих эффективно предоставлять функции доступ к большому объекту, — передача этого объекта по ссылке. Однако здесь кроется потенциальная проблема. Как мы помним из программы Swap, в таком случае возникает возможность изменения переменной-аргумента. Что же делать, если вы не желаете допускать такие изменения? Существует ли способ и воспользоваться преимуществами передачи информации по ссылке и обеспечить неприкосновенность переменной-аргумента? Да. Нужно передавать константную ссылку.

ПОДСКАЗКА

В принципе, переменную-аргумент лучше не изменять. Старайтесь писать такие функции, которые сообщают вызвавшему их коду новую информацию посредством возвращаемого значения.

Объявление параметров как константных ссылок

Функция `display()` отображает содержимое снаряжения героя. В заголовке функции я указываю один параметр — константную ссылку на вектор `vec`, содержащий строковые объекты:

```
void display(const vector<string>& vec)
```

Константная ссылка — это вид ограниченной ссылки. Она работает, как и любая другая ссылка, с той оговоркой, что мы не можем изменить значение, на которое она указывает. Для создания константной ссылки просто поставьте ключевое слово `const` перед типом в объявлении этой ссылки.

Какое значение все это имеет для функции `display()`? Поскольку параметр `vec` является константной ссылкой, это означает, что функция `display()` не может изменить `vec`. Следовательно, вектор `inventory` защищен и его нельзя изменить функцией `display()`. В принципе, можно эффективно передавать аргумент функции, если этот аргумент — константная ссылка. Таким образом аргумент окажется доступен, а изменить его будет нельзя. Можно сказать, что мы предоставляем функции доступ к этому аргументу в режиме «только для чтения». Хотя константные ссылки особенно полезны именно при указании параметров функции, их можно использовать в любой части программы.

ПОДСКАЗКА

Константные ссылки удобны и в ином отношении. Если требуется создать ссылку на константу, то такое значение должно присваиваться только константной ссылке (обычная ссылка в данном случае неприменима).

Передача константной ссылки

Вернувшись в функцию `main()`, я создаю вектор `inventory`, а затем вызываю функцию `display()` с помощью следующей строки кода, которая передает вектор как константную ссылку:

```
display(inventory);
```

В результате получается эффективный и безопасный вызов функции. Он эффективен, так как здесь передается только ссылка, а сам вектор не копируется. Он безопасен, так как ссылка на вектор является константной и функция `display()` не может изменить вектор `inventory`.

ОСТОРОЖНО!

Невозможно изменить параметр, помеченный как константная ссылка. Если вы попытаетесь это сделать, то получите ошибку компиляции.

Далее функция `display()` перечисляет все элементы вектора, применяя при этом константную ссылку на вектор `inventory`. Затем управление вновь передается функции `main()` и программа завершается.

Решение о том, как передавать аргументы

Итак, мы уже рассмотрели три различных способа передачи аргументов: по значению, по ссылке и в виде константной ссылки. Как же решить, который из способов использовать? Вот некоторые рекомендации.

- **По значению.** Передавайте информацию по значению, когда переменная-аргумент относится к одному из примитивов (встроенных типов), например `bool`, `int` или `float`. Объекты этих типов настолько миниатюрны, что при передаче их по ссылке эффективность работы программы нисколько не возрастает. Кроме того, передавать информацию по значению целесообразно в тех случаях, когда вы хотите, чтобы компьютер сделал копию переменной. Такая копия может вам понадобиться, например, если вы планируете изменить один из параметров функции, но при этом не желаете распространять это изменение на саму переменную-аргумент.
- **В качестве константной ссылки.** Сообщайте информацию в виде константной ссылки, если хотите эффективно передать значение, которое не собираетесь изменять.
- **По ссылке.** Передавайте информацию по ссылке лишь в тех случаях, когда собираетесь изменять значение переменной-аргумента. Правда, следует всячески избегать изменения таких переменных.

Возврат ссылок

Как и при передаче значения, при возврате значения от функции вы фактически возвращаете не само значение, а его копию. Если значение относится к одному из примитивов (встроенных типов), то беспокоиться вам не о чем. Но такая операция может быть довольно затратной в том случае, если приходится возвращать крупный объект. Рассмотрим эффективное альтернативное решение — возврат ссылки.

Знакомство с программой Inventory Referencer

Программа InventoryReferencer демонстрирует возврат ссылок. Она отображает элементы вектора, в котором содержатся артефакты из снаряжения героя, с помощью возврата ссылок. Затем программа изменяет один из элементов посредством возвращенной ссылки. Результат выполнения программы показан на рис. 6.4.

Рис. 6.4. Элементы снаряжения героя отображаются и изменяются с помощью возвращаемых ссылок (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 6, имя файла — `inventory_referencer.cpp`.

```
// Программа Inventory Referencer
// Демонстрирует возврат ссылки
#include <iostream>
#include <string>
#include <vector>
using namespace std;
// возвращает ссылку на строку
string& refToElement(vector<string>& inventory, int i);
int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");
    // отображается строка, на которую указывает возвращенная ссылка
    cout << "Sending the returned reference to cout:\n";
    cout << refToElement(inventory, 0) << "\n\n";
    // присваивает одну ссылку другой – малозатратная операция присваивания
    cout << "Assigning the returned reference to another reference.\n";
    string& rStr = refToElement(inventory, 1);
    cout << "Sending the new reference to cout:\n";
    cout << rStr << "\n\n";
```

```
    // копирует строковый объект – затратная операция присваивания
    cout << "Assigning the returned reference to a string object.\n";
    string str = refToElement(inventory, 2);
    cout << "Sending the new string object to cout:\n";
    cout << str << "\n\n";
    // изменение строкового объекта посредством возвращенной ссылки
    cout << "Altering an object through a returned reference.\n";
    rStr = "Healing Potion";
    cout << "Sending the altered object to cout:\n";
    cout << inventory[1] << endl;
    return 0;
}
// возвращает ссылку на строку
string& refToElement(vector<string>& vec, int i)
{
    return vec[i];
}
```

Возврат ссылки

Прежде чем вы сможете вернуть ссылку от функции, нужно указать, что вы собираетесь ее возвращать. Именно это я делаю в заголовке функции `refToElement()`:

```
string& refToElement(vector<string>& inventory, int i)
```

Используя ссылочный оператор в коде `string&` при указании возвращаемого типа, я указываю, что функция будет возвращать ссылку на объект `string` (а не сам объект `string`). Можете использовать ссылочный оператор так, как это сделано здесь, если хотите указать, что функция возвращает ссылку на объект конкретного типа. Просто поставьте ссылочный оператор после имени типа.

В теле функции `refToElement()` содержится всего одна инструкция, возвращающая ссылку на элемент, который располагается в векторе на позиции `i`:

```
returnvec[i];
```

Обратите внимание: в инструкции `return` ничто не указывает, что функция возвращает ссылку. Заголовок функции и ее прототип определяют, что будет возвращать эта функция, объект или ссылку на объект.

ОСТОРОЖНО!

Хотя возврат ссылки — эффективный способ передачи информации обратно той функции, которая делала вызов, будьте внимательны и не возвращайте ссылку на объект, расположенный за пределами области видимости, то есть на объект, который прекращает существование. Например, следующая функция возвращает ссылку на объект string, который не будет существовать после завершения данной функции. Это недопустимо:

```
string& badReference()
{
string local = "This string will cease to exist once the function ends.";
returnlocal;
}
```

Один из способов избежать подобных проблем — никогда не возвращать ссылку на локальную переменную.

Отображение значения возвращенной ссылки

Создав вектор элементов inventory, я отображаю первый из содержащихся в нем элементов с помощью возвращенной ссылки:

```
cout << refToElement(inventory, 0) << "\n\n";
```

Данная строка кода вызывает функцию refToElement(), которая возвращает ссылку на элемент, расположенный на позиции 0 в векторе inventory, а затем посылает эту ссылку в cout. В результате на экран выводится слово sword.

Присваивание возвращенной ссылки другой ссылке

В следующей строке я присваиваю возвращенную ссылку другой ссылке. Эта вторая ссылка указывает на элемент, расположенный на позиции 1 в векторе inventory, после чего этот элемент присваивается ссылке rStr:

```
string&rStr = refToElement(inventory, 1);
```

Такие операции присваивания эффективны, поскольку, когда мы присваиваем одну ссылку другой, не приходится копировать объект. Затем я посылаю rStr в cout, и на экране отображается armor.

Присваивание возвращенной ссылки переменной

Далее я присваиваю возвращенную ссылку переменной.

```
stringstr = refToElement(inventory, 2);
```

Предыдущий код не присваивает ссылку str. Он и не может этого сделать, так как str — это объект string. Вместо этого код копирует элемент, на который указывает возвращенная ссылка (это элемент, расположенный в векторе inventory на позиции 2). Затем данная новая копия объекта string присваивается str. Поскольку такой вариант присваивания связан с копированием объекта, он затратнее, чем присваивание одной ссылки другой. Иногда затраты, возникающие при подобном копировании объекта, вполне оправданны, но необходимо учитывать, что данный метод присваивания сопряжен с дополнительными издержками, и по возможности его избегать. Далее я посылаю новый строковый объект Str в cout, и на экране отображается shield.

Изменение объекта с помощью возвращенной ссылки

Можно также изменить объект, на который указывает возвращенная ссылка. Это означает, что вы можете модифицировать снаряжение героя с помощью rStr, как делается в следующей строке кода:

```
rStr = "HealingPotion";
```

Поскольку ссылка rStr относится к элементу, расположенному в векторе inventory на позиции 1, этот код изменяет inventory[1], то есть он равен "HealingPotion". Чтобы подтвердить это, я вывожу элемент на экран с помощью следующей строки кода:

```
cout<<inventory[1] <<endl;
```

Действительно, видим HealingPotion.

Если бы я хотел защитить inventory так, чтобы ссылка, возвращенная функцией refToElement(), не могла применяться для изменения вектора, то должен был бы указать возвращаемый тип функции как константную ссылку.

Знакомство с игрой «Крестики-нолики»

В проекте к этой главе вы узнаете, как создавать компьютерного соперника. Для этого мы используем толику *искусственного интеллекта*. В игре пользователь и компьютер, человек и машина, померяются силами в бескомпромиссном поединке в «Крестики-нолики». Компьютер играет превосходно (хотя и неидеально) и может действовать достаточно изобретательно для того, чтобы любая партия получилась интересной. На рис. 6.5 показано начало партии.

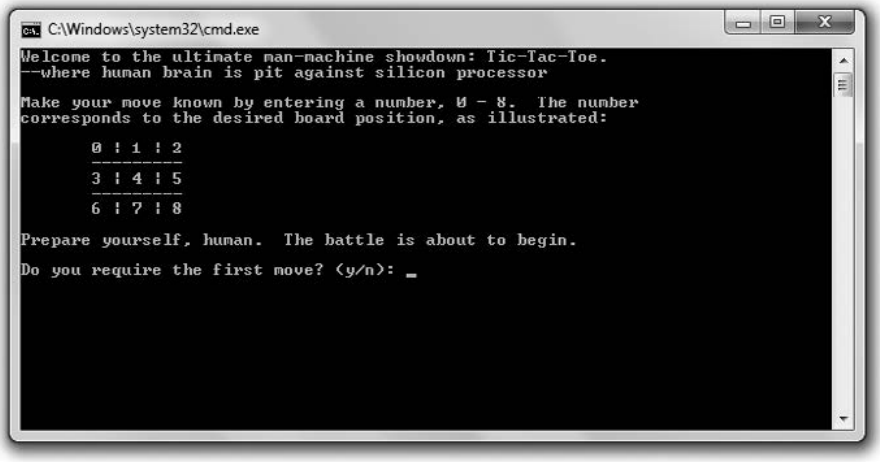


Рис. 6.5. Компьютер полон уверенности в себе (опубликовано с разрешения компании Microsoft)

Планирование игры

На данный момент эта игра — ваш самый амбициозный проект на языке C++. Вы определенно обладаете всеми необходимыми навыками для его реализации, но я решил подробно описать планирование этой игры, чтобы вы видели ее в широком контексте и понимали, как создать более крупную программу. Не забывайте, что планирование — это самый важный этап программирования. Без карты

невозможно достичь цели (в лучшем случае дорога затянется, поскольку вы пойдете окольным путем).

НА ПРАКТИКЕ

Геймдизайнеры тратят массу времени на подготовку концептуальных описаний, проектных документов и прототипов игры до того, как программист приступит к написанию кода. Лишь когда дизайнерская работа завершена, программист берется за дело — выполняет свою часть планирования. Только когда программист подготовит технический проект, он всерьез берется за написание кода. Вывод? Планируйте. Проще переделать чертеж, чем 50-этажное здание.

Написание псевдокода

Возвращаемся к нашему любимому языку, который, строго говоря, языком называется условно. Это псевдокод. Поскольку большинство задач в программе будет решаться с помощью функций, я могу без опаски подготовить довольно абстрактную модель кода. Каждая строка псевдокода условно соответствует одному вызову функции. Затем мне останется всего лишь написать функции, предусмотренные в плане. Вот псевдокод:

```
Создать пустое поле для игры в крестики-нолики
Вывести на экран правила игры
Определить, кто ходит первым
Отобразить поле
Если пока никто не победил и не наступила ничья
И если сейчас ход пользователя
Получить ход пользователя
Обновить игровое поле с учетом хода пользователя
Иначе
Вычислить ход компьютера
Обновить поле с учетом хода компьютера
Отобразить поле
Передать ход сопернику
Поздравить победителя или объявить ничью
```

Представление данных

Итак, у нас готов неплохой план. Но пока он довольно абстрактный, и речь в нем идет о манипуляциях различными элементами, которые мы представляем себе очень приблизительно. Мне кажется, что можно запрограммировать ход, как будто на поле ставится новый символ. Но как именно в программе будет представлено поле? А символ? А ход?

Поскольку я собираюсь отображать игровое поле на экране, почему бы не представить крестик как символ «X», а нолик — как символ «O»? В качестве пустой клетки можно просто оставить пробел. Следовательно, само игровое поле может представлять собой вектор, содержащий экземпляры типа `char`. На игровом поле «Крестиков-ноликов» девять клеток, соответственно, наш вектор должен содержать девять элементов. На рис. 6.6 показано, что я имею в виду.

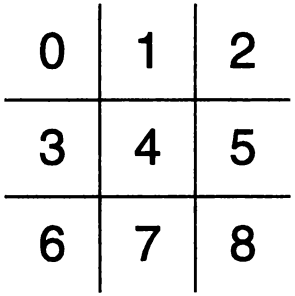


Рис. 6.6. Номер в каждой клетке соответствует позиции этой клетки в векторе

Каждая клетка (позиция) на поле представлена числом от 0 до 8. Это означает, что в векторе будет девять элементов (позиций), пронумерованных от 0 до 8. Поскольку при каждом ходе указывается клетка, в которой должен быть поставлен символ, ходы также будут пронумерованы от 0 до 8. Таким образом, ход должен быть представлен как целое число (`int`).

Та сторона, которую занимает пользователь, и та, которую занимает компьютер, должны быть представлены символом `char` `X` или `O`, точно так же, как фигуры, которыми ходит тот или иной соперник. Переменная, определяющая, чья очередь сейчас делать ход, также будет представлять собой символ `char` `X` или `O`.

Создание списка функций

Уже в псевдокоде просматриваются различные функции, которые мне понадобятся. Я создам список этих функций, продумав, какую задачу станет решать каждая из них, какие параметры у них будут и какие значения эти функции будут возвращать. В табл. 6.1 показано, что у меня получилось.

Таблица 6.1. Функции для игры «Крестики-нолики»

Функция	Описание
void instructions()	Отображает правила игры
char askYesNo(string question)	Задает вопрос, предполагающий ответ «да» или «нет». Получает вопрос. Возвращает символ y или n
int askNumber(string question, int high, int low = 0)	Запрашивает число из диапазона. Получает вопрос, малое число и большое число. Возвращает число из диапазона от low до high
char humanPiece()	Определяет, какими фигурами будет ходить пользователь. Возвращает X или O
char opponent(char piece)	Зная, какими фигурами будет ходить пользователь, определяет фигуру для компьютера. Получает X или O. Возвращает X или O
void displayBoard(const vector<char>&board)	Отображает поле на экране. Получает поле

Продолжение ⇩

Таблица 6.1 *(продолжение)*

Функция	Описание
char winner(const vector<char>&board)	Определяет победителя игры. Получает поле. Возвращает символ X, или O, или T (этот случай соответствует ничьей), или N (означает, что пока у обоих соперников остаются шансы на победу)
bool isLegal(const vector<char>&board, int move)	Определяет, по правилам ли сделан ход. Получает поле и ход. Возвращает true или false
int humanMove(const vector<char>&board, char human)	Узнает ход пользователя. Получает поле и фигуру, которой ходит пользователь. Возвращает ход пользователя
int computerMove(vector<char>board, char computer)	Узнает ход компьютера. Получает поле и фигуру, которой ходит компьютер. Возвращает ход компьютера
void announceWinner(char winner, char computer, char human)	Поздравляет победителя или объявляет ничью. Получает победившую сторону, фигуру, которой ходил пользователь, и фигуру, которой ходил компьютер

Подготовка программы

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 6, имя файла — `tic-tac-toe.cpp`. В этом разделе я проанализирую весь код игры, фрагмент за фрагментом.

В начале программы я включаю в код необходимые файлы, определяю некоторые глобальные константы и пишу прототипы функций:

```
// Программа Крестики-нолики
// Компьютер играет в "Крестики-нолики" против пользователя
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
// глобальные константы
constcharX = 'X';
const char O = 'O';
const char EMPTY = ' ';
const char TIE = 'T';
const char NO_ONE = 'N';:
// прототипы функций
void instructions();
char askYesNo(string question);
int askNumber(string question, int high, int low = 0);
char humanPiece();
char opponent(char piece);
void displayBoard(const vector<char>& board);
char winner(const vector<char>& board);
bool isLegal(const vector<char>& board, int move);
int humanMove(const vector<char>& board, char human);
int computerMove(vector<char> board, char computer);
void announceWinner(char winner, char computer, char human);
```

В разделе с глобальными константами X — это сокращенная запись charX, первого участника игры, O — сокращенная запись charO, второго участника игры. EMPTY — это также константа char, она соответствует пустой клетке на игровом поле. Поскольку EMPTY соответствует пробелу, на поле вместо этой константы будет пустая клетка. TIE — это константа char, соответствующая ничейному исходу игры. Наконец, charNO_ONE — это константа, не соответствующая ни одному из соперников, а означающая, что в игре пока никто не победил.

Функция main()

Как видите, функция main() практически буквально соответствует псевдокоду, который я написал ранее:

```
// функция main
int main()
{
    int move;
    const int NUM_SQUARES = 9;
    vector<char> board(NUM_SQUARES, EMPTY);
    instructions();
    char human = humanPiece();
    char computer = opponent(human);
    char turn = X;
    displayBoard(board);
    while (winner(board) == NO_ONE)
    {
        if (turn == human)
        {
            move = humanMove(board, human);
            board[move] = human;
        }
        else
        {
            move = computerMove(board, computer);
            board[move] = computer;
        }
        displayBoard(board);
        turn = opponent(turn);
    }
    announceWinner(winner(board), computer, human);
    return 0;
}
Функция instructions()
```

Эта функция выводит на экран правила игры и позволяет компьютеру немного похвастаться:

```
void instructions()
{
    cout << "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.\n";
    cout << "--where human brain is pit against silicon processor\n\n";
```

```
    cout << "Make your move known by entering a number, 0 - 8. The number\n";
    cout << "corresponds to the desired board position, as illustrated:\n\n";
    cout << " 0 | 1 | 2\n";
    cout << " ---------\n";
    cout << " 3 | 4 | 5\n";
    cout << " ---------\n";
    cout << " 6 | 7 | 8\n\n";
    cout << "Prepare yourself, human. The battle is about to begin.\n\n";
}
```

Функция askYesNo()

Эта функция задает пользователю вопрос, на который можно ответить «да» или «нет». Программа продолжает задавать этот вопрос, пока пользователь не введет символ y или n:

```
char askYesNo(string question)
{
    char response;
    do
    {
        cout << question << " (y/n): ";
        cin >> response;
    } while (response != 'y' && response != 'n');
    returnresponse;
}
```

Функция askNumber()

Эта функция запрашивает у пользователя число из определенного диапазона и продолжает задавать этот вопрос, пока пользователь не введет удовлетворяющее запросу число. Она принимает вопрос, максимальное число, минимальное число, а возвращает число из указанного диапазона:

```
int askNumber(string question, int high, int low)
{
    int number;
    do
    {
        cout << question << " (" << low << " - " << high << "): ";
        cin >> number;
    } while (number > high || number < low);
    returnnumber;
}
```

Внимательно посмотрев на прототип этой функции, вы можете заметить, что минимальное число по умолчанию имеет значение 0. Когда я буду вызывать эту функцию в программе, я воспользуюсь данным фактом.

Функция humanPiece()

Эта функция спрашивает пользователя, хочет ли он пойти первым, и в зависимости от сделанного выбора возвращает ту фигуру, которой будет ходить пользователь. По традиции крестики ходят первыми:

```
char humanPiece()
{
    char go_first = askYesNo("Do you require the first move?");
    if (go_first == 'y')
    {
        cout << "\nThen take the first move. You will need it.\n";
        return X;
    }
    else
    {
        cout << "\nYour bravery will be your undoing...I will go first.\n";
        return0;
    }
}
```

Функция opponent()

Эта функция узнает фигуру пользователя (X или 0) и на основании этой информации возвращает фигуру, которой будет ходить соперник-компьютер (X или 0):

```
char opponent(char piece)
{
    if (piece == X)
    {
        return 0;
    }
    else
    {
        return X;
    }
}
```

Функция displayBoard()

Эта функция отображает переданное ей игровое поле. Поскольку в игре присутствуют элементы всего трех видов: X, 0 или пробел, функция может отображать каждый из этих элементов. Чтобы нарисовать красивое поле для игры в «Крестики-нолики», я использую и некоторые символы с клавиатуры:

```
void displayBoard(const vector<char>& board)
{
    cout << "\n\t" << board[0] << " | " << board[1] << " | " << board[2];
    cout << "\n\t" << "---------";
```

```
    cout << "\n\t" << board[3] << " | " << board[4] << " | " << board[5];
    cout << "\n\t" << "---------";
    cout << "\n\t" << board[6] << " | " << board[7] << " | " << board[8];
    cout<< "\n\n";
}
```

Обратите внимание: вектор, представляющий игровое поле, передается с помощью константной ссылки. Таким образом, передача вектора организована эффективно, он не копируется. Это также означает, что вектор надежно защищен от любых изменений. Поскольку в этой функции я собираюсь просто отобразить игровое поле, но ничего не планирую с ним делать, такой вариант идеален.

Функция winner()

Данная функция получает игровое поле и возвращает победителя. Она может иметь один из четырех вариантов значения. Если один из игроков победил, то функция вернет `X` или `0`. Если игра окончилась вничью, то функция вернет `TIE`. Наконец, если пока никто не победил, а на поле остается еще хотя бы одна свободная клетка, эта функция вернет `NO_ONE`:

```
char winner(const vector<char>& board)
{
    // все возможные выигрышные ряды
    const int WINNING_ROWS[8][3] = { {0, 1, 2},
        {3, 4, 5},
        {6, 7, 8},
        {0, 3, 6},
        {1, 4, 7},
        {2, 5, 8},
        {0, 4, 8},
        {2, 4, 6} };
```

Первым делом здесь следует отметить, что вектор, представляющий игровое поле, передается через константную ссылку. Таким образом, передача вектора организована эффективно: он не копируется. Это также означает, что вектор надежно защищен от любых изменений.

В начальной части функции я определяю константу — двухмерный массив целых чисел (`int`); эта константа будет называться `WINNING_ROWS`. В ней представлены все восемь способов заполнить своими фигурами одну из линий и выиграть. Каждый выигрышный ряд представлен группой из трех чисел (тремя выигрышными позициями). Например, группа `{0, 1, 2}` соответствует верхней строке — позициям 0, 1 и 2. Следующая группа `{3, 4, 5}` соответствует средней строке — позициям 3, 4 и 5 и т. д.

Затем я проверяю, не победил ли уже один из игроков:

```
const int TOTAL_ROWS = 8;
// если в одном из выигрышных рядов уже присутствуют три одинаковых значения
// (причем они не равны EMPTY), то победитель определился
for(int row = 0; row < TOTAL_ROWS; ++row)
{
    if ( (board[WINNING_ROWS[row][0]] != EMPTY) &&
```

```
    (board[WINNING_ROWS[row][0]] == board[WINNING_ROWS[row][1]]) &&
    (board[WINNING_ROWS[row][1]] == board[WINNING_ROWS[row][2]]) )
    {
        return board[WINNING_ROWS[row][0]];
    }
}
```

Я перебираю все возможные способы, которыми может победить игрок, проверяя, нет ли в любом из выигрышных рядов трех одинаковых символов. Инструкция `if` проверяет, содержатся ли во всех клетках того или иного выигрышного ряда три одинаковых значения, и, если так, не равны ли эти значения `EMPTY`. При выполнении обоих условий можно констатировать, что хотя бы в одном из рядов есть три крестика или три нолика, то есть один из игроков победил. Затем функция возвращает символ, стоящий в первой позиции выигравшего ряда.

Если ни один из соперников не победил, я проверяю, не наступила ли ничья:

```
// поскольку победитель не определился, проверяем, не наступила ли ничья
// (остались ли на поле пустые клетки)
if (count(board.begin(), board.end(), EMPTY) == 0)
    return TIE;
```

Если на поле не осталось пустых клеток, а победитель не определился, то игра закончилась вничью. Я использую алгоритм `count()` из библиотеки `STL`, который подсчитывает, сколько раз заданное значение встречается в контейнере элементов, чтобы определить количество еще остающихся элементов, равных `EMPTY`. Если это число равно 0, то функция возвращает `TIE`.

Наконец, если ни один из игроков пока не победил, но и ничья в игре пока не наступила, то игра продолжается. Функция возвращает константу `NO_ONE`:

```
// Поскольку победитель не определился, но и ничья еще не наступила,
// игра продолжается
    return NO_ONE;
}
```

Функция isLegal()

Эта функция получает игровое поле и сделанный ход. Она возвращает `true`, если ход сделан по правилам, и `false` — если не по правилам. Ход по правилам заключается во вставке символа X или O в пустую клетку:

```
inline bool isLegal(int move, const vector<char>& board)
{
    return (board[move] == EMPTY);
}
```

Здесь необходимо еще раз отметить, что вектор, представляющий игровое поле, передается через константную ссылку. Таким образом, передача вектора организована эффективно: он не копируется. Это также означает, что вектор надежно защищен от любых изменений.

Как видите, я подставляю функцию `isLegal()`. В современных компиляторах хорошо организована автоматическая оптимизация, но, поскольку эта функция состоит всего из одной строки, подставить ее будет довольно удобно.

Функция humanMove()

Эта функция получает игровое поле и ту фигуру, которой ходит пользователь. Она возвращает номер той клетки, в которую пользователь хочет поставить свой символ. Функция запрашивает у пользователя номер клетки, в которую он хочет поставить символ, и ждет, пока пользователь не походит по правилам. Затем функция возвращает сделанный ход:

```
int humanMove(const vector<char>& board, char human)
{
    int move = askNumber("Where will you move?", (board.size() - 1));
    while (!isLegal(move, board))
    {
        cout << "\nThat square is already occupied, foolish human.\n";
        move = askNumber("Where will you move?", (board.size() - 1));
    }
    cout << "Fine...\n";
    return move;
}
```

Отмечу, что вектор, представляющий игровое поле, передается через константную ссылку. Таким образом, передача вектора организована эффективно; он не копируется. Это также означает, что вектор надежно защищен от любых изменений.

Функция computerMove()

Функция получает игровое поле и ту фигуру, которой ходит компьютер. Она возвращает ход компьютера. Первым делом отмечу, что здесь я не передаю поле по ссылке:

```
int computerMove(vector<char> board, char computer)
```

Вместо этого я решаю передать поле по значению, пусть этот способ и менее эффективен, чем передача по ссылке. Я передаю поле по значению, так как далее мне придется с ним работать и изменять копию поля, пока я буду пробовать различные варианты хода и определять за компьютер оптимальный вариант. Работая с копией, я оставляю исходный вектор нетронутым — оригинал поля по-прежнему защищен от изменений.

Далее рассмотрим внутренний механизм этой функции. Как реализовать в программе немного искусственного интеллекта, чтобы она была интересным соперником? При выборе хода я придерживаюсь базовой трехэтапной стратегии, которая строится так.

1. Если у компьютера есть возможность сделать ход, который принесет ему победу, — сделать этот ход.
2. Иначе, если человек сможет победить следующим ходом, блокировать этот ход.
3. Иначе занять лучшую из оставшихся клеток. Самая лучшая клетка расположена в центре поля, менее ценны угловые клетки, еще ниже ценятся все оставшиеся клетки

В следующей части функции я реализую шаг 1:

```
unsigned int move = 0;
bool found = false;
// если компьютер может выиграть следующим ходом, то он делает этот ход
while (!found && move < board.size())
{
    if (isLegal(move, board))
    {
        board[move] = computer;
        found = winner(board) == computer;
        board[move] = EMPTY;
    }
    if (!found)
    {
        ++move;
    }
}
```

Сначала я перебираю в цикле все возможные варианты хода от 0 до 8. Проверяю, не противоречит ли каждый из ходов правилам. Если ход допустим правилами, то я ставлю фигуру компьютера в выбранную клетку и проверяю, приводит ли такой ход компьютер к победе. Если этот ход не дает победы, пробую поставить фигуру в следующую свободную клетку. Однако если ход привел компьютер к победе, то цикл завершается. Я нашел ход (`found` равно `true`), который должен сделать компьютер (занять клетку номер `move`), чтобы победить.

Далее я проверяю, должен ли искусственный интеллект перейти к следующему этапу описанной стратегии. Если выигрышный ход пока не найден (`found` равно `false`), проверяю, может ли пользователь выиграть следующим ходом:

```
// иначе, если человек может победить следующим ходом, блокировать этот ход
if (!found)
{
    move = 0;
    char human = opponent(computer);
    while (!found && move < board.size())
    {
        if (isLegal(move, board))
        {
            board[move] = human;
            found = winner(board) == human;
            board[move] = EMPTY;
        }
        if (!found)
        {
            ++move;
        }
    }
}
```

Сначала я перебираю в цикле все возможные варианты хода от 0 до 8. Проверяю, не противоречит ли каждый из ходов правилам. Если ход допустим правилами, то я ставлю фигуру пользователя в выбранную клетку и проверяю, приводит ли такой ход пользователя к победе. Затем отменяю найденный выигрышный ход пользователя так, чтобы эта клетка вновь опустела. Если проверенный ход не приводит пользователя к победе, проверяю следующую пустую клетку. Правда, если ход приводит пользователя к победе, то я нашел тот ход, который сейчас должен обязательно сделать компьютер (found равно true). Компьютер поставит свой символ в клетку номер move, чтобы не дать пользователю выиграть следующим ходом.

Далее я проверяю, должен ли компьютер перейти к третьему этапу стратегии, заложенной в искусственном интеллекте. Если я пока не нашел выигрышного хода (found равно false), то просматриваю ряд желательных ходов, начиная с самого выгодного, и выбираю первый, который не противоречит правилам:

```
// иначе занять следующим ходом оптимальную свободную клетку
if (!found)
{
    mòve = 0;
    unsigned int i = 0;
    constintBEST_MOVES[] = {4, 0, 2, 6, 8, 1, 3, 5, 7};
    // выбрать оптимальную свободную клетку
    while (!found && i < board.size())
    {
        move = BEST_MOVES[i];
        if (isLegal(move, board))
        {
            found = true;
        }
        ++i;
    }
}
```

На данном этапе выполнения функции я нашел ход, который должен сделать компьютер. В зависимости от ситуации этот ход может принести компьютеру победу, не допустить победы пользователя либо просто позволяет занять оптимальную из оставшихся свободных клеток. Итак, компьютер должен объявить ход и вернуть номер соответствующей клетки:

```
    cout << "I shall take square number " << move << endl;
    returnmove;
}
```

НА ПРАКТИКЕ

В нашей игре «Крестики-нолики» компьютер продумывает лишь ближайший возможный ход. Программы серьезных стратегических игр, например компьютерных шахмат, гораздо глубже анализируют последствия конкретного хода, учитывают много этапов ходов и ответных ходов. Хороший симулятор шахмат успевает проанализировать миллионы игровых позиций, прежде чем сделает ход.

Функция announceWinner()

Эта функция получает победителя игры, фигуру, которой играл компьютер, и фигуру, которой играл пользователь. Далее функция объявляет победителя партии или констатирует ничью:

```
void announceWinner(charwinner, charcomputer, charhuman)
{
    if (winner == computer)
    {
        cout << winner << "'s won!\n";
        cout << "As I predicted, human, I am triumphant once more -- proof\n";
        cout << "that computers are superior to humans in all regards.\n";
    }
    else if (winner == human)
    {
        cout << winner << "'s won!\n";
        cout << "No, no! It cannot be! Somehow you tricked me, human.\n";
        cout << "But never again! I, the computer, so swear it!\n";
    }
    else
    {
        cout << "It's a tie.\n";
        cout << "You were most lucky, human, and somehow managed to tie me.\n";
        cout << "Celebrate...for this is the best you will ever achieve.\n";
    }
}
```

Резюме

В этой главе мы изучили следующий материал.

- Ссылка — это псевдоним, то есть второе название переменной.
- Ссылка создается с помощью символа & — ссылочного оператора.
- При определении ссылка должна инициализироваться.
- Ссылку нельзя изменить так, чтобы она стала указывать на иную переменную.
- Все ваши манипуляции со ссылкой применяются и к той переменной, на которую указывает эта ссылка.
- Когда вы присваиваете ссылку переменной, вы создаете копию того значения, на которое ссылаетесь.
- Сообщая переменную функции по значению, вы передаете функции копии этой переменной.
- Сообщая переменную функции по ссылке, вы предоставляете функции доступ к этой переменной.

- Передача по ссылке обычно более эффективна, чем по значению, особенно если речь идет о передаче крупных объектов.
- Передача по ссылке обеспечивает непосредственный доступ к переменной-аргументу, переданной в функцию. В результате функция может вносить изменения в переменную-аргумент.
- Константная ссылка неприменима для изменения того значения, на которое она указывает. Константные ссылки объявляются с помощью ключевого слова `const`.
- Константную ссылку или значение-константу нельзя присвоить константной ссылке.
- Передавая функции константную ссылку, мы защищаем переменную-аргумент от изменений, которые могла бы в нее внести эта функция.
- Если менять значение переменной-аргумента, передаваемой в функцию, может возникнуть путаница. Поэтому программисты-игровики сначала проверяют, можно ли обойтись константной ссылкой, и, только если это невозможно, прибегают к неконстантной.
- Возврат ссылки обычно более эффективен, чем возврат копии значения, особенно если бы потребовалось вернуть крупный объект.
- Можно вернуть константную ссылку на объект так, чтобы объект нельзя было изменить через возвращенную ссылку.
- Простейший игровой искусственный интеллект действует так: компьютер рассматривает все потенциальные ходы, разрешенные правилами, а также все возможные ответные ходы оппонента и лишь затем решает, какой ход сделать.

Вопросы и ответы

1. *При объявлении функции разные программисты ставят ссылочный оператор (&) при объявлении ссылки в разных частях функции. Где же его лучше ставить?*

   Существует три основных стиля, применяемых при использовании ссылочного оператора. Одни программисты пишут `int&ref = var;`, другие — `int&ref = var;`, третьи — `int&ref = var;`. Компьютер разберется с любым из трех вариантов. Конкретный стиль зависит от вашего вкуса, однако, выбрав один стиль, его нужно придерживаться.

2. *Почему нельзя инициализировать неконстантную ссылку константным значением?*

   Потому что неконстантная ссылка позволяет вносить изменения в то значение, на которое указывает.

3. *Если я инициализирую константную ссылку на неконстантную переменную, смогу ли я изменить значение этой переменной?*

   С помощью константной ссылки — нет, поскольку, объявляя константную ссылку, вы тем самым сообщаете, что эта ссылка не может использоваться для изме-

нения значения, на которое указывает (даже если это значение может быть изменено другими способами).

4. *В чем заключается экономность передачи константной ссылки?*

   Когда мы передаем функции крупный объект по значению, программа, разумеется, копирует этот объект. В зависимости от размера объекта такая операция может быть довольно затратной. Передавая ссылку, мы просто предоставляем функции доступ к большому объекту — это очень легкая операция.

5. *Можно ли поставить ссылку на ссылку?*

   Не совсем. Можно присвоить одну ссылку другой, но новая ссылка просто будет указывать на то же значение, что и старая.

6. *Что произойдет, если я объявлю ссылку, но не инициализирую ее?*

   Компилятор выдаст ошибку, поскольку так делать нельзя.

7. *Почему следует избегать изменения переменной, которая сообщается по ссылке?*

   Потому что это может привести к путанице. По вызову функции как таковому невозможно определить, может ли изменяться значение передаваемой в нем переменной.

8. *Если я не изменяю переменные-аргументы, сообщаемые функциям, как же мне передавать новую информацию тому коду, который сделал вызов?*

   Пользуйтесь возвращаемыми значениями.

9. *Можно ли передать литерал как неконстантную ссылку?*

   Нет. Если вы попытаетесь передать литерал как неконстантную ссылку, то произойдет ошибка компиляции.

10. *Существует ли возможность передать литерал параметру, принимающему ссылку?*

    Существует. Можно передать литерал как константную ссылку.

11. *Что присходит, когда я возвращаю объект от функции?*

    Как правило, программа создает копию этого объекта, которую и возвращает. Эта операция может быть довольно затратной в зависимости от размера объекта.

12. *Зачем возвращать ссылку?*

    Такой прием может быть эффективнее, чем возврат значения, так как не связан с копированием объекта.

13. *Какие частичные потери эффективности возможны при возврате ссылки?*

    Издержки могут возникнуть при присваивании возвращенной ссылки переменной. Когда вы присваиваете ссылку переменной, компьютер должен сделать копию того объекта, на который указывает данная ссылка.

14. *Какие проблемы могут возникнуть при возврате ссылки на локальную переменную?*

    Локальная переменная не существует после завершения функции, а значит, вы возвращаете ссылку на несуществующий объект, что недопустимо.

Вопросы для обсуждения

1. Каковы достоинства и недостатки передачи аргумента по значению?
2. Каковы достоинства и недостатки передачи ссылки?
3. Каковы достоинства и недостатки передачи константной ссылки?
4. Каковы достоинства и недостатки возврата ссылки?
5. Должен ли игровой искусственный интеллект жульничать, чтобы соперник был более интересным?

Упражнения

1. Усовершенствуйте игру «Безумные библиотекари» из главы 5 с помощью ссылок и сделайте эту игру более эффективной.
2. Что неверно в следующей программе?

```
int main()
{
    int score;
    score = 1000;
    float& rScore = score;
    return 0;
}
```

3. Что неверно в следующей функции?

```
int& plusThree(int number)
{
    int threeMore = number + 3;
    returnthreeMore;
}
```

7 Указатели. Игра «Крестики-нолики 2.0»

Указатели — мощная составляющая языка C++. В некотором смысле они напоминают итераторы из библиотеки STL. Зачастую их можно использовать вместо ссылок. Но вдобавок указатели предоставляют функционал, которым не обладает никакая другая часть языка. В этой главе мы изучим основы работы с указателями, вы узнаете, для чего они особенно удобны. В частности, вы научитесь:

- объявлять и инициализировать указатели;
- разыменовывать указатели;
- работать с константами и указателями;
- передавать и возвращать указатели;
- работать с указателями и массивами.

Основы работы с указателями

Распространено мнение, что разобраться с указателями нелегко. На самом же деле они довольно незамысловаты. *Указатель* — это переменная, которая может содержать адрес из памяти. Соответственно, указатели обеспечивают возможность непосредственно и при этом эффективно взаимодействовать с компьютерной памятью. Подобно итераторам из библиотеки STL, они зачастую используются для доступа к содержимому других переменных. Но прежде, чем вы сможете как следует задействовать указатели в ваших игровых программах, нужно подробно изучить, как именно они работают.

НА ПРАКТИКЕ

Компьютерная память напоминает городской квартал. Но вместо домов, где жители хранят свое имущество, память разделена на ячейки, в которых можно хранить данные. Так же как в городском квартале, где дома тесно примыкают друг к другу и снабжены адресами, фрагменты компьютерной памяти тоже расположены вплотную и имеют адреса. Оказавшись в незнакомом квартале, вы можете свериться с адресом, записанным на бумажке, и попасть в конкретный дом (а возможно, и воспользоваться вещами, которые в нем хранятся). В компьютере можно взять указатель с адресом, чтобы обратиться к конкретному фрагменту памяти (и к данным, которые хранятся в этом фрагменте).

Знакомство с программой Pointing

Программа Pointing демонстрирует механизм работы с указателями. В этой программе создается переменная для хранения пользовательских очков, а потом — указатель, в котором хранится адрес этой переменной. Программа позволяет убедиться, что можно непосредственно изменять значение переменной, причем это изменение сразу же отразится и на указателе. Кроме того, в этой программе показано, как можно изменить значение переменной посредством указателя. Здесь вы увидите, как переставить указатель, чтобы направить его на совершенно другую переменную. Наконец, программа демонстрирует, что указатели не менее удобны и при работе с объектами. Результат выполнения программы показан на рис. 7.1.

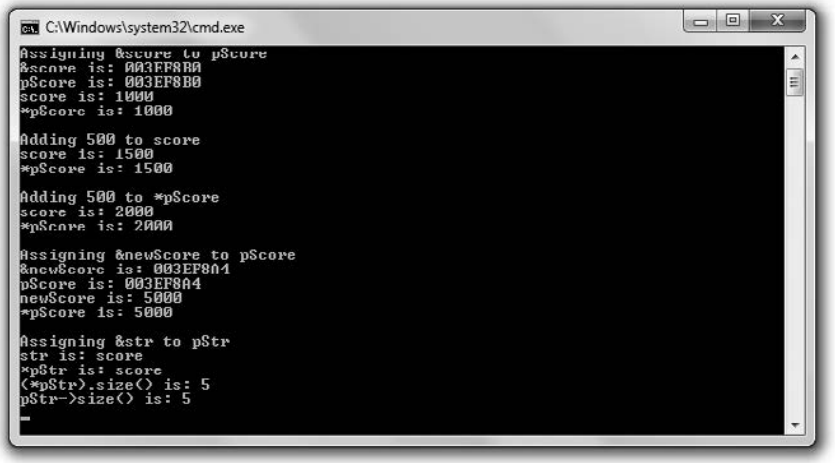


Рис. 7.1. Указатель pScore сначала направлен на переменную score, а затем на переменную newScore. Указатель pStr направлен на переменную str (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 7, имя файла — `pointing.cpp`.

```
// Программа Pointing
// Демонстрирует работу с указателями
#include <iostream>
#include <string>
using namespace std;
int main()
{
    int* pAPointer; // объявляем указатель
    int* pScore = 0; // объявляем и инициализируем указатель
    int score = 1000;
    pScore = &score; // присваиваем указателю pScore адрес переменной score
    cout << "Assigning &score to pScore\n";
```

```
    cout << "&score is: " <<&score << "\n"; // адрес переменной score
    cout << "pScore is: " << pScore << "\n"; // адрес, сохраненный в указателе
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n"; // значение, на которое
                                                // направлен указатель
    cout << "Adding 500 to score\n";
    score += 500;
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";
    cout << "Adding 500 to *pScore\n";
    *pScore += 500;
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";
    cout << "Assigning &newScore to pScore\n";
    int newScore = 5000;
    pScore = &newScore;
    cout << "&newScore is: " <<&newScore << "\n";
    cout << "pScore is: " << pScore << "\n";
    cout << "newScore is: " << newScore << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";
    cout << "Assigning &str to pStr\n";
    string str = "score";
    string* pStr = &str; // указатель на строковый объект
    cout << "str is: " << str << "\n";
    cout << "*pStr is: " << *pStr << "\n";
    cout << "(*pStr).size() is: " << (*pStr).size() << "\n";
    cout << "pStr->size() is: " << pStr->size() << "\n";
    return 0;
}
```

Объявление указателей

В первой инструкции в рамках функции `main()` я объявляю указатель `pAPointer`:

```
int* pAPointer; // объявление указателя
```

Поскольку указатели работают таким необычным образом, принято ставить перед именами переменных-указателей букву «p» (от англ. pointer), чтобы потом не забыть, что эта переменная — действительно указатель.

Подобно итератору, указатель объявляется как направленный на значение определенного типа. Так, указатель `pAPointer` направлен на `int`, и это означает, что он может указывать только на целочисленные значения. Например, этот указатель не может быть направлен на `float` или `char`. Можно переформулировать эту мысль так: единственный тип адресов, которые могут содержаться в `pAPointer`, — это `int`.

Чтобы объявить собственный указатель, начните с выбора типа того объекта, на который он будет направлен. Запишите этот тип, после него поставьте астериск,

а затем — имя указателя. При объявлении указателя можно поставить пробел слева или справа от него. Поэтому все следующие варианты кода: `int* pAPointer;`, `int *pAPointer;`, `int * pAPointer;` — объявляют указатель `pAPointer`.

ОСТОРОЖНО!

При объявлении указателя астериск применяется только к имени той единственной переменной, которая следует сразу за ним. Итак, следующая инструкция объявляет pScore как указатель на int, причем набранные очки здесь представлены целым числом (int):

```
int* pScore, score;
```

Еще раз, score — это не указатель! Это переменная типа int. Чтобы пояснить это различие, можно немного переставить пробелы и переписать приведенную инструкцию так:

```
int *pScore, score;
```

Правда, лучше всего объявлять каждый указатель в отдельной инструкции, как показано далее:

```
int* pScore;
intscore;
```

Инициализация указателей

Как и при работе с другими переменными, указатель можно инициализировать в той же инструкции, в которой он был объявлен. Именно это я делаю в следующей строке, где указателю `pScore` присваивается значение `0`:

```
int* pScore = 0; // объявление и инициализация указателя
```

Если указателю присвоен `0`, то это значение трактуется по-особенному. Можно сказать, оно переводится как «указатель в никуда». Программисты называют указатель, имеющий значение `0`, *нулевым указателем*. При объявлении указателя его всегда нужно инициализировать с каким-либо значением, даже если это значение равно нулю.

ПОДСКАЗКА

Многие программисты, желая создать нулевой указатель, присваивают ему значение NULL, а не 0. NULL — это константа, определенная во многих библиотечных файлах, в частности iostream.

Присваивание адресов указателям

Поскольку в указателях хранятся адреса объектов, нам понадобится способ получать адреса указателей. Например, это можно сделать, получив адрес имеющейся переменной и присвоив данный адрес указателю. Именно это я делаю в следующей строке, которая получает адрес переменной `score` и присваивает его указателю `pScore`:

```
pScore = &score; // присвоить указателю адрес переменной score
```

Чтобы получить адрес переменной `score`, я ставлю перед именем переменной символ `&`, *оператор взятия адреса*. Да, мы уже встречали символ `&` ранее, в значении

ссылочного оператора. Однако в описываемом здесь контексте оператор & получает адрес объекта.

После выполнения приведенной строки кода в указатель pScore будет записан адрес score. Можно сказать, что теперь pScore точно знает, в какой точке компьютерной памяти расположена переменная score. Далее мы можем использовать указатель pScore, чтобы получить значение score, и оперировать значением, сохраненным в score. Рисунок 7.2 схематически иллюстрирует взаимосвязь между pScore и score.

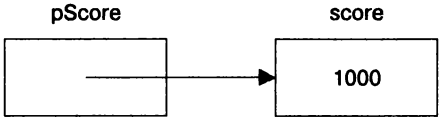


Рис. 7.2. Указатель pScore направлен на переменную score, в которой записано значение 1000

Чтобы убедиться, что указатель pScore содержит адрес score, я отображаю адрес переменной и значение указателя в следующих строках:

```
cout << "&score is: " <<&score << "\n"; // адрес переменной score
cout << "pScore is: " << pScore << "\n"; // адрес, сохраненный в указателе
```

Как показано на рис. 7.1, указатель pScore содержит значение 003EF8B0, представляющее собой адрес score. Конкретные адреса, отображаемые в программе Pointing, на моей и вашей машинах могут различаться. Самое важное, что значения pScore и &score равны.

Разыменование указателей

Как вы помните, для получения доступа к объекту, на который указывает итератор, этот итератор необходимо разыменовать. Для обращения к объекту, на который направлен указатель, также применяется операция разыменования. Разыменование указателя выполняется точно так же, как и в случае с итератором, — с помощью оператора *. Оператор разыменования продемонстрирован в действии в следующей строке, которая выводит на экран значение 1000, поскольку код *pScore обращается к значению, сохраненному в score:

```
cout << "*pScore is: " << *pScore << "\n\n"; // значение, на которое направлен указатель
```

Запомните: *pScore означает «объект, на который указывает pScore».

ОСТОРОЖНО!

Не разыменовывайте нулевой указатель — это может привести к катастрофическим последствиям!

В следующей строке кода я увеличиваю значение score на 500:

```
score += 500;
```

Теперь при отправке score в cout на экране отображается значение 1500, как и ожидалось. Когда я посылаю *pScore в cout, содержимое score вновь отправляется в cout, а на экран выводится значение 1500.

С помощью следующей строки кода я прибавляю 500 к тому значению, на которое направлен указатель pScore:

```
*pScore += 500;
```

Поскольку pScore указывает на score, предыдущая строка кода прибавляет 500 к score. Таким образом, при следующей отправке score в cout на экране отображается значение 2000. Затем, после отправки *pScore в cout... вы угадали, на экране опять отображается 2000.

ОСТОРОЖНО!

Не меняйте значение указателя, когда хотите изменить значение того объекта, на который он направлен. Например, если мне требуется прибавить 500 к значению int, на которое указывает pScore, то я не должен использовать следующую строку — она в корне ошибочна:

```
pScore += 500;
```

Код из этой строки прибавляет 500 к адресу, сохраненному в pScore, а не к тому значению, на которое изначально указывал pScore. В результате теперь pScore указывает на какой-то адрес, по которому может находиться что угодно. Разыменование такого указателя может вызвать катастрофические последствия.

Переприсваивание указателей

В отличие от ссылок указатели могут быть направлены на разные объекты в различные периоды жизненного цикла программы. Переприсваивание указателя не отличается от переприсваивания любой другой переменной. С помощью следующей строки кода я переприсваиваю pScore:

```
pScore = &newScore;
```

В результате pScore теперь указывает на newScore. Чтобы убедиться в этом, я отображаю адрес newScore, отправляя в cout&newScore, а затем задавая адрес, сохраненный в pScore. Обе инструкции отображают один и тот же адрес. После этого я посылаю newScore и *pScore в cout. В обоих случаях имеем на выходе 5000, так как обе инструкции обращаются к одному и тому же участку памяти, в котором сохранено это значение.

ОСТОРОЖНО!

Не меняйте значение, на которое направлен указатель, если хотите изменить сам указатель. Например, если я хочу, чтобы теперь pScore указывал на newScore, то не должен использовать следующую строку — она глубоко ошибочна:

```
*pScore = newScore;
```

Этот код просто изменяет значение, на которое в данный момент указывает pScore, но не меняет сам указатель pScore. Если переменная newScore равна 5000, то приведенный код эквивалентен *pScore = 5000;, а pScore указывает на ту же переменную, что и до присваивания.

Использование указателей на объекты

До сих пор программа Pointing работала только со значениями встроенного типа, примитива int. Но указатели можно столь же легко использовать и с объектами. Я продемонстрирую это в следующих строках, где создам str — строковый объект, равный score, а также pStr — указатель, направленный на этот объект:

```
string str = "score";
string* pStr = &str; // указатель на объект string
```

pStr — это указатель на string, то есть он может быть направлен на любой строковый объект. Иными словами, в указателе pStr может храниться адрес любого объекта string.

К объекту можно обратиться по указателю, воспользовавшись оператором разыменования. Именно это я и делаю в следующей строке:

```
cout << "*pStr is: " << *pStr << "\n";
```

Применив оператор разыменования в коде *pStr, я посылаю str (тот объект, на который указывает pStr) в cout. В результате на экране отображается текст score.

С помощью указателя можно вызвать функцию-член объекта, приниципиально эта операция не отличается от вызова функций-членов объекта посредством итератора. Для этого можно использовать оператор разыменования и оператор доступа к члену, что я и делаю в следующей строке:

```
cout << "(*pStr).size() is: " << (*pStr).size() << "\n";
```

Код (*pStr).size() означает: «Возьми результат разыменования указателя pStr, затем вызови функцию-член size() этого объекта». Поскольку указатель pStr направлен на объект string, равный "score", этот код возвращает 5.

ПОДСКАЗКА

Всякий раз при разыменовании указателя для доступа к члену данных или функции-члену заключайте разыменованный оператор в круглые скобки. Этим гарантируется, что оператор доступа к члену будет применен именно к тому объекту, на который направлен указатель.

Как и при работе с итераторами, с указателями можно использовать оператор ->, помогающий сделать код для доступа к членам объекта более удобочитаемым. Именно этот прием продемонстрирован в следующей строке:

```
cout << "pStr->size() is: " << pStr->size() << "\n";
```

Данная инструкция вновь отображает количество символов в объекте string, равном "score", однако теперь я могу подставить pStr->size() на место (*pStr).size(), значительно упростив восприятие кода.

Понятие об указателях и константах

Существуют еще некоторые аспекты работы указателей, с которыми нужно освоиться, прежде чем вы сможете эффективно пользоваться этими единицами в своих игровых программах. Так, чтобы ограничить функциональность указателя, с ним

можно употребить ключевое слово const. Подобные ограничения могут действовать в качестве предохранителей, а также прояснять логику вашей программы. Поскольку указатели довольно универсальны, некоторое ограничение их применения вполне согласуется с мантрой программирования «запрашивать только то, что действительно нужно».

Использование константного указателя

Как вы убедились ранее, в разные периоды жизненного цикла программы-указатели могут быть направлены на различные объекты. Однако, если применить ключевое слово const при объявлении и инициализации указателя, можно устранить такую многозначность указателя и жестко привязать его к тому объекту, на который он указывал в момент инициализации. Подобный указатель именуется *константным*. Иными словами, адрес, сохраненный в константном указателе, не может измениться — он постоянный. Вот пример создания константного указателя:

```
int score = 100;
int* const pScore = &score; // константный указатель
```

В данном коде создается константный указатель pScore, направленный на score. Чтобы создать константный указатель, нужно поставить ключевое слово const прямо перед именем указателя, когда вы его объявляете.

Константный указатель, как и любую константу, необходимо инициализировать сразу же после объявления. Следующий код абсолютно недопустим, он вызовет большую и жирную ошибку компиляции:

```
int* const pScore; // недопустимо – вы должны инициализировать константный указатель
```

Поскольку pScore — это константный указатель, он всегда будет направлен на один и только один фрагмент памяти. Следующий код также совершенно недопустим:

```
pScore = &anotherScore; // недопустимо – pScore не может указывать на иной объект
```

Хотя сам указатель pScore изменить невозможно, через этот указатель можно изменить то значение, на которое он направлен. Следующая строка кода абсолютно корректна:

```
*pScore = 500;
```

Запутались? Напрасно! Изменять само значение, на которое направлен константный указатель, вполне допустимо. Помните, что ограничение, налагаемое на константные указатели, таково: меняться не может только сам адрес, на который направлен константный указатель.

По своему функционалу константный указатель наверняка напоминает вам ссылку. Как и ссылка, он может относиться только к тому объекту, на который был направлен в момент инициализации.

ПОДСКАЗКА

Ничто не мешает вам использовать в своих программах константные указатели вместо ссылок, однако по возможности следует обходиться ссылками. Их синтаксис гораздо четче, чем у указателей, со ссылками ваш код будет намного легче читать.

Использование указателя на константу

Ранее вы убедились, что указатели можно применять для изменения тех значений, на которые они направлены. Однако если при объявлении указателя используется ключевое слово const, то функционал указателя можно ограничить так, чтобы он не мог изменять значение, на которое направлен. Подобный указатель называется *указателем на константу*. Вот пример объявления такого указателя:

```
const int* pNumber; // указатель на константу
```

Данный код объявляет указатель на константу, pNumber. Объявляя указатель на константу, мы ставим ключевое слово const прямо перед типом того значения, на которое будет направлен указатель.

Адрес присваивается указателю на константу точно так же, как это делалось ранее:

```
int lives = 3;
pNumber = &lives;
```

Правда, такой указатель нельзя использовать для изменения значения, на которое он указывает. Например, следующая строка недопустима:

```
*pNumber -= 1; // недопустимо – указатель на константу не может применяться
// для изменения того значения, на которое он указывает
```

Хотя указатель на константу и нельзя использовать для изменения того значения, на которое он направлен, сам этот указатель может меняться. Это означает, что на различных этапах жизненного цикла программы такой указатель может быть направлен на разные объекты. Например, следующий код абсолютно корректен:

```
const int MAX_LIVES = 5;
pNumber = &MAX_LIVES; // сам указатель может меняться
```

Использование константного указателя на константу

Константный указатель на константу совмещает ограничения, присущие константным указателям и указателям на константы. Это означает, что константный указатель на константу может быть направлен только на тот объект, на который он указывал при инициализации. При этом он не может использоваться для изменения объекта, на который указывает. Далее приведены объявление и инициализация такого указателя:

```
const int* const pBONUS = &BONUS; // константный указатель на константу
```

В предыдущем коде создается константный указатель на константу, он называется pBONUS и указывает на константу BONUS.

ПОДСКАЗКА

Как и указатель на константу, константный указатель на константу может быть направлен или на константное, или на неконстантное значение.

Константный указатель на константу нельзя переприсвоить. Следующий код недопустим:

```
pBONUS = &MAX_LIVES; // недопустимо -- pBONUS не может указывать
                     // на другой объект
```

Константный указатель на константу нельзя использовать для изменения того значения, на которое он направлен. Таким образом, следующий код недопустим:

```
*pBONUS = MAX_LIVES; // недопустимо – нельзя изменить
                     // значение посредством такого указателя
```

Функционально константный указатель на константу во многом подобен константной ссылке. Такая ссылка также может указывать лишь на то значение, на которое указывала при инициализации, причем не может использоваться для изменения этого значения.

ПОДСКАЗКА

Ничто не мешает вам использовать в своих программах константные указатели на константу вместо константных ссылок, однако по возможности следует обходиться константными ссылками. Их синтаксис гораздо четче, чем у указателей, со ссылками ваш код будет намного легче читать.

Константы и указатели: итог

Ранее я подробно рассказал об указателях и константах, поэтому хочу подвести промежуточный итог, так сказать, сделать выжимку из этих концепций. Далее я приведу три примера того, как можно использовать ключевое слово `const` при объявлении указателей:

- `int* const p = &i;;`
- `const int* p;;`
- `const int* const p = &I;.`

В первом примере объявляется и инициализируется константный указатель. Константный указатель может быть направлен только на тот объект, на который он указывал в момент инициализации. Значение — адрес в памяти, — сохраненное в таком указателе, является константным и изменяться не может. Константный указатель может быть направлен только на неконстантное значение — на константу он указывать не может.

Во втором примере объявляется указатель на константу. Указатель на константу не может использоваться для изменения того значения, на которое направлен. В течение жизненного цикла программы указатель на константу может быть направлен на различные объекты. Указатель на константу может быть направлен на константное или неконстантное значение.

В третьем примере объявляется константный указатель на константу. Константный указатель на константу может быть направлен лишь на то значение, на которое он указывал в момент инициализации. При инициализации константный указатель на константу может указывать либо на константное, либо на неконстантное значение.

Передача указателей

Конечно, аргументы лучше передавать с помощью ссылок (их синтаксис сравнительно прост), но иногда может потребоваться передавать аргументы и посредством указателей. Предположим, вы используете графический движок, возвращающий указатель на 3D-объект. Если вы хотите, чтобы этот объект использовала другая функция, то для повышения эффективности будет целесообразно передать функции указатель на этот объект. Следовательно, уметь передавать указатели не менее важно, чем правильно передавать ссылки.

Знакомство с программой Swap Pointer Version

Программа Swap Pointer Version работает аналогично программе Swap из главы 6, за тем исключением, что в программе Swap Pointer Version вместо ссылок используются указатели. В программе Swap Pointer Version определяются две переменные. В одной содержится мой жалкий результат, а в другой — ваш, чемпионский. Отобразив эти результаты, программа вызывает функцию, которая меняет их местами. Поскольку в функцию передаются лишь копии значений счета, а не сами значения, с оригиналами переменных ничего не происходит. Далее программа вызывает еще одну функцию обмена. На этот раз применяются константные указатели, с помощью которых происходит успешный обмен значениями между оригинальными переменными (вы вновь получаете солидный результат, а мне остается довольствоваться малым). На рис. 7.3 эта программа продемонстрирована в действии.

Рис. 7.3. При передаче указателей функция может изменять переменные, находящиеся вне ее области видимости (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 7, имя файла — `swap_pointer_ver.cpp`.

```
// Программа Swap Pointer
// Демонстрирует передачу константных указателей для изменения переменных-аргументов
#include <iostream>
using namespace std;
```

```
void badSwap(int x, int y);
void goodSwap(int* const pX, int* const pY);
int main()
{
    int myScore = 150;
    int yourScore = 1000;
    cout << "Original values\n";
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";
    cout << "Calling badSwap()\n";
    badSwap(myScore, yourScore);
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";
    cout << "Calling goodSwap()\n";
    goodSwap(&myScore, &yourScore);
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n";
    return 0;
}
void badSwap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
void goodSwap(int* const pX, int* const pY)
{
    // Сохраняем в temp значение, на которое указывает pX
    int temp = *pX;
    // сохраняем значение, на которое указывал pY,
    // по адресу, на который указывает pX
    *pX = *pY;
    // сохраняем значение, на которое изначально указывал pX,
    // по адресу, на который указывает pY
    *pY = temp;
}
```

Передача по значению

Объявив и инициализировав переменные myScore и yourScore, я посылаю их в cout. Как и следовало ожидать, на экран выводятся значения 150 и 1000. Далее вызываю функцию badSwap(), которая передает оба аргумента по значению. Таким образом, когда я вызываю функцию с помощью следующей строки кода, копии моих переменных myScore и yourScore записываются в параметры x и y:

```
badSwap(myScore, yourScore);
```

В частности, переменной x присваивается значение 150, а переменной y — значение 1000. Таким образом, что бы я ни делал с x и y в функции badSwap(), это никак не отразится на переменных myScore и yourScore.

После выполнения функции badSwap() параметры x и y обмениваются значениями: x получает 1000, а y — 150. Но по завершении функции x и y выходят из области видимости. Управление программой возвращается к функции main(), в которой переменные myScore и yourScore не менялись. Когда я посылаю myScore и yourScore в cout, на экране вновь отображаются значения 150 и 1000. Обидно, но у меня снова мало очков, а у вас — очень много.

Передача константного указателя

Вы уже знаете, как предоставлять функции доступ к переменным с помощью передачи ссылок. Такую задачу можно решить и с помощью указателей. Передавая указатель, вы сообщаете функции лишь адрес объекта. Такой прием может быть достаточно эффективен, особенно если вы работаете с объектами, занимающими большие фрагменты памяти. Передавая указатель, вы как будто отправляете другу ссылку на сайт, а не пересылаете ему весь этот сайт целиком.

Прежде чем вы сможете передать функции указатель, потребуется задать параметры функции как указатели. Именно это я делаю в заголовке функции goodSwap():

```
void goodSwap(int* const pX, int* const pY)
```

Таким образом, pX и pY — это константные указатели, каждый из них принимает адрес в памяти. Я сделал параметры константными указателями, потому что, хотя и планирую изменять те значения, на которые они направлены, при этом не собираюсь изменять сами указатели. Ситуация аналогична работе со ссылками: можно менять значение, на которое указывает ссылка, но не ее саму.

В функции main() я передаю адреса myScore и yourScore, когда вызываю функцию goodSwap() следующей строкой кода:

```
goodSwap(&myScore, &yourScore);
```

Обратите внимание: я посылаю адреса переменных в функцию goodSwap(), пользуясь *оператором взятия адреса*. При передаче объекта указателю мы пересылаем адрес этого объекта.

В функции goodSwap() в указателе pX хранится адрес myScore, а в указателе pY — адрес yourScore. Все операции, выполняемые с *pX, будут применяться к myScore, аналогичная взаимосвязь существует между *pY и yourScore.

Первая строка функции goodSwap() принимает значение, на которое направлен указатель pX, и присваивает это значение переменной temp:

```
int temp = *pX;
```

Поскольку pX указывает на myScore, temp принимает значение 150.

Следующая строка кода присваивает значение, на которое указывает pY, тому объекту, на который указывает pX:

```
*pX = *pY;
```

Данная инструкция копирует значение, сохраненное в yourScore, — 1000 и присваивает его тому фрагменту памяти, в котором сохранено myScore. В результате переменная myScore получает значение 1000.

Последняя инструкция в данной функции сохраняет значение переменной `temp` — 150 по адресу, на который указывает `pY`:

```
*pY = temp;
```

Поскольку `pY` указывает на `yourScore`, `yourScore` получает значение 150.

После окончания функции управление программой возвращается к функции `main()`, где я посылаю `myScore` и `yourScore` в `cout`. На этот раз отображаются значения 1000 и 150. Переменные поменялись значениями. Все хорошо, что хорошо кончается!

ПОДСКАЗКА

Также существует возможность передавать константный указатель на константу. Технически эта операция довольно сильно напоминает передачу константной ссылки, то есть эффективную передачу объекта, изменять который не требуется. Я адаптировал для такой задачи программу Inventory Displayer из главы 6. Код программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 7, имя файла — inventory_displayer_pointer_ver.cpp.

Возврат указателей

До появления ссылок у программистов был только один вариант эффективного возврата объектов от функций — с помощью указателей. Хотя при использовании функций получается гораздо более ясный синтаксис, чем при применении указателей, вы все равно можете столкнуться с необходимостью возвращать объекты с помощью указателей.

Знакомство с программой Inventory Pointer

Программа Inventory Pointer демонстрирует возврат указателей. С помощью возвращаемых указателей программа отображает и даже изменяет значения в векторе, содержащем элементы снаряжения героя. На рис. 7.4 показан результат выполнения программы.

Рис. 7.4. Функция возвращает указатель (а не объект string) на каждый из элементов в снаряжении героя (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 7, имя файла — `inventory_pointer.cpp`.

```
// Программа Inventory Pointer
// Демонстрирует возврат указателя
#include <iostream>
#include <string>
#include <vector>
using namespace std;
// возвращает указатель на строковый элемент
string* ptrToElement(vector<string>* const pVec, int i);
int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");
    // отображает строковый элемент, на который направлен возвращенный указатель
    cout << "Sending the object pointed to by returned pointer to cout:\n";
    cout << *(ptrToElement(&inventory, 0)) << "\n\n";
    // присваивает один указатель другому – малозатратная операция
    cout << "Assigning the returned pointer to another pointer.\n";
    string* pStr = ptrToElement(&inventory, 1);
    cout << "Sending the object pointed to by new pointer to cout:\n";
    cout << *pStr << "\n\n";
    // копирует строковый объект – затратная операция присваивания
    cout << "Assigning object pointed to by pointer to a string object.\n";
    string str = *(ptrToElement(&inventory, 2));
    cout << "Sending the new string object to cout:\n";
    cout << str << "\n\n";
    // изменение строкового объекта посредством возвращенного указателя
    cout << "Altering an object through a returned pointer.\n";
    *pStr = "Healing Potion";
    cout << "Sending the altered object to cout:\n";
    cout << inventory[1] << endl;
    return 0;
}
string* ptrToElement(vector<string>* const pVec, int i)
{
    // возвращает адрес строкового объекта, расположенного на позиции i
    // в том векторе, на который направлен указатель pVec
    return &((*pVec)[i]);
}
```

Возврат указателя

Прежде чем вы сможете вернуть указатель от функции, необходимо указать, что вы собираетесь его возвращать. Именно это я делаю в заголовке функции `ptrToElement()`:

```
string* ptrToElement(vector<string>* const pVec, int i)
```

Начиная заголовок со `string*`, я указываю, что функция будет возвращать указатель на объект `string` (а не объект `string` как таковой). Чтобы обозначить, что функция возвращает указатель на объект определенного типа, поставьте астериск после имени возвращаемого типа.

В теле функции `ptrToElement()` содержится всего одна инструкция, возвращающая указатель на тот элемент, который расположен на позиции `i` в векторе (на этот вектор направлен указатель `pVec`):

```
return &((*pVec)[i]);
```

Данная инструкция возврата может показаться немного мудреной, поэтому я подробно разберу ее. Всякий раз, встречая сложное выражение, интерпретируйте его именно так, как это делал бы компьютер, — начиная с середины. Начинаем с конструкции `(*pVec)[i]`, означающей «элемент, расположенный на позиции `i` в том векторе, на который направлен указатель `pVec`». Если применить к выражению *оператор взятия адреса* (`&`), то полученный адрес будет соответствовать адресу элемента, расположенного на позиции `i` в том векторе, на который направлен указатель `pVec`.

ПОДСКАЗКА

Хотя возврат указателя порой весьма эффективен при отправке информации обратно той функции, которая выполняла вызов, никогда не возвращайте указатель на объект, расположенный вне области видимости данной функции. Например, следующая функция возвращает такой указатель, который, сработав, приведет к аварийному завершению программы:

```
string* badPointer()
{
string local = "This string will cease to exist once the function ends.";
string* pLocal = &local;
return pLocal;
}
```

Программа может аварийно завершиться, поскольку функция badPointer() возвращает указатель на строку, которой после завершения функции уже не существует. Указатель на несуществующий объект называется висячим указателем. Результаты попытки разыменовать висячий указатель могут быть катастрофическими. Один из способов, позволяющих исключить появление висячих указателей, — никогда не возвращать указатель на локальную переменную.

Использование возвращенного указателя для отображения значения

Создав вектор элементов `inventory`, я отображаю значение, соответствующее возвращенному указателю:

```
cout << *(ptrToElement(&inventory, 0)) << "\n\n";
```

Этот код вызывает функцию `ptrToElement()`, которая возвращает указатель на элемент `inventory[0]` (не забывайте, функция `ptrToElement()` не возвращает копию одного из элементов, содержащихся в векторе `inventory`, а просто дает указатель

на один из этих элементов). Затем эта строка кода посылает в cout тот объект string, на который направлен указатель. В результате на экране отображается слово sword.

Присваивание указателю возвращенного указателя

Далее с помощью следующей строки кода я присваиваю возвращенный указатель другому указателю:

```
string* pStr = ptrToElement(&inventory, 1);
```

Вызов функции prtToElement() возвращает указатель на элемент inventory[1]. Инструкция присваивает этот указатель указателю pStr. Такое присваивание очень эффективно, поскольку при подобной операции не требуется копировать объект string.

Чтобы вы могли нагляднее представить себе результаты выполнения этой строки кода, обратите внимание на рис. 7.5, где схематически представлен указатель pStr после присваивания. Эта иллюстрация довольно абстрактна, поскольку вектор inventory не содержит строковые литералы "sword", "armor" и "shield" — он содержит объекты string.

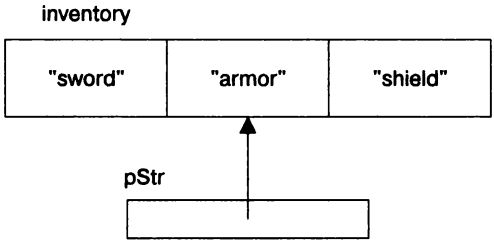


Рис. 7.5. Указатель pStr направлен на элемент, расположенный в векторе inventory на позиции 1

Далее я посылаю *pStr в cout, и на экране отображается слово armor.

Присваивание переменной значения, на которое указывает возвращенный указатель

Далее я присваиваю переменной значение, на которое указывает возвращенный указатель:

```
stringstr = *(ptrToElement(&inventory, 2));
```

Вызов функции ptrToElement() возвращает указатель на содержащийся в векторе элемент inventory[2]. Однако предыдущая инструкция не присваивает этот указатель переменной str — это просто невозможно, так как str — это объект string.

Вместо этого компьютер бесшумно делает копию объекта string, на который направлен указатель, после чего присваивает этот объект переменной str. Чтобы прочно усвоить этот момент, рассмотрите рис. 7.6, где схематически представлены результаты данной операции присваивания.

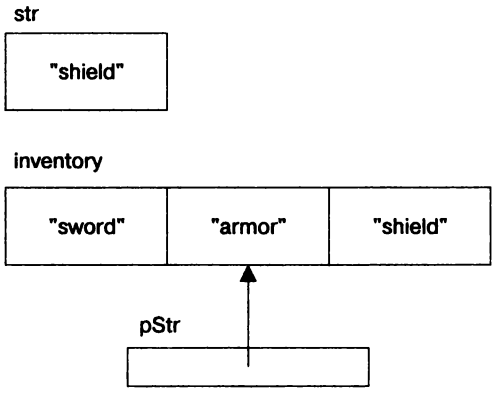


Рис. 7.6. str — это новый строковый объект, независимый от inventory

Подобное присваивание, сопряженное с копированием объекта, является более затратной операцией, чем присваивание одного указателя другому. Иногда копирование объекта не доставляет никаких неудобств, но необходимо учитывать дополнительные издержки, возникающие при таком варианте присваивания, и по возможности их избегать.

Изменение объекта посредством возвращенного указателя

Существует возможность изменять тот объект, на который направлен возвращенный указатель. Это означает, что я могу изменить снаряжение героя с помощью pStr:

```
*pStr = "HealingPotion";
```

Поскольку указатель pStr направлен на элемент, расположенный в позиции 1 в векторе inventory, данный код изменяет inventory[1] и этот элемент становится равен "HealingPotion". Чтобы убедиться в этом, я отображаю элемент с помощью следующей строки кода:

```
cout<<inventory[1] <<endl;
```

Действительно, на экран выводится HealingPotion.

Абстрактное представление этой операции показано на рис. 7.7, где схематически изображено состояние переменных после операции присваивания.

ПОДСКАЗКА

Если вы хотите защитить объект, на который направлен указатель, — ограничьте сам указатель. Возвращайте либо указатель на константу, либо константный указатель на константу.

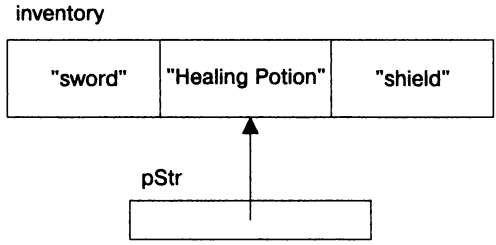


Рис. 7.7. inventory[1] изменяется с помощью возвращенного указателя, сохраненного в pStr

Понятие о взаимоотношениях указателей и массивов

На практике указатели тесно связаны с массивами. Фактически имя массива — это просто константный указатель на первый элемент, содержащийся в этом массиве. Поскольку все элементы массива хранятся в одном непрерывном блоке памяти, имя массива можно использовать как указатель для доступа к элементам в случайном порядке. Такие взаимосвязи серьезно влияют на детали передачи и возврата массивов, о чем мы поговорим в дальнейшем.

Знакомство с программой Array Passer

Программа Array Passer создает массив рекордов, а затем отображает их, используя имя массива как константный указатель. Далее программа передает специальной функции имя массива в качестве константного указателя, и функция увеличивает очки. Наконец, программа передает имя массива еще одной функции, но уже как константный указатель на константу, и на экран выводится новая таблица рекордов. Результат выполнения программы показан на рис. 7.8.

Рис. 7.8. Рекорды из списка отображаются на экране, модифицируются и передаются функциям, при этом имя массива используется в качестве константного указателя (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 7, имя файла — array_passer.cpp.

```
// Программа Array Passer
// Демонстрирует взаимоотношение между указателями и массивами
#include <iostream>
using namespace std;
void increase(int* const array, const int NUM_ELEMENTS);
void display(const int* const array, const int NUM_ELEMENTS);
int main()
{
    cout << "Creating an array of high scores.\n\n";
    const int NUM_SCORES = 3;
    int highScores[NUM_SCORES] = {5000, 3500, 2700};
    cout << "Displaying scores using array name as a constant pointer.\n";
    cout << *highScores << endl;
    cout << *(highScores + 1) << endl;
    cout << *(highScores + 2) << "\n\n";
    cout << "Increasing scores by passing array as a constant pointer.\n\n";
    increase(highScores, NUM_SCORES);
    cout << "Displaying scores by passing array as a constant pointer to a
constant.\n";
    display(highScores, NUM_SCORES);
    return 0;
}
void increase(int* const array, const int NUM_ELEMENTS)
{
    for (int i = 0; i < NUM_ELEMENTS; ++i)
    {
        array[i] += 500;
    }
}
void display(const int* const array, const int NUM_ELEMENTS)
{
    for (int i = 0; i < NUM_ELEMENTS; ++i)
    {
        cout<<array[i] <<endl;
    }
}
```

Использование имени массива в качестве константного указателя

Поскольку имя массива — это константный указатель на первый элемент данного массива, можно разыменовать массив, чтобы получить его первый элемент. Именно это я делаю после создания массива рекордов, который называется highScores:

```
cout<< *highScores<<endl;
```

Я разыменовываю массив highScores, чтобы получить доступ к его первому элементу и послать этот элемент в cout. В результате на экране отображается 5000.

Можно в случайном порядке обращаться к элементам массива по имени этого массива, выступающего в качестве константного указателя, при этом применяется обычная операция сложения. Вам всего лишь потребуется приплюсовать номер позиции интересующего вас элемента к имени массива, прежде чем разыменовать массив. Эта операция проще, чем может показаться. Например, в следующей строке я обращаюсь к результату, расположенному на позиции 1 в массиве highScores, и получаю результат 3500:

```
cout<< *(highScores + 1) <<endl;
```

В данном коде *(highScores + 1) эквивалентно highScores[1]. В обоих случаях я возвращаю элемент, расположенный в позиции 1 в массиве highScores.

С помощью следующей строки кода я обращаюсь к результату, расположенному на позиции 2 в массиве highScores, в результате отображается 2700:

```
cout<< *(highScores + 2) <<endl;
```

В этом коде *(highScores + 2) эквивалентно highScores[2]. В обоих случаях я возвращаю элемент, расположенный в позиции 2 в массиве highScores. В принципе, любое выражение вида arrayName[i] можно переписать как *(arrayName + i), где arrayName — имя массива.

Передача и возврат массивов

Поскольку имя массива — это константный указатель, такое имя можно использовать для эффективной передачи массива к функции. Именно это я делаю в следующей строке кода, которая передает функции increase() константный указатель на первый элемент массива, а также количество элементов в массиве:

```
increase(highScores, NUM_SCORES);
```

ПОДСКАЗКА

Передавая массив к функции, обычно бывает целесообразно также сообщать количество элементов, входящих в этот массив. Функция сможет ориентироваться на этот показатель, чтобы программа не пыталась обращаться к несуществующим элементам.

Как понятно из заголовка функции increase(), имя массива принимается в качестве константного указателя:

```
void increase(int* const array, const int NUM_ELEMENTS)
```

Тело функции увеличивает на 500 каждый из результатов:

```
for (int i = 0; i < NUM_ELEMENTS; ++i)
{
    array[i] += 500;
}
```

Я работаю с array как с любым массивом и использую для доступа к любому из его элементов оператор индексации. Я также мог бы оперировать array как указателем и заменить выражение array[i] += 500 на *(array + i) += 500, но остановлюсь на более удобочитаемом варианте.

Когда функция increase() завершится, управление возвращается к функции main(). Чтобы убедиться в том, что функция increase() действительно увеличила значения рекордов, я вызываю функцию, отображающую очки:

```
display(highScores, NUM_SCORES);
```

Функция display() также принимает highScore в качестве указателя. Правда, как можно убедиться по заголовку функции, highScore в данном случае является константным указателем на константу:

```
void display(const int* const array, const int NUM_ELEMENTS)
```

Передавая массив таким образом, я защищаю его от изменений. Поскольку мне требуется всего лишь отобразить все элементы, это отличный выход.

Наконец, выполняется тело функции display() и на экран выводятся все результаты. Мы можем убедиться, что каждый из них увеличился на 500.

ПОДСКАЗКА

Можно передать функции си-строку, как и любой другой массив. Кроме того, функции можно передать и строковый литерал как константный указатель на константу.

Поскольку имя массива — это указатель, можно вернуть массив по имени массива точно так же, как вы поступили бы с любым другим указателем на объект.

Знакомство с программой «Крестики-нолики 2.0»

Проект для этой главы представляет собой модифицированную версию игры из главы 6 — «Крестики-нолики». С пользовательской точки зрения игра «Крестики-нолики 2.0» совершенно не изменилась по сравнению с оригиналом, поскольку все нововведения сделаны на внутрисистемном уровне: я заменил ссылки на указатели. Таким образом, все объекты — в частности, игровое поле — передаются не как ссылки, а как константные указатели. Такая перемена влечет за собой определенные последствия — в частности, теперь мы не передаем сам объект игрового поля, а лишь сообщаем его адрес.

Код данной версии программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 7, имя файла — tic-tac-toe2.cpp. Я не буду подробно разбирать здесь ее код, поскольку во многом он остался без изменений. Но пусть изменений в нем и немного, все они очень важные. Эта программа хороша в качестве учебного материала: хотя в работе и следует стараться обходиться одними ссылками, при случае вы должны не менее умело обращаться и с указателями.

Резюме

В этой главе были изучены следующие концепции.

- Компьютерная память организована строго упорядоченным образом — каждый фрагмент памяти имеет собственный уникальный адрес.
- Указатель — это переменная, содержимым которой является адрес из памяти.
- Во многих отношениях указатели функционально подобны итераторам из библиотеки STL. В частности, как и итераторы, указатели применяются для непрямого доступа к объекту.
- Чтобы объявить указатель, необходимо записать его тип, далее поставить астериск, затем — имя указателя.
- Зачастую программисты начинают имена переменных-указателей с буквы «p», чтобы не забыть, что данная переменная действительно является указателем.
- Точно так же, как и итератор, указатель уже при объявлении указывает на значение конкретного типа.
- Рекомендуется инициализировать указатель непосредственно после объявления.
- Указатель, которому присвоено значение `0`, называется нулевым.
- Чтобы получить адрес переменной, поставьте перед именем этой переменной оператор взятия адреса (`&`).
- Когда указатель содержит адрес объекта, принято говорить, что он указывает на объект.
- В отличие от ссылок указатели можно переприсваивать. Это означает, что в разные периоды жизненного цикла программы указатель может быть направлен на разные объекты.
- Точно так же, как и при работе с итераторами, указатель можно разыменовать, чтобы получить доступ к объекту, на который он направлен. Для этого нужно поставить перед указателем оператор разыменования `*`.
- Точно так же, как и при работе с итераторами, при работе с указателями можно использовать оператор `->`, позволяющий более удобочитаемо оформлять в коде обращение к членам данных и функциям-членам объектов.
- Константный указатель может быть направлен лишь на тот объект, на который он указывал в момент инициализации. Чтобы объявить константный указатель, нужно поставить прямо перед именем указателя ключевое слово `const`, вот так: `int* const p = &i;`.
- Указатель на константу можно использовать для изменения того значения, на которое он направлен. Чтобы объявить указатель на константу, нужно поставить прямо перед типом указателя ключевое слово `const`, вот так: `const int* p;`.
- Константный указатель на константу может быть направлен только на то значение, на которое указывал при инициализации, причем не может использоваться для изменения этого значения. Для объявления константного указателя

на константу нужно поставить ключевое слово `const` как перед именем типа указателя, так и перед именем самого указателя, вот так: `const int* const p = &I;`.

- Передавать указатели целесообразно для повышения эффективности или для обеспечения непосредственного доступа к объекту.
- Если указатель применяется для повышения эффективеости, то нужно передавать указатель на константу или константный указатель на константу, чтобы объект, на который вы таким образом ссылаетесь, нельзя было изменить через указатель.
- Висячий указатель — это указатель, направленный на недействительный адрес в памяти. Висячие указатели часто возникают из-за удаления объекта, на который данный указатель был направлен. Разыменование такого указателя может привести к катастрофическому результату.
- Можно возвращать указатель от функции, но будьте осторожны, чтобы случайно не вернуть висячий указатель.

Вопросы и ответы

1. *Чем указатель отличается от переменной, на которую он направлен?*

   Содержимое указателя — это адрес в памяти. Если указатель направлен на переменную, то в нем хранится адрес этой переменной.

2. *Почему стоит хранить в указателе адрес переменной, которая уже существует?*

   Значительное преимущество хранения адреса существующей переменной в указателе заключается в следующем: для повышения эффективности можно передавать указатель на переменную, а не передавать саму переменную по значению.

3. *Всегда ли указатель должен быть направлен на существующую переменную?*

   Нет. При необходимости можно создать указатель, направленный на фрагмент памяти, не имеющий определенного имени. Мы подробнее поговорим о динамическом выделении памяти в главе 9.

4. *Почему по возможности следует передавать переменные с помощью ссылок, а не с помощью указателей?*

   Потому что ссылки — это сладкий синтаксический сахар. Передача ссылки или указателя — это эффективный способ предоставления доступа к объектам, но для работы с указателями требуется дополнительный синтаксис (в частности, оператор разыменования), нужный для доступа к самому объекту.

5. *Почему нужно инициализировать указатель сразу или вскоре после объявления?*

   Потому что результаты разыменования неинициализированного указателя могут быть катастрофическими, вплоть до аварийного завершения программы.

6. *Что такое висячий указатель?*

   Это указатель, направленный на недопустимый адрес в памяти — по этому адресу могут находиться абсолютно любые данные.

7. *Чем опасны висячие указатели?*

   Как и в случае с неинициализированным указателем, использование висячего указателя может привести к катастрофическим результатам, вплоть до аварийного завершения программы.

8. *Почему может потребоваться инициализировать указатель в значении 0?*

   Инициализируя указатель в значении 0, мы получаем нулевой указатель — фактически указатель, который ни на что не направлен.

9. *Значит, разыменовывать нулевой указатель безопасно, верно?*

   Нет! Хотя при программировании и рекомендуется присваивать значение 0 тому указателю, который не направлен на объект, разыменование нулевого указателя не менее опасно, чем разыменование висячего указателя.

10. *Что произойдет при разыменовании нулевого указателя?*

    Как и при разыменовании висячего или неинициализированного указателя, результаты могут быть непредсказуемыми. Скорее всего, программа аварийно завершится.

11. *Для чего удобны нулевые указатели?*

    Часто функция возвращает такой указатель, сигнализируя, что операция завершилась неудачно. Например, если функция должна вернуть указатель на объект, представляющий собой графический экран, но функции не удается инициализировать этот экран, она может вернуть нулевой указатель.

12. *Если при объявлении указателя используется ключевое слово const, как это влияет на указатель?*

    Все зависит от конкретного случая. Обычно ключевое слово `const` используется при объявлении указателя для того, чтобы ограничить возможности этого указателя.

13. *Какие ограничения можно наложить на указатель, объявив его с ключевым словом const?*

    Например, можно сделать так, чтобы указатель мог быть направлен лишь на тот объект, на который он указывал при инициализации, или чтобы через него нельзя было изменять значение того объекта, на который он указывает, или применить сразу оба упомянутых ограничения.

14. *Почему может потребоваться ограничить возможности указателя?*

    В целях безопасности. Например, если вы работаете с объектом, который, насколько вам известно, определенно не придется изменять.

15. *Указателям каких типов можно присваивать константные значения?*

    Указателям на константу или константным указателям на константу.

16. *Как можно безопасно свернуть указатель от функции?*

    Один из вариантов — вернуть указатель на объект, который вы получили от вызывающей функции. Таким образом, вы возвращаете указатель на объект, который существует в коде, сделавшем вызов. В главе 9 будет изучен еще один способ такого рода, когда мы будем говорить о динамической памяти.

Вопросы для обсуждения

1. Перечислите достоинства и недостатки, связанные с передачей указателя.
2. В каких ситуациях нужен константный указатель?
3. В каких ситуациях нужен указатель на константу?
4. В каких ситуациях нужен константный указатель на константу?
5. В каких ситуациях нужен неконстантный указатель на неконстантный объект?

Упражнения

1. Напишите программу с указателем, направленным на указатель объекта `string`. С помощью указателя на указатель вызовите функцию-член `size()` объекта `string`.
2. Перепишите проект «Безумные библиотекари» из главы 5 так, чтобы сюжетной функции не передавались объекты `string`. Вместо этого ваша функция должна принимать указатели на объекты `string`.
3. Будут ли одинаковы все три адреса в памяти, отображаемые следующей программой? Объясните, что происходит в этом коде.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 10;
    int& b = a;
    int* c = &b;
    cout <<&a << endl;
    cout <<&b << endl;
    cout <<&(*c) << endl;
    return 0;
}
```

8 Классы. Игра «Тамагочи»

Объектно-ориентированное программирование (ООП) — это целая философия программирования. Это современная методология, применяемая при создании абсолютного большинства игр (а также многих других программ для коммерческого ПО). В ООП мы определяем различные типы объектов, которые обладают конкретными взаимосвязями. Благодаря этим взаимосвязям обеспечивается взаимодействие объектов. Мы уже работали с объектами библиотечных типов, но одно из ключевых свойств ООП — возможность определения собственных типов, на базе которых вы можете создавать свои объекты. В этой главе будет рассмотрено, как определять собственные типы и создавать объекты на их основе. В частности, вы научитесь:

- создавать новые типы, определяя классы;
- объявлять члены данных и функции-члены класса;
- инстанцировать объекты из классов;
- устанавливать уровни доступа к членам;
- объявлять статические члены данных и функции-члены.

Определение новых типов

Любые игровые элементы, будь то инопланетные корабли, отравленные стрелы или злобные курочки-мутанты, — это разнообразные объекты. К счастью, язык C++ позволяет представлять игровые сущности в качестве программных объектов, оснащенных членами данных и функциями-членами. Эти объекты функционально очень подобны тем, которые мы уже рассмотрели, например объектам `string` и `vector`. Но чтобы использовать в программе объект нового вида, скажем злобную курочку-мутанта, сначала требуется определить тип для этого объекта.

Знакомство с программой Simple Critter

В программе Simple Critter мы определим совершенно новый тип `Critter` для создания виртуальных зверушек-любимцев (тамагочи). Программа применяет этот новый тип для создания двух объектов `Critter`. Затем она присваивает каждому

любимцу уровень голода. Наконец, каждый тамагочи издает характерное приветствие и сообщает, насколько он проголодался (выдает свой уровень голода). Результат выполнения программы показан на рис. 8.1.

Рис. 8.1. Каждый тамагочи приветствует игрока, а затем сообщает, насколько он (тамагочи) голоден (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 8, имя файла — `simple_critter.cpp`.

```
// Программа Simple Critter
// Демонстрирует создание нового типа
#include <iostream>
using namespace std;
classCritter // определение нового класса – определяет новый тип Critter
{
public:
    intm_Hunger; // член данных
    voidGreet(); // прототип члена функции
};
voidCritter::Greet() // определение члена функции
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n";
}
int main()
{
    Critter crit1;
    Critter crit2;
    crit1.m_Hunger = 9;
    cout << "crit1's hunger level is " << crit1.m_Hunger << ".\n";
    crit2.m_Hunger = 3;
    cout << "crit2's hunger level is " <<crit2.m_Hunger << ".\n\n";
```

```
    crit1.Greet();
    crit2.Greet();
    return 0;
}
```

Определение класса

Чтоб создать новый тип, можно определить *класс*. Класс — это код, объединяющий в группу ряд членов данных и функций-членов. На основе класса создаются отдельные объекты, в каждом из которых имеются собственные копии членов данных; кроме того, объект обладает доступом ко всем функциям-членам класса. Таким образом, класс напоминает чертеж. Как на чертеже описана вся конструкция здания, так в классе определяется структура объекта. Умелый прораб может выстроить множество домов по одному и тому же чертежу, а хороший программист-игровик — создать множество объектов из имеющегося класса. Этот теоретический материал будет проще усвоить на примере реального кода. Я начинаю определение класса в программе Simple Critter с помощью следующей строки кода, определяющей класс `Critter`:

```
classCritter // определение нового класса – определяет новый тип Critter
```

Чтобы определить класс, начинаем строку с ключевого слова `class`, за которым идет имя класса. Принято начинать имена классов с прописной буквы. Тело класса заключается в фигурные скобки, а весь код класса завершается точкой с запятой.

Объявление членов данных

При определении класса можно объявлять члены данных класса, которые будут представлять свойства объекта. Я присваиваю тамагочи всего одно свойство — голод. Голод будет реализован как диапазон значений, причем это свойство может принимать целочисленное значение из данного диапазона. Итак, я объявляю член данных `m_Hunger` типа `int`:

```
int m_Hunger; // член данных
```

Таким образом, у каждого объекта `Critter` будет собственный уровень голода, представленный специальным членом данных этого объекта, который называется `m_Hunger`. Обратите внимание: я ставлю перед именем члена данных префикс `m_`. Некоторые программисты следуют этому соглашению, чтобы имена членов данных в коде были легко узнаваемыми.

Объявление функций-членов

В определении класса также объявляются функции-члены, представляющие возможности объекта. Я присваиваю тамагочи всего одну такую функцию — возможность поприветствовать пользователя и сообщить ему свой уровень голода. Для этого я объявляю функцию-член `Greet()`:

```
void Greet(); // прототип члена функции
```

Таким образом, каждый объект Critter сможет сказать «Привет» и с помощью функции-члена Greet() сообщить, насколько он голоден. Принято начинать имена функций-членов с прописной буквы. На данном этапе мы успели только объявить функцию-член Greet(). Правда, не стоит волноваться — я определяю эту функцию за пределами класса.

ПОДСКАЗКА

Вероятно, вы заметили в определении класса ключевое слово public. Пока можете не обращать на него внимания. Мы обсудим это ключевое слово в данной главе в разделе «Задание открытых и закрытых уровней доступа».

Определение функций-членов

Можно определять функции-члены за пределами определения класса. Итак, за пределами определения класса Critter я определяю функцию-член этого класса Greet(), которая выводит на экран приветствие и отображает уровень голода данного тамагочи:

```
void Critter::Greet() // определение функции-члена
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n";
}
```

Это определение очень похоже на определения функций, рассмотренные нами ранее, за одним исключением: перед именем функции стоит префикс Critter::. При определении функции-члена вне того класса, к которому она относится, необходимо квалифицировать ее именем класса и оператором разрешения области видимости, чтобы компилятор знал, что это определение относится к классу.

В данной функции-члене я посылаю m_Hunger в cout. Это означает, что функция Greet() отображает значение m_Hunger для конкретного объекта — того, через который эта функция была вызвана. В результате функция отображает уровень голода, присущий конкретному тамагочи. Чтобы получить доступ к членам-данным и функциям-членам любого объекта через функцию-член, достаточно воспользоваться именем требуемого члена.

Инстанцирование объектов

При создании объекта принято говорить, что мы *инстанцируем* его из класса. Конкретные объекты именуются *экземплярами* класса. В функции main() я инстанцирую два экземпляра класса Critter:

```
Critter crit1;
Critter crit2;
```

Итак, теперь у меня есть два объекта: Critter — crit1 и crit2.

Доступ к членам данных

Давайте поработаем с нашими тамагочи. Я присваиваю первому тамагочи уровень голода:

```
crit1.m_Hunger = 9;
```

Этот код задает значение 9 для члена данных `m_Hunger` объекта `crit1`. Точно так же, как вы обращаетесь к доступной функции-члену объекта, можно получить доступ и к его члену данных — если он доступен. Для этого применяется оператор доступа к члену.

Чтобы убедиться, что операция присваивания сработала, я вывожу на экран уровень голода тамагочи:

```
cout << "crit1's hunger level is " << crit1.m_Hunger << ".\n";
```

Этот код отображает в стандартном выводе член данных `m_Hunger`, относящийся к объекту `crit1`, на экране видим верное значение 9. Как и при присваивании значения доступному члену данных, можно получить значение доступного члена данных, воспользовавшись оператором доступа к члену:

Далее вы можете убедиться, что аналогичный процесс работает и с другим объектом `Critter`.

```
crit2.m_Hunger = 3;
cout << "crit2's hunger level is " << crit2.m_Hunger << ".\n\n";
```

На этот раз я присваиваю значение 3 члену данных `m_Hunger` объекта `crit2` и отображаю это значение.

Итак, объекты `crit1` и `crit2` являются экземплярами класса `Critter`, но существуют независимо друг от друга и каждый из них обладает собственными характерными чертами. Так, каждый из этих объектов обладает собственным членом данных `m_Hunger`, у которого есть свое значение.

Вызов функций-членов

Далее я снова проверяю, нормально ли работают мои тамагочи. Первый из них должен выдать приветствие:

```
crit1.Greet();
```

Данный код вызывает функцию-член `Greet()` объекта `crit1`. Эта функция обращается к члену данных `m_Hunger` вызывающего объекта, чтобы сформировать приветствие, которое затем будет выведено на экран. Поскольку член данных `m_Hunger` объекта `crit1` имеет значение 9, функция выводит на экран такой текст: `Hi. I'm a critter. My hunger level is 9.`

Наконец, я даю слово второму тамагочи:

```
crit2.Greet();
```

Данный код вызывает функцию-член `Greet()` объекта `crit2`. Эта функция обращается к члену данных `m_Hunger` вызывающего объекта, чтобы сформировать

приветствие, которое затем будет выведено на экран. Поскольку член данных `m_Hunger` объекта `crit2` имеет значение 3, функция выводит на экран такой текст: `Hi. I'm a critter. My hunger level is 3.`

Использование конструкторов

При инстанцировании объектов зачастую требуется выполнить инициализацию — как правило, она заключается в присваивании значений членам данных. К счастью, у класса есть специальная функция-член, называемая *конструктором*, которая автоматически вызывается всякий раз при инстанцировании объекта. Это очень удобно, поскольку конструктор можно использовать для инициализации нового объекта.

Знакомство с программой Constructor Critter

Программа Constructor Critter демонстрирует работу с конструкторами. Программа инстанцирует новый объект `Critter`, при этом автоматически вызывается конструктор этого класса. Сначала конструктор объявляет, что родился новый тамагочи. Затем конструктор присваивает переданное ему значение текущему уровню голода новорожденного тамагочи. Наконец, программа вызывает «приветственную» функцию тамагочи, которая выводит на экран присущее этому тамагочи значение голода и тем самым доказывает, что конструктор действительно инициализировал тамагочи. Результат выполнения программы показан на рис. 8.2.

Рис. 8.2. Конструктор Critter автоматически инициализирует уровень голода, присущий новому объекту (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 8, имя файла — `constructor_critter.cpp`.

```
// Программа Constructor Critter
// Демонстрирует работу с конструкторами
#include <iostream>
```

```
using namespace std;
class Critter
{
public:
    int m_Hunger;
    Critter(int hunger = 0); // прототип конструктора
    void Greet();
};
Critter::Critter(int hunger) // определение конструктора
{
    cout << "A new critter has been born!" << endl;
    m_Hunger = hunger;
}
void Critter::Greet()
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n\n";
}
int main()
{
    Critter crit(7);
    crit.Greet();
    return 0;
}
```

Объявление и определение конструктора

Я объявляю конструктор класса `Critter` с помощью следующего кода:

```
Critter(int hunger = 0); // прототип конструктора
```

Как понятно из объявления, конструктор не имеет возвращаемого типа. Он и не может его иметь — указывать возвращаемый тип для конструктора не разрешается. Кроме того, при именовании конструктора действует жесткое правило: он должен называться точно так же, как соответствующий класс.

ПОДСКАЗКА

Конструктор по умолчанию не требует аргументов. Если вы сами не определите конструктор по умолчанию, то компилятор определит минималистичный вариант такой функции, которая просто будет вызывать конструкторы по умолчанию, имеющиеся у любых членов данных в классе. Если вы напишете собственный конструктор, то компилятор не будет предоставлять вам конструкторы по умолчанию. Как правило, целесообразно иметь конструктор по умолчанию, поэтому обязательно пишите такой конструктор сами, если возникает подобная необходимость. Для этого можно просто задать аргументы по умолчанию для всех параметров, указываемых в определении конструктора.

Я определяю конструктор вне класса с помощью следующего кода:

```
Critter::Critter(int hunger) // определение конструктора
{
    cout << "A new critter has been born!" <<endl;
    m_Hunger = hunger;
}
```

Конструктор отображает сообщение о том, что родился новый тамагочи, и инициализирует член данных m_Hunger этого объекта в значении аргумента, переданном конструктору. Если конструктору не было передано никакого значения, он использует задаваемое по умолчанию значение аргумента, равное 0.

ПРИЕМ

Можно использовать инициализаторы членов для ускоренного присваивания значений членам данных в конструкторе. Чтобы написать инициализатор члена, для начала поставьте двоеточие после списка параметров конструктора. Затем напишите имя того члена данных, который вы хотите инициализировать, далее — выражение, которое хотите присвоить члену данных, заключенное в круглые скобки. Если у вас несколько инициализаторов, разделяйте их запятыми. Такой прием гораздо проще, чем может показаться на первый взгляд (не говоря о том, как он полезен). Далее приведен пример, в котором значение hunger присваивается члену m_Hunger, а boredom — члену m_Boredom:

```
Critter::Critter(int hunger = 0, int boredom = 0):
m_Hunger(hunger),
m_Boredom(boredom)
{} // тело пустого конструктора
```

Инициализаторы членов особенно полезны в тех случаях, когда вам требуется инициализировать сразу множество членов.

Автоматический вызов конструктора

Конструктор не приходится вызывать явно, однако при инициализации нового объекта он вызывается автоматически. В функции main() я задействую мой конструктор с помощью следующего кода:

```
Crittercrit(7);
```

При инстанцировании объекта crit программа автоматически вызывает его конструктор, после чего на экран выводится сообщение A new critter has been born!. Затем конструктор присваивает значение 7 члену данных m_Hunger этого объекта.

Далее нужно убедиться в том, что конструктор сработал. Поэтому, вернувшись в функцию main(), я вызываю функцию-член Greet() этого объекта — и она, как и ожидалось, выводит на экран сообщение Hi. I'm a critter. My hunger level is 7.

Установка уровней доступа к членам

Объекты, как и функции, следует считать инкапсулированными сущностями. Это означает, что, в принципе, не рекомендуется напрямую обращаться к членам данных объекта или непосредственно изменять их. Вместо этого нужно вызывать функции-члены объекта, позволяя объекту поддерживать информацию о собственных членах данных и обеспечивать их целостность. К счастью, мы можем принудительно налагать ограничения на члены данных уже на этапе определения класса. Для этого устанавливаются уровни доступа к членам.

Знакомство с программой Private Critter

Программа Private Critter демонстрирует установку уровней доступа к членам данных при объявлении класса, представляющего в программе объекты тамагочи. При этом класс ограничивает непосредственный доступ к члену данных объекта, содержащему информацию об уровне голода данного тамагочи. Класс предоставляет две функции-члена: одна из этих функций обеспечивает доступ к члену данных, а другая позволяет вносить в этот член данных изменения. Программа создает нового тамагочи, опосредованно обращается к нему и изменяет его уровень голода с помощью функций-членов данного объекта. Однако, если программа пытается задать для уровня голода тамагочи недопустимое значение, то функция-член, отвечающая за изменение значений, отловит его и не позволит заменить старое значение новым. Наконец, программа вновь использует функцию-член, предназначенную для установки уровня голода, но уже с допустимым значением. Все работает как часы. На рис. 8.3 показан результат выполнения программы.

Рис. 8.3. Пользуясь функциями-членами GetHunger() и SetHunger() объекта Critter, программа опосредованно обращается к члену данных m_Hunger этого объекта (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 8, имя файла — `private_critter.cpp`.

```
// Программа Private Critter
// Демонстрирует установку уровней доступа к членам
#include <iostream>
using namespace std;
classCritter
{
    public: // начало открытого раздела
        Critter(int hunger = 0);
        int GetHunger() const;
        void SetHunger(int hunger);
    private: // начало закрытого раздела
```

```
        int m_Hunger;
};
Critter::Critter(int hunger):
m_Hunger(hunger)
{
    cout << "A new critter has been born!" << endl;
}
int Critter::GetHunger() const
{
    return m_Hunger;
}
void Critter::SetHunger(int hunger)
{
    if (hunger < 0)
    {
        cout << "You can't set a critter's hunger to a negative number.\n\n";
    }
    else
    {
        m_Hunger = hunger;
    }
}
int main()
{
    Critter crit(5);
    //cout<<crit.m_Hunger; // недопустимо, член m_Hunger является закрытым!
    cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";
    cout << "Calling SetHunger() with -1.\n";
    crit.SetHunger(-1);
    cout << "Calling SetHunger() with 9.\n";
    crit.SetHunger(9);
    cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";
    return 0;
}
```

Задание открытых и закрытых уровней доступа

Все члены данных и функции-члены, присутствующие в классе, имеют свои уровни доступа. Уровень доступа определяет, в какой части программы вы можете обратиться к конкретному члену. До сих пор я везде задавал для членов класса открытый уровень доступа, для этого применялось ключевое слово `public`. В классе `Critter` я начинаю открытый раздел следующей строкой кода:

```
public: // начало открытого раздела
```

С помощью ключевого слова `public:` я сообщаю, что все члены данных или функции-члены, которые пойдут далее, будут открытыми (пока не будет поставлен

указатель иного уровня доступа). Таким образом, к этим членам можно будет обратиться из любой части программы. Поскольку в этом разделе я объявляю все функции-члены, это означает, что из любой части моего кода я смогу вызвать любую функцию-член через объект Critter.

Следующая строка указывает, что начинается закрытый раздел кода:

```
private: // начало закрытого раздела
```

С помощью ключевого слова private: я сообщаю, что все члены данных или функции-члены, которые пойдут далее, будут закрытыми (пока не будет поставлен указатель иного уровня доступа). Таким образом, к этим членам можно будет обратиться только из кода, относящегося к классу Critter. Поскольку в этом разделе я объявляю член данных m_Hunger, это означает, что лишь код, относящийся к классу Critter, сможет обратиться к члену данных m_Hunger этого объекта. Следовательно, будучи в функции main(), я не смогу непосредственно обратиться к члену данных m_Hunger через объект, как делал это в предыдущих программах. Итак, если не закомментировать следующую строку в функции main(), то содержащаяся в ней инструкция будет недопустимой:

```
//cout<<crit.m_Hunger; // недопустимо, член m_Hunger является закрытым!
```

Поскольку член m_Hunger является закрытым, я не могу обратиться к нему из какого-либо кода, не относящегося к классу Critter. Верно и обратное: напрямую получить доступ к этому члену данных может только код, относящийся к классу Critter.

Я показал, как делать закрытыми члены данных, но подобная операция применима и к функциям-членам. Кроме того, можно повторять модификаторы доступа. Итак, если хотите, можете сделать в коде закрытый раздел, после которого пойдет открытый, далее — снова закрытый, и все это в рамках одного класса. Наконец, следует отметить, что по умолчанию доступ к членам является закрытым. Если не указать модификатор доступа, то все объявляемые вами члены класса будут получаться закрытыми.

Определение функций доступа

Функция доступа обеспечивает опосредованный доступ к члену данных. Поскольку член данных m_Hunger является закрытым, я написал функцию доступа GetHunger() для возврата значения члена данных (ключевое слово const можете пока игнорировать):

```
int Critter::GetHunger() const
{
    return m_Hunger;
}
```

В функции main() я запускаю эту функцию-член следующей строкой кода:

```
cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";
```

В приведенном коде функция доступа `crit.GetHunger()` просто возвращает значение члена данных `m_Hunger`, относящегося к объекту `crit`, — это значение равно 5.

ПРИЕМ

Точно так же, как и обычные функции, функции-члены можно подставлять. Один из способов подстановки функции-члена — определить ее непосредственно в определении класса, где обычно мы лишь объявляем функции-члены. Если включить в класс определение функции-члена, то, разумеется, определять эту функцию вне класса не придется.

Из этого правила есть исключение: при определении функции-члена в рамках определения класса с применением ключевого слова virtual автоматической подстановки функции не происходит. Мы подробнее поговорим о виртуальных функциях в главе 10.

Возможно, у вас уже возник вопрос: зачем же делать члены данных закрытыми, если к любому члену данных все равно можно обратиться через функцию доступа? Дело в том, что неограниченный доступ к членам данных обычно не допускается. Например, рассмотрим функцию доступа `SetHunger()`, которую я определил для установки уровня голода для члена данных `m_Hunger` моего объекта:

```
void Critter::SetHunger(int hunger)
{
    if (hunger < 0)
    {
        cout << "You can't set a critter's hunger to a negative number.\n\n";
    }
    else
    {
        m_Hunger = hunger;
    }
}
```

В этой функции доступа я сначала удостоверяюсь, что значение, переданное функции-члену, больше нуля. В противном случае это значение является недопустимым и я отображаю сообщение, оставляя член данных нетронутым. Если значение больше нуля, то я вношу изменение. Таким образом, функция `SetHunger()` защищает целостность члена `m_Hunger` и гарантирует, что он не может получить в качестве значения отрицательное число. Точно так же, как показано здесь, имена большинства функций доступа в реальном программировании игр начинаются с `Get` или `Set`.

Определение константных функций-членов

Константная функция-член не может изменять член данных своего класса или вызывать неконстантную функцию-член своего класса. Зачем ограничивать возможности функции-члена таким образом? Мы возвращаемся к правилу «требуй лишь того, что действительно нужно». Если вы не собираетесь изменять какие-либо члены данных в функции-члене, то будет разумно объявить эту функцию-член как константную. Так вы защититесь от нечаянного изменения члена данных в функции-члене, а другим программистам будет проще понять ваши намерения.

ОСТОРОЖНО!

Честно говоря, я немного слукавил. Константная функция-член может изменять статический член данных. Мы подробнее поговорим о статических членах данных чуть позже в этой главе, в разделе «Объявление и инициализация статических членов данных». Кроме того, если квалифицировать член данных ключевым словом mutable, то его сможет изменить даже константная функция-член. Пока не будем останавливаться на таких тонкостях.

Чтобы объявить константную функцию-член, можно поставить в конце заголовка функции ключевое слово `const`. Именно это я делаю в следующей строке кода из класса `Critter`, где функция-член `GetHunger()` объявляется как константная:

```
int GetHunger() const;
```

Таким образом, функция `GetHunger()` не может изменить значение какого-либо нестатического члена данных, объявленного в классе `Critter`, а также не может вызвать какую-либо неконстантную функцию-член класса `Critter`. Я сделал функцию `GetHunger()` константной, так как она просто возвращает значение и ей не требуется менять каких-либо членов данных. Вообще функцию-член `Get` можно смело определять как константную.

Использование статических членов данных и функций-членов

В программе Static Critter мы объявим новый вид тамагочи. В этом классе тамагочи будет статический член данных, хранящий число уже созданных тамагочи. В классе также определяется статическая функция-член, выводящая сумму. Прежде чем программа инстанцирует какие-либо новые объекты `Critter`, она отобразит общее количество имеющихся тамагочи, напрямую обратившись к статическому члену данных, в котором содержится сумма. Далее программа инстанцирует трех новых тамагочи. После этого она отобразит общее количество имеющихся тамагочи, вызвав статическую функцию-член, обращающуюся к статическому члену данных. Результат выполнения этой программы показан на рис. 8.4.

Рис. 8.4. Программа хранит общее количество объектов Critter в статическом члене данных s_Total и обращается к этому члену данных двумя разными способами (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (www.cengageptr.com/downloads). Программа находится в каталоге к главе 8, имя файла — static_critter.cpp.

```
// Программа Static Critter
// Демонстрирует работу со статическими членами данных и функциями-членами
#include <iostream>
using namespace std;
class Critter
{
    public:
        static int s_Total; // объявление статической переменной-члена
        // Общее количество существующих объектов Critter
        Critter(int hunger = 0);
        static int GetTotal(); // прототип статической функции-члена
    private:
        int m_Hunger;
};
int Critter::s_Total = 0; // инициализация статической переменной-члена
Critter::Critter(int hunger):
m_Hunger(hunger)
{
    cout << "A critter has been born!" << endl;
    ++s_Total;
}
int Critter::GetTotal() // определение статической функции-члена
{
    return s_Total;
}
int main()
{
    cout << "The total number of critters is: ";
    cout <<Critter::s_Total << "\n\n";
    Critter crit1, crit2, crit3;
    cout << "\nThe total number of critters is: ";
    cout << Critter::GetTotal() << "\n";
    return 0;
}
```

Объявление и инициализация статических членов данных

Статический член данных — это отдельно взятый член, являющийся глобальным в своем классе. В определении класса я объявляю статический член данных `s_Total` и собираюсь хранить в нем количество объектов `Critter`, которые уже были инстанцированы:

```
staticints_Total; // объявление переменной статического члена
```

Вы можете объявлять собственные статические члены данных так, как это сделал я, начиная объявление с ключевого слова `static`. Перед именем переменной я поставил префикс `s_`, поэтому с первого взгляда понимаю, что передо мной статический член данных.

За пределами определения класса я инициализирую статический член данных в значении 0:

```
int Critter::s_Total = 0; // инициализация переменной статического члена
```

Обратите внимание: я квалифицирую имя члена данных как `Critter::`. Нужно квалифицировать статический член данных именем его класса вне определения этого класса. После того как выполнится предыдущая строка кода, с классом `Critter` будет ассоциировано всего одно значение (`0`), сохраненное в его статическом члене данных `s_Total`.

ПОДСКАЗКА

Можно объявить статическую переменную и в свободных функциях (не относящихся к конкретному классу). Статическая переменная сохраняет значение между отдельными вызовами функций.

Обращение к статическим членам данных

Вы можете обратиться к открытому статическому члену данных из любой точки вашей программы. В функции `main()` я обращаюсь к члену данных `Critter::s_Total` с помощью следующей строки кода, которая выводит значение 0 (это значение статического члена данных) и общее количество объектов `Critter`, которые были инстанцированы:

```
cout << Critter::s_Total << "\n\n";
```

ПОДСКАЗКА

Можно обратиться к статическому члену данных и через любой объект, относящийся к классу. Если предположить, что объект crit1 относится к классу Critter, то отобразить общее количество тамагочи можно с помощью следующей строки:

```
cout << crit1.s_Total << "\n\n";
```

Кроме того, я обращаюсь к этому статическому члену данных в конструкторе `Critter` с помощью следующей строки, которая увеличивает значение `s_Total` на единицу.

```
++s_Total;
```

Таким образом, всякий раз при инстанцировании нового объекта значение `s_Total` увеличивается на единицу. Обратите внимание: я не квалифицирую `s_Total` с помощью `Critter::`. Точно так же, как и при работе с нестатическими членами данных, внутри класса не требуется квалифицировать статический член данных именем этого класса.

Хотя я и сделал мой статический член данных открытым, его можно сделать и закрытым. Но в таком случае, как и при работе с любым иным членом данных, доступ к статическому члену данных будет возможен только через функцию-член класса.

Объявление и определение статических функций-членов

Статическая функция-член является глобальной для всего класса. В следующей строке кода я объявляю статическую функцию-член для класса `Critter`, вот так:

```
static int GetTotal(); // прототип статической функции-члена
```

Вы можете объявить собственную статическую функцию-член точно так же, как это сделал я, начав объявление с ключевого слова `static`. Статические функции-члены часто пишутся для работы со статическими членами данных.

Я определяю статическую функцию-член `GetTotal()`, возвращающую значение статического члена данных `s_Total`:

```
int Critter::GetTotal() // определение статической функции-члена
{
    return s_Total;
}
```

Определение статической функции-члена во многом напоминает определение нестатической функции-члена (подобные определения уже были рассмотрены). Основное различие заключается в том, что статическая функция-член не может обращаться к нестатическим членам данных. Все дело в том, что статическая функция-член является глобальной в пределах класса и не ассоциирована с каким-либо конкретным экземпляром этого класса.

Вызов статических функций-членов

Инстанцировав в функции `main()` три объекта `Critter`, я вновь вывожу на экран общее число моих тамагочи — это делается с помощью следующей строки, дающей значение 3:

```
cout << Critter::GetTotal() << "\n\n";
```

Чтобы правильно идентифицировать эту статическую функцию-член, ее нужно квалифицировать с помощью `Critter::`. Для вызова статической функции-члена извне того класса, к которому она относится, ее необходимо квалифицировать именем класса.

ПОДСКАЗКА

Можно обратиться к статической функции-члену и через любой объект, относящийся к классу. Если предположить, что объект crit1 относится к классу Critter, то отобразить общее количество тамагочи можно с помощью следующей строки:

```
cout << crit1.s_Total << "\n\n";
```

Поскольку статические функции-члены являются глобальными в пределах своего класса, можно вызвать статическую функцию-член даже в том случае, если

ни одного экземпляра ее класса в данный момент не существует. Точно так же, как и при работе с закрытыми статическими членами данных, получить доступ к закрытым статическим функциям-членам можно только через другие функции-члены того же класса.

Знакомство с игрой «Тамагочи»

В игре «Тамагочи» пользователь будет заботиться о собственном виртуальном питомце. Игрок целиком отвечает за то, чтобы тамагочи был полностью доволен жизнью, а добиться этого не так просто. Чтобы тамагочи был в хорошем настроении, его нужно кормить, с ним нужно играть. Кроме того, можно «послушать» тамагочи и выяснить, как он себя чувствует. Диапазон ощущений тамагочи может варьироваться от «счастлив» до «безутешен». Игра продемонстрирована на рис. 8.5.

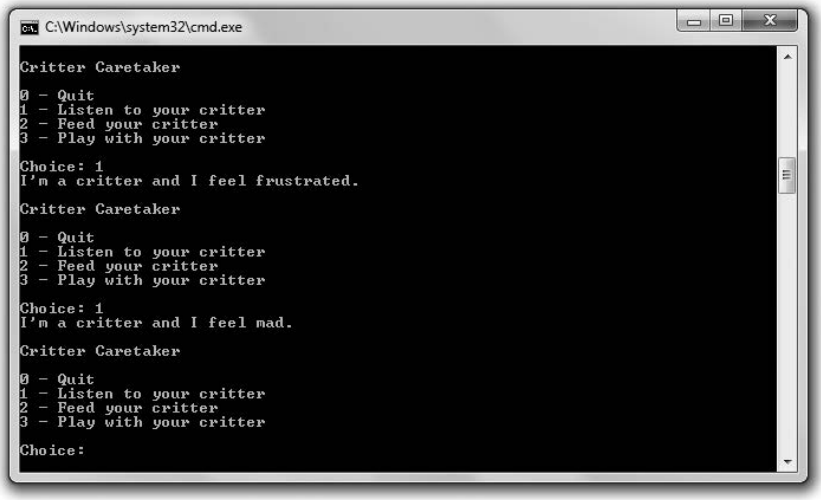


Рис. 8.5. Если вы забудете покормить или повеселить вашего тамагочи, то его настроение изменится к худшему. Но не волнуйтесь — при бережной заботе ваш тамагочи всегда будет в хорошем настроении (опубликовано с разрешения компании Microsoft)

Код этой программы можно скачать на сайте Cengage Learning (**www.cengageptr.com/downloads**). Программа находится в каталоге к главе 8, имя файла — `critter_caretaker.cpp`.

Планирование игры

Основным элементом игры является сам тамагочи. Поэтому для начала я продумываю класс `Critter`. Поскольку я хочу, чтобы у тамагочи были независимые уровни голода и скуки, сразу понятно, что в классе должны быть соответствующие закрытые члены данных:

```
m_Hunger
m_Boredom
```

Кроме того, у тамагочи должно быть и настроение, напрямую зависящее от его уровней голода и скуки. Сначала я думал, что это должен быть закрытый член данных, но на самом деле настроение тамагочи — это вычисляемое значение, основанное на значениях голода и скуки. Поэтому я решил написать закрытую функцию-член, которая на лету вычисляет настроение тамагочи, опираясь на актуальные значения скуки и голода:

```
GetMood()
```

Далее обдумываю открытые функции-члены. Я хочу, чтобы тамагочи мог сообщить пользователю о своем состоянии. Кроме того, у пользователя должна быть возможность кормить тамагочи и играть с ним, чтобы снижать уровни его голода и скуки. Для решения трех этих задач мне требуются три открытые функции-члена:

```
Talk()
Eat()
Play()
```

Наконец, мне нужна еще одна функция-член, которая будет отвечать за ход времени, чтобы некормленый и одинокий тамагочи постепенно становился все более голодным и расстроенным:

```
PassTime()
```

Полагаю, что эта функция-член должна быть закрытой, так как ее будут вызывать только другие функции-члены, например `Talk()`, `Eat()` или `Play()`.

В этом классе также будет конструктор, задача которого — инициализировать члены данных. Расмотрим рис. 8.6, на котором смоделирован класс `Critter`. Перед каждым членом данных и каждой функцией-членом я ставлю символ - или + (- означает «закрытый», а + — «открытый»).

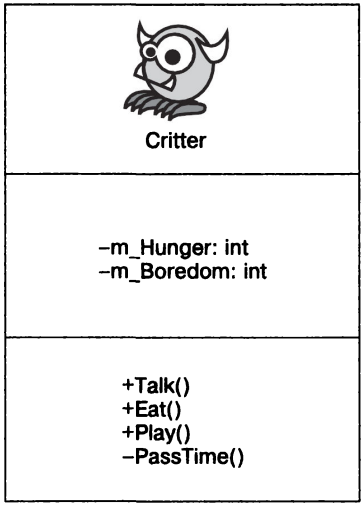


Рис. 8.6. Модель класса Critter

Планирование псевдокода

Оставшаяся часть программы будет довольно проста. В принципе, вся программа — это игровой цикл, в котором мы спрашиваем пользователя, хочет ли он выслушать тамагочи, покормить его, поиграть с ним или завершить игру. Вот какой псевдокод у меня получился:

```
Создать тамагочи
Если пользователь не собирается завершить игру
Представить ему меню с вариантами выбора
Если пользователь хочет выслушать тамагочи
Предложить тамагочи что-то сообщить
Если пользователь хочет покормить тамагочи
Предложить тамагочи поесть
Если пользователь хочет поиграть с тамагочи
Предложить тамагочи поиграть
```

Класс Critter

Класс `Critter` — это фактически чертеж объекта, представляющего пользовательского тамагочи. Сам этот класс несложен, в основном он должен показаться вам знакомым, тем не менее разберем его по частям.

Определение класса

После вводных комментариев и инструкций начинается код класса `Critter`:

```
// Программа Тамагочи
// Имитирует заботу о виртуальном питомце
#include <iostream>
using namespace std;
class Critter
{
public:
    Critter(int hunger = 0, int boredom = 0);
    void Talk();
    void Eat(int food = 4);
    void Play(int fun = 4);
private:
    int m_Hunger;
    int m_Boredom;
    int GetMood() const;
    voidPassTime(inttime = 1);
};
```

`m_Hunger` — это закрытый член данных, представляющий уровень голода тамагочи, а `m_Boredom` — закрытый член, представляющий уровень скуки нашего питомца. Каждой функции-члену я посвящу отдельный раздел.

Конструктор класса

Конструктор принимает два аргумента: hunger и boredom. По умолчанию оба этих аргумента равны 0, что я указал в прототипе конструктора еще на этапе определения класса. Я использую аргумент hunger для инициализации члена m_Hunger и boredom — для инициализации m_Boredom:

```
Critter::Critter(int hunger, int boredom):
m_Hunger(hunger),
m_Boredom(boredom)
{}
```

Функция-член GetMood()

Далее я определяю функцию GetMood():

```
inline int Critter::GetMood() const
{
    return (m_Hunger + m_Boredom);
}
```

Возвращаемое значение этой подставляемой функции-члена представляет настроение тамагочи. Настроение тамагочи определяет сумма значений голода и скуки, поэтому оно ухудшается по мере возрастания этих значений. Эту функцию-член я делаю закрытой, так как инициировать ее может только другая функция-член из того же класса. Я делаю это значение константным, поскольку оно не будет приводить к каким-либо изменениям в членах данных.

Функция-член PassTime()

Функция PassTime() — это закрытая функция-член, увеличивающая уровни голода и скуки тамагочи. Она вызывается по завершении каждой функции-члена, в которой тамагочи совершает то или иное действие (ест, играет или говорит), моделируя таким образом ход времени. Эту функцию-член я делаю закрытой, так как инициировать ее может только другая функция-член из того же класса:

```
void Critter::PassTime(int time)
{
    m_Hunger += time;
    m_Boredom += time;
}
```

Можно передать функции-члену количество истекшего времени, в противном случае переменная time получит задаваемое по умолчанию значение аргумента 1, которое я указываю в прототипе функции-члена при определении класса Critter.

Функция-член Talk()

Функция-член Talk() объявляет о настроении тамагочи, которое может принимать значения happy (счастлив), okay (доволен), frustrated (чувствует себя неуютно) и mad (возмущен). Функция Talk() вызывает функцию GetMood() и в зависимости от ее возвращаемого значения отображает на экране соответствующее сообщение

с указанием настроения тамагочи. Наконец, функция `Talk()` вызывает `PassTime()`, чтобы смоделировать ход времени:

```
void Critter::Talk()
{
    cout << "I'm a critter and I feel ";
    int mood = GetMood();
    if (mood > 15)
    {
        cout << "mad.\n";
    }
    else if (mood > 10)
    {
        cout << "frustrated.\n";
    }
    else if (mood > 5)
    {
        cout << "okay.\n";
    }
    else
    {
        cout << "happy.\n";
    }
    PassTime();
}
```

Функция-член Eat()

Функция `Eat()` уменьшает значение голода тамагочи на значение, указанное в параметре `food`. Если в этом параметре не передается никакое значение, то `food` принимает значение 4, задаваемое для этого аргумента по умолчанию. Уровень голода тамагочи постоянно находится под контролем и не может опускаться ниже 0. Наконец, вызывается функция `PassTime()`, чтобы смоделировать ход времени:

```
void Critter::Eat(int food)
{
    cout << "Brruppp.\n";
    m_Hunger -= food;
    if (m_Hunger < 0)
    {
        m_Hunger = 0;
    }
    PassTime();
}
```

Функция-член Play()

Функция `Play()` снижает уровень скуки тамагочи на величину, передаваемую в параметре `fun`. Если в этом параметре не передается никакое значение, то `fun` принимает значение 4, задаваемое для этого аргумента по умолчанию. Уровень скуки

тамагочи постоянно находится под контролем и не может опускаться ниже 0. Наконец, вызывается функция PassTime(), чтобы смоделировать ход времени:

```
void Critter::Play(int fun)
{
    cout << "Wheee!\n";
    m_Boredom -= fun;
    if (m_Boredom < 0)
    {
        m_Boredom = 0;
    }
    PassTime();
}
```

Функция main()

В функции main() я инстанцирую новый объект Critter. Поскольку я не задаю значений для его членов m_Hunger или m_Boredom, их значения начинаются с 0 — тамагочи рождается счастливым и довольным. Далее я создаю систему меню. Если пользователь вводит 0, программа завершается. Если пользователь вводит 1, программа вызывает функцию-член Talk() данного объекта. Если пользователь вводит 2, программа вызывает функцию-член Eat() данного объекта. Если пользователь вводит 3, программа вызывает функцию-член Play() данного объекта. Если пользователь вводит какое-либо иное значение, то получает сообщение, что выбранный вариант недопустим:

```
int main()
{
    Critter crit;
    crit.Talk();
    int choice;
    do
    {
        cout << "\nCritter Caretaker\n\n";
        cout << "0 - Quit\n";
        cout << "1 - Listen to your critter\n";
        cout << "2 - Feed your critter\n";
        cout << "3 - Play with your critter\n\n";
        cout << "Choice: ";
        cin >> choice;
        switch (choice)
        {
            case 0:
                cout << "Good-bye.\n";
                break;
            case 1:
                crit.Talk();
                break;
            case 2:
```

```
                crit.Eat();
                break;
            case 3:
                crit.Play();
                break;
            default:
                cout << "\nSorry, but " << choice << " isn't a valid choice.\n";
        }
    } while (choice != 0);
    return 0;
}
```

Резюме

В этой главе вы должны были выучить следующий материал.

- Объектно-ориентированное программирование — это философия программирования, предусматривающая определение различных типов объектов, между которыми существуют определенные взаимосвязи, — эти объекты взаимодействуют друг с другом.
- Можно создать новый тип, определив класс.
- Класс — это фактически чертеж объекта.
- В классе можно определять члены данных и функции-члены.
- При определении функции-члена вне определения класса требуется квалифицировать эту функцию именем класса и оператором разрешения области видимости (`::`).
- Можно подставлять функцию-член, определяя ее непосредственно в определении класса.
- Можно обращаться к членам данных и функциям-членам объекта с помощью оператора доступа к члену (`.`).
- У каждого класса есть конструктор. Это особая функция-член, которая автоматически вызывается всякий раз при инстанцировании нового объекта. Конструкторы часто применяются для инициализации членов данных.
- Конструктор, применяемый по умолчанию, не требует аргументов. Если в классе отсутствует определение конструктора, то компилятор сам создаст для вас конструктор по умолчанию.
- Инициализаторы членов обеспечивают ускоренную процедуру присваивания значений членам данных в конструкторе.
- В классе можно устанавливать уровни доступа к членам, пользуясь ключевыми словами `public`, `private` и `protected`. (О модификаторе доступа `protected` мы поговорим в главе 9.)
- К открытому члену можно обратиться через объект из любой части вашего кода.

- К закрытому члену можно обратиться только с помощью функции-члена, относящейся к тому же классу, что и этот член.
- Функция доступа обеспечивает опосредованный доступ к члену данных.
- Статический член данных является глобальным в рамках целого класса.
- Статическая функция-член является глобальной в рамках целого класса.
- Некоторые программисты-игровики ставят перед именами закрытых членов данных префикс `m_`, а перед именами статических членов данных — префикс `s_`, чтобы такие члены было легко распознавать в коде.
- Константная функция-член не может изменять нестатические члены данных или вызывать неконстантные функции-члены своего класса.

Вопросы и ответы

1. *Что такое процедурное программирование?*

   Это парадигма, в рамках которой задача подразделяется на серию более мелких задач, реализуемых в виде сравнительно удобоваримых фрагментов кода, например функций. Функции и данные в процедурном программировании являются отдельными друг от друга.

2. *Что такое объект?*

   Сущность, комбинирующая в себе данные и функции.

3. *Зачем создавать объекты?*

   Поскольку мир, а тем более игровые миры, полон объектов, то, создавая собственные типы, мы можем представлять свои объекты и их взаимосвязи с прочими объектами более непосредственным и логичным образом, чем это получалось бы иными способами.

4. *Что такое объектно-ориентированное программирование?*

   Это парадигма, в рамках которой задачи решаются с помощью объектов. В ее рамках программисты могут определять собственные типы объектов. Объекты обычно имеют взаимосвязи и могут взаимодействовать друг с другом.

5. *C++ — это язык для объектно-ориентированного или для процедурного программирования?*

   C++ — это мультипарадигмальный язык программирования. Он позволяет писать игры как в процедурном, так и в объектно-ориентированном стиле либо комбинируя оба стиля (как вариант).

6. *Всегда ли следует пытаться писать игры в объектно-ориентированном стиле?*

   Хотя объектно-ориентированное программирование применяется практически во всех играх, имеющихся сегодня на рынке, вы не обязаны писать игры с помощью именно этой парадигмы. В языке C++ можно работать с несколькими парадигмами программирования. Однако объектно-ориентированный подход будет полезен практически в любом игровом проекте.

7. *Почему бы не сделать все члены классов открытыми?*

   Потому что это противоречит идее инкапсуляции.

8. *Что такое инкапсуляция?*

   Это самодостаточность. В мире ООП инкапсуляция не позволяет клиентскому коду напрямую обращаться к внутренней структуре объекта. Напротив, благодаря инкапсуляции такой код использует для доступа к объекту строго определенный интерфейс.

9. *Каковы достоинства инкапсуляции?*

   В мире ООП инкапсуляция обеспечивает целостность объекта. Например, у вас может быть объект «звездолет», имеющий член данных «топливо». Предотвращая непосредственный доступ к этому члену данных, вы гарантируете, что он определенно не примет недопустимого значения (например, не станет отрицательным числом).

10. *Следует ли предоставлять доступ к членам данных через функции доступа?*

    Некоторые специалисты по программированию игр сказали бы, что делать этого нельзя ни в коем случае, поскольку такой доступ, хотя и является опосредованным, противоречит идее инкапсуляции. Вместо этого, считают такие специалисты, нужно писать классы с функциями-членами, предоставляющими клиенту любые возможности, которые могут ему понадобиться. В таком случае клиенту не требуется обращаться к конкретному члену данных.

11. *Что такое изменяемые члены данных?*

    Это члены данных, которые можно изменять даже с помощью константных функций-членов. Изменяемый член данных создается с помощью ключевого слова `mutable`. Кроме того, можно модифицировать изменяемые члены данных константного объекта.

12. *Почему полезно иметь конструктор по умолчанию?*

    Потому что в некоторых случаях объекты могут создаваться автоматически без передачи каких-либо значений аргументов конструктору, например при создании массива объектов.

13. *Что такое структура?*

    Структура очень напоминает класс. Единственное отличие структуры от класса заключается в том, что по умолчанию все структуры открыты для доступа. Структура определяется с помощью ключевого слова `struct`.

14. *Почему в языке C++ есть как структуры, так и классы?*

    Ради обеспечения обратной совместимости с языком C.

15. *В каких случаях следует использовать структуры?*

    Некоторые программисты применяют структуры, если требуется сгруппировать только члены данных (без функций-членов) — именно так действуют структуры в языке C. Однако лучше стараться обходиться без структур и использовать классы, когда это только возможно.

Вопросы для обсуждения

1. Каковы достоинства и недостатки процедурного программирования?
2. Каковы достоинства и недостатки объектно-ориентированного программирования?
3. Можно ли считать наличие функций доступа признаком того, что класс спроектирован плохо? Обоснуйте ответ.
4. Чем программисту-игровику полезны константные функции-члены?
5. Почему целесообразно вычислять атрибуты объекта на лету, а не хранить объект как член данных?

Упражнения

1. Усовершенствуйте программу «Тамагочи» таким образом, чтобы в ней можно было указывать отсутствующий в меню вариант, выдающий точные значения голода и скуки, присущие тамагочи в настоящий момент.
2. Измените программу «Тамагочи» таким образом, чтобы любимец мог точнее выражать свои ощущения, указывая, насколько он голоден или как сильно загрустил.
3. Какая ошибка была допущена при проектировании следующей задачи?

```
#include <iostream>
using namespace std;
class Critter
{
    public:
        int GetHunger() const {return m_Hunger;}
    private:
        int m_Hunger;
};
int main()
{
    Critter crit;
    cout << crit.GetHunger() << endl;
    return 0;
}
```

9 Более сложные классы и работа с динамической памятью. Игровое лобби

Язык C++ позволяет программисту-игровику в значительной мере управлять компьютером. Одна из фундаментальных способностей — прямое управление памятью. В этой главе вы узнаете о *динамической памяти* — памяти, которой вы управляете самостоятельно. Но помните о том, что с большой силой приходит и большая ответственность, поэтому вы узнаете о ловушках, подстерегающих при работе с динамической памятью, и о том, как их избегать. Вы также узнаете еще кое-что о классах. В частности, вы научитесь:

- комбинировать объекты;
- использовать дружественные функции;
- перегружать операторы;
- динамически выделять и освобождать память;
- избегать утечек памяти;
- создавать глубокие копии объектов.

Использование агрегирования

Игровые объекты часто создаются из других объектов. Например, в гонках машина может представлять собой объект, созданный из других объектов — корпуса, четырех шин и двигателя. Также вы можете встретить объект, представляющий собой коллекцию связанных объектов. В симуляторе зоопарка сам зоопарк можно представить как коллекцию, состоящую из произвольного количества животных. Вы можете имитировать подобные отношения среди объектов ООП с помощью *агрегирования* — объединения двух объектов таким образом, что один из них будет являться частью второго. Например, вы можете написать класс `Drag_racer`, который имеет член данных `engine`, являющийся объектом класса `Engine`. Или же можете написать класс `Zoo`, который имеет член данных `animals`, представляющий собой коллекцию объектов `Animal`.

Знакомство с программой Critter Farm

В программе «Ферма тамагочи» определяется новый тип тамагочи, имеющий имя. После того как программа объявляет новое имя тамагочи, создается ферма тамагочи — коллекция зверьков. Наконец, программа проводит перекличку на ферме, во время которой каждый тамагочи произносит свое имя. На рис. 9.1 показан результат работы программы.

Рис. 9.1. Ферма тамагочи — это коллекция зверьков, каждый из которых имеет имя (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (www.cengageptr.com/downloads). Эта программа находится в каталоге к главе 9, файл называется critter_farm.cpp.

```
// Critter Farm
// Демонстрация вложения объектов
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Critter
{
    public:
        Critter(const string& name = "");
        string GetName() const;
    private:
        string m_Name;
};

Critter::Critter(const string& name):
m_Name(name)
```

```
{}

inline string Critter::GetName() const
{
    return m_Name;
}

class Farm
{
    public:
        Farm(int spaces = 1);
        void Add(const Critter& aCritter);
        void RollCall() const;
    private:
        vector<Critter> m_Critters;
};

Farm::Farm(int spaces)
{
    m_Critters.reserve(spaces);
}

void Farm::Add(const Critter& aCritter)
{
    m_Critters.push_back(aCritter);
}
void Farm::RollCall() const
{
    for(vector<Critter>::const_iterator iter=m_Critters.begin();
    iter != m_Critters.end(); ++iter)
    {
        cout << iter->GetName() << " here.\n";
    }
}

int main()
{
    Critter crit("Poochie");
    cout << "My critter's name is " << crit.GetName() << endl;

    cout << "\nCreating critter farm.\n";
    Farm myFarm(3);
    cout << "\nAdding three critters to the farm.\n";
    myFarm.Add(Critter("Moe"));
    myFarm.Add(Critter("Larry"));
    myFarm.Add(Critter("Curly"));
    cout << "\nCalling Roll...\n";
    myFarm.RollCall();
    return 0;
}
```

Использование членов данных, являющихся объектами

При определении класса агрегирование можно использовать для того, чтобы объявить член данных, который содержит другой объект. Я сделал это в классе Critter с помощью следующей строки, в которой объявляется член данных m_Name, хранящий объект типа string:

```
string m_Name;
```

Как правило, агрегирование используется, когда объект имеет еще один объект. В этом случае тамагочи имеет имя. Отношения такого рода называются отношениями *принадлежности*.

Я поместил объявление используемого имени тамагочи при создании нового объекта с помощью следующей строки:

```
Critter crit("Poochie");
```

в которой вызывается конструктор класса Critter:

```
Critter::Critter(const string& name):
m_Name(name)
{}
```

Путем передачи строкового литерала Poochie вызывается конструктор и для имени создается объект типа string, который конструктор присваивает переменной m_Name. Рождается новый тамагочи по имени Poochie.

Далее я отображаю имя тамагочи с помощью следующей строки:

```
cout << "My critter's name is " << crit.GetName() << endl;
```

Вызов crit.GetName() возвращает копию объекта типа string, представляющего собой имя тамагочи, которая отправляется в поток cout и отображается на экране.

Использование контейнерных членов данных

Вы также можете использовать в ваших объектах контейнеры как члены данных. Именно так я и поступил, когда объявлял класс Farm. Его единственным объявленным членом данных является вектор с именем m_Critters, который хранит объекты типа Critter:

```
vector<Critter> m_Critters;
```

Когда я создаю объект типа Farm с помощью следующей строки:

```
Farm myFarm(3);
```

он вызывает конструктор:

```
Farm::Farm(int spaces)
{
    m_Critters.reserve(spaces);
}
```

который выделяет память для трех объектов типа Critter, расположенных в векторе m_Critters объекта типа Farm.

Далее я добавляю на ферму трех тамагочи путем вызова функции-члена Add() объекта типа Farm:

```
myFarm.Add(Critter("Moe"));
myFarm.Add(Critter("Larry"));
myFarm.Add(Critter("Curly"));
```

Следующая функция-член принимает постоянную ссылку на объект типа Critter и добавляет копию объекта в вектор m_Critters:

```
void Farm::Add(const Critter& aCritter)
{
    m_Critters.push_back(aCritter);
}
```

ОСТОРОЖНО!

Функция push_back() добавляет копию объекта в вектор — это значит, что я создаю дополнительную копию каждого объекта типа Critter каждый раз, когда вызываю функцию Add(). Для программы «Ферма тамагочи» это не имеет особого значения, но, если бы я добавлял множество крупных объектов, возникла бы проблема с производительностью. Вы можете избежать этой проблемы путем использования, например, вектора указателей на объекты. Принципы работы с указателями на объекты мы рассмотрим далее в этой главе.

Наконец, я устраиваю перекличку путем вызова функции-члена RollCall() класса Farm:

```
myFarm.RollCall();
```

Эта функция проходит по вектору, вызывая функцию-член GetName() каждого объекта типа Critter и заставляя каждого тамагочи «произнести» свое имя.

Использование дружественных функций и перегрузка операторов

Дружественные функции и перегруженные операторы — два более сложных понятия, связанных с классами. Дружественные функции имеют полный доступ к любому члену класса. Перегрузка операторов позволяет вам определять новые значения встроенных операторов, которые используют объекты ваших классов. Как вы увидите далее, эти два приема можно использовать одновременно.

Знакомство с программой Friend Critter

Программа Friend Critter создает объект типа Critter. Далее она использует дружественную функцию, которая может получать значения закрытых членов данных, хранящих имя тамагочи, чтобы отобразить это имя. Наконец, программа отображает объект типа Critter, отправив его в стандартный поток ввода/вывода. Это достигается с помощью дружественной функции и перегрузки операторов. На рис. 9.2 показан результат работы программы.

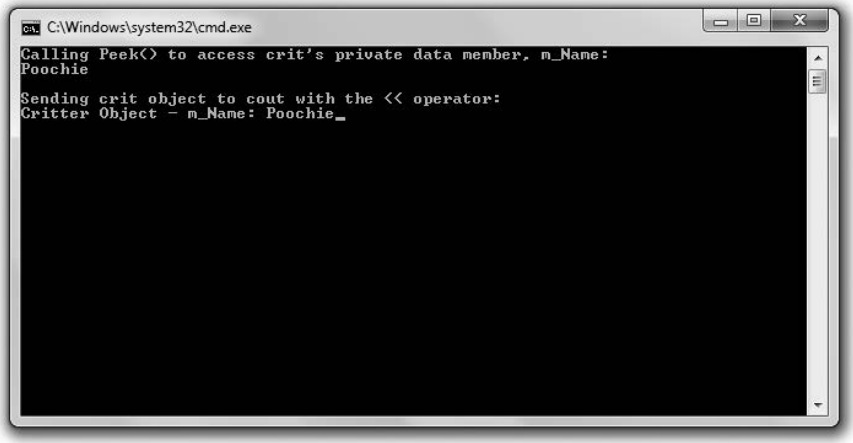


Рис. 9.2. Имя тамагочи отображается с помощью дружественной функции, объект типа Critter отображается путем его отправки в стандартный поток ввода/вывода (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 9, файл называется `friend_critter.cpp`.

```
// Friend Critter
// Демонстрируется использование дружественных функций
// и перегрузки операторов
#include <iostream>
#include <string>
using namespace std;
class Critter
{
    // Указываем, что следующие глобальные функции являются
    // дружественными по отношению к классу Critter
    friend void Peek(const Critter& aCritter);
    friend ostream& operator<<(ostream& os, const Critter& aCritter);
    public:
        Critter(const string& name = "");
    private:
        string m_Name;
};

Critter::Critter(const string& name):
m_Name(name)
{}

void Peek(const Critter& aCritter);

ostream& operator<<(ostream& os, const Critter& aCritter);

int main()
{
    Critter crit("Poochie");
    cout << "Calling Peek() to access crit's private data member, m_Name: \n";
```

```
    Peek(crit);
    cout << "\nSending crit object to cout with the << operator:\n";
    cout << crit;
    return 0;
}

// глобальная дружественная функция, которая может получать
// доступ ко всем членам объекта класса Critter
void Peek(const Critter& aCritter)
{
    cout << aCritter.m_Name << endl;
}

// глобальная дружественная функция, которая может получать
// доступ ко всем членам объекта класса Critter
// перегруженный оператор <<, позволяющий отправить объект типа Critter
// в поток cout
ostream& operator<<(ostream& os, const Critter& aCritter)
{
    os << "Critter Object - ";
    os << "m_Name: " << aCritter.m_Name;
    return os;
}
```

Создание дружественных функций

Дружественные функции могут получать доступ к любому члену класса, по отношению к которому они являются дружественными. Вы можете указать, что функция является дружественной определенному классу, разместив внутри описания класса прототип функции, перед которым стоит ключевое слово `friend`. Я сделал это внутри определения класса `Critter` с помощью следующей строки, которая указывает, что глобальная функция `Peek()` является дружественной по отношению к классу `Critter`:

```
friend void Peek(const Critter& aCritter);
```

Это значит, что функция `Peek()` может получить доступ к любому члену объекта класса `Critter`, несмотря на то что она не является членом этого класса. Функция `Peek()` использует подобные отношения, чтобы получить доступ к закрытому члену данных `m_Name` и отобразить имя тамагочи, переданное в функцию:

```
void Peek(const Critter& aCritter)
{
    cout << aCritter.m_Name << endl;
}
```

Когда я вызываю функцию `Peek()` из функции `main()` с помощью следующей строки:

```
Peek(crit);
```

отображается закрытый член данных `m_Name` объекта `crit` — на экране появляется слово `Poochie`.

Перегрузка операторов

Фраза «перегрузка операторов» звучит как нечто, чего вы хотели бы избежать любой ценой, например: «Осторожно! Этот оператор перегружен и сейчас рванет!» Но это не так. Перегрузка операторов позволяет вам указать новое значение встроенным операторам с помощью определенных вами новых типов. Например, вы можете перегрузить оператор *, чтобы при перемножении двух трехмерных матриц (объектов некоторого класса) результатом являлось произведение матриц.

Для того чтобы перегрузить оператор, определите функцию с именем `operatorX`, где `X` — это оператор, который вы хотите перегрузить. Я перегрузил оператор <<, определив функцию с именем `operator<<`:

```
ostream& operator<<(ostream& os, const Critter& aCritter)
{
    os << "Critter Object - ";
    os << "m_Name: " << aCritter.m_Name;
    return os;
}
```

Функция перегружает оператор << так, что, когда я с его помощью отправляю в поток `cout` объект типа `Critter`, на экран выводится член данных `m_Name`. Эта функция позволяет мне с легкостью отображать объекты типа `Critter`. Функция может получать прямой доступ к приватному члену данных `m_Name` объекта типа `Critter`, поскольку я указал, что эта функция является дружественной по отношению к классу `Critter`, с помощью следующей строки, расположенной в определении класса `Critter`:

```
friend ostream& operator<< (ostream& os, const Critter& aCritter);
```

Эта строка указывает, что я могу просто отобразить объект типа `Critter`, отправив его в поток `cout` с помощью оператора <<, что я делаю в функции `main()` с помощью следующей строки:

```
cout << crit;
```

которая выводит на экран текст `Critter Object m_Name: Poochie`.

ПОДСКАЗКА

Благодаря большому количеству инструментов отладки, которые доступны программистам-игровикам, иногда простое отображение значений переменных — это лучший способ понять, что происходит в ваших программах. Перегрузка оператора << может вам в этом помочь.

Эта функция сработает благодаря тому, что поток `cout` имеет тип `ostream`, в котором оператор << уже перегружен, что позволяет вам отправлять в поток `cout` встроенные типы.

Динамическое выделение памяти

До сих пор при объявлении переменной C++ выделял необходимый для нее объем памяти. Когда функция, в которой была создана эта переменная, завершается, C++ освобождает память. Память, используемая для локальных переменных, называ-

ется *стеком*. Однако существует еще один тип памяти, который не зависит от функций программы. Вы, программист, отвечаете за выделение и освобождение этой памяти, которая называется *кучей* (или *свободным хранилищем*).

Сейчас вы, возможно, задаетесь вопросом: «Зачем тратить время на другой тип памяти? Стек и без того отлично работает, спасибо». Использование динамической памяти кучи предоставляет значительные преимущества, которые можно описать одним словом — «эффективность». Используя кучу, вы можете использовать лишь необходимый объем памяти в любой момент времени. Если у вас есть игра, в которой на одном уровне находятся 100 врагов, вы можете выделить память для врагов в начале уровня и освободить ее в конце. Куча также позволяет вам создавать объект в теле функции и использовать его даже после того, как функция завершится (без необходимости возвращать копию объекта). Вы можете создавать экранный объект в одной функции и возвращать доступ к нему. Вы узнаете, что динамическая память — важный инструмент при написании любой значительной игры.

Знакомство с программой Heap

Программа Heap демонстрирует работу с динамической памятью. Программа динамически выделяет память в куче для целочисленной переменной, присваивает ей значение и затем отображает его. Далее программа вызывает функцию, которая динамически выделяет память в куче для другой целочисленной переменной, присваивает ей значение и возвращает указатель на нее. Программа принимает возвращенный указатель, использует его для отображения значения и затем освобождает выделенную в куче память. Также программа имеет две функции, демонстрирующие неправильное использование динамической памяти. Я не вызываю эти функции, а лишь использую для того, чтобы проиллюстрировать, чего *не нужно* делать с динамической памятью. Результат работы программы показан на рис. 9.3.

Рис. 9.3. Два целочисленных значения, хранящихся в куче (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 9, файл называется heap.cpp.

```
// Heap
// Демонстрация работы с динамической памятью
#include <iostream>
using namespace std;
int* intOnHeap(); // возвращает целочисленную переменную из кучи
void leak1(); // создает утечку памяти
void leak2(); // создает другую утечку памяти
int main()
{
    int* pHeap = new int;
    *pHeap = 10;
    cout << "*pHeap: " << *pHeap << "\n\n";
    int* pHeap2 = intOnHeap();
    cout << "*pHeap2: " << *pHeap2 << "\n\n";
    cout << "Freeing memory pointed to by pHeap.\n\n";
    delete pHeap;
    cout << "Freeing memory pointed to by pHeap2.\n\n";
    delete pHeap2;
    // избавляемся от висящих указателей
    pHeap = 0;
    pHeap2 = 0;
    return 0;
}

int* intOnHeap()
{
    int* pTemp = new int(20);
    return pTemp;
}

void leak1()
{
    int* drip1 = new int(30);
}

void leak2()
{
    int* drip2 = new int(50);
    drip2 = new int(100);
    delete drip2;
}
```

Использование оператора new

Оператор new выделяет память в куче и возвращает ее адрес. После оператора new следует указывать тип значения, для которого вы хотите зарезервировать место. Я делаю это в первой строке функции main():

```
int* pHeap = new int;
```

Часть утверждения, new int, выделяет достаточно памяти для одного целочисленного значения и возвращает адрес в куче этого фрагмента памяти. Другая часть этого утверждения, int* pHeap, объявляет локальный указатель с именем pHeap, который указывает на только что выделенный фрагмент кучи.

Используя указатель pHeap, я могу манипулировать фрагментом памяти кучи, зарезервированным для целочисленного значения. Это я делаю далее — присваиваю фрагменту памяти значение 10, а затем отображаю значение, хранящееся в куче, с помощью указателя pHeap, так же как с помощью любого другого указателя на целочисленную переменную. Единственное различие заключается в том, что pHeap указывает на фрагмент памяти в куче, а не в стеке.

ПОДСКАЗКА

Вы можете инициализировать память в куче в момент ее выделения путем размещения значения, заключенного в скобки, после типа. Сделать это проще, чем сказать. Например, следующая строка выделяет фрагмент памяти в куче для целочисленной переменной и присваивает ей значение 10:

```
int* pHeap = new int(10);
```

Далее утверждение присваивает адрес этого фрагмента памяти указателю pHeap.

Одно из наиболее значительных преимуществ использования кучи — устойчивость памяти даже после того, как завершится функция, в которой переменная была объявлена. Это значит, что вы можете создать объект в куче во время работы функции и вернуть указатель или ссылку на него. Я демонстрирую это в следующей строке:

```
int* pHeap2 = intOnHeap();
```

Это утверждение вызывает функцию intOnHeap(), которая выделяет фрагмент памяти в куче для целочисленной переменной и присваивает ей значение 20:

```
int* intOnHeap()
{
    int* pTemp = new int(20);
    return pTemp;
}
```

Далее функция возвращает указатель на этот фрагмент памяти. После этого в функции main() с помощью утверждения указателю pHeap2 присваивается адрес этого фрагмента памяти. Далее я использую указатель, чтобы отобразить значение:

```
cout << "*pHeap2: " << *pHeap2 << "\n\n";
```

ПОДСКАЗКА

До этого момента, если вы хотели вернуть значение, созданное в функции, вам требовалось возвращать его копию. Но с помощью динамической памяти вы можете создать объект в куче во время работы функции и вернуть указатель на новый объект.

Использование оператора delete

В отличие от локальных переменных, которые хранятся в стеке, память, которую вы выделили в куче, должна быть явно освобождена. Когда вы закончите работать с памятью, выделенной с помощью оператора new, следует освободить ее с помощью

оператора delete. Я делаю это, вызвав следующее утверждение, освобождающее фрагмент памяти в куче, в котором хранилось значение 10:

```
delete pHeap;
```

Этот фрагмент памяти возвращается в кучу для последующего использования. Данные, которые хранились в нем, более недоступны. Затем я освобождаю еще один фрагмент памяти, в котором хранилось значение 20:

```
delete pHeap2;
```

Этот фрагмент памяти возвращается в кучу для последующего использования, и данные, которые хранились в нем, более недоступны. Обратите внимание на то, что не имеет значения, в каком месте программы я выделил память в куче, если я освобождаю ее с помощью оператора delete.

ПРИЕМ

Поскольку вам необходимо освобождать выделенную память, когда она больше не нужна, возьмите за правило создавать оператор delete для каждого созданного оператора new. Фактически некоторые программисты помещают оператор delete сразу после оператора new везде, где это возможно.

Важно понять, что два предыдущих утверждения освобождают память в куче, но не влияют непосредственно на локальные переменные pHeap и pHeap2. Это создает потенциальную проблему, поскольку указатели pHeap и pHeap2 теперь указывают на память, которая была возвращена в кучу. Это означает, что они указывают на участок памяти, который компьютер может использовать каким-либо способом в любой момент времени. Подобные указатели называются *висящими указателями*, они представляют большую опасность. Вы никогда не должны пытаться разыменовать висящий указатель. Один из доступных вариантов — присвоить им значение 0, что я и делаю в следующих строках:

```
pHeap = 0;
pHeap2 = 0;
```

Это позволяет присвоить переменным-указателям другие значения, чтобы они не указывали на тот фрагмент памяти, на который указывать не должны.

Еще один вариант борьбы с висящими указателями — присваивание им корректного адреса в памяти.

ОСТОРОЖНО!

Использование оператора delete на висящем указателе приведет к аварийному завершению программы. Убедитесь, что висящий указатель имеет значение 0, или присвойте ему другое корректное значение.

Избегаем утечек памяти

Существует одна проблема, возникающая при работе с динамической памятью, — программист может выделить память и потерять способ обратиться к ней, соответственно теряя возможность ее очистить. Такая потеря называется *утечкой памяти*.

Если возникнет крупная утечка, у программы может закончиться память и она аварийно завершит работу. Как программист-игровик, вы несете ответственность за то, чтобы в программе не было утечек памяти.

Я написал две функции в программе Heap, которые намеренно создают утечки памяти. Это было сделано для того, чтобы показать вам, чего *не нужно* делать при работе с динамической памятью. Первая функция называется `leak1()`, в ней просто выделяется фрагмент памяти в куче для целочисленного значения, после чего функция заканчивается:

```
void leak1()
{
    int* drip1 = new int(30);
}
```

Если я вызову эту функцию, память будет потеряна навсегда. (Ладно, не навсегда, а только пока программа не завершится.) Проблема заключается в том, что `drip1` — это локальная переменная, которая является единственным способом связи с только что полученным фрагментом памяти из кучи и которая перестанет существовать по завершении функции `leak1()`. Соответственно, способа освободить выделенную память после завершения функции не существует. Взгляните на рис. 9.4, чтобы получить графическое представление об утечках памяти.

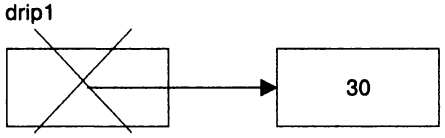


Рис. 9.4. К памяти, в которой хранится значение 30, больше нельзя обратиться для того, чтобы ее очистить, — она утекла из системы

Избежать возникновения этой утечки памяти можно двумя путями. Я мог бы использовать ключевое слово `delete` для того, чтобы освободить память в функции `leak1()`, или мог бы вернуть копию указателя `drip1`. Если бы я выбрал второй вариант, то должен был бы освободить эту память в каком-то другом месте программы.

Вторая функция, вызывающая утечку памяти, называется `leak2()`:

```
void leak2()
{
    int* drip2 = new int(50);
    drip2 = new int(100);
    delete drip2;
}
```

Эта утечка памяти чуть менее заметна, но она существует. В первой строке функции, `int* drip2 = new int(50);`, выделяется новый фрагмент памяти в куче, которому присваивается значение `50`, и `drip2` начинает указывать на этот фрагмент. До этих пор все хорошо. Во второй строке, `drip2 = new int(100);`, `drip2` начинает указывать на новый фрагмент памяти в куче, в котором хранится значение `100`. Проблема заключается в том, что на фрагмент памяти, в котором хранится

значение 50, больше ничто не указывает, поэтому способа освободить эту память не существует. Как результат этот фрагмент памяти естественным образом утекает из системы. Обратите внимание на рис. 9.5, где продемонстрировано визуальное представление того, как происходит подобная утечка.

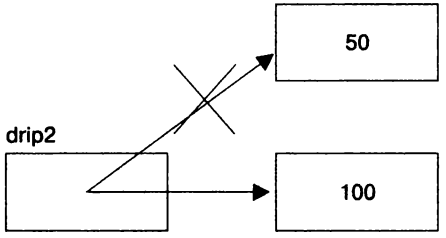


Рис. 9.5. После изменения значения указателя drip2 таким образом, чтобы он указывал на фрагмент памяти, в котором хранится значение 100, к участку памяти, где хранится значение 50, больше нельзя получить доступ и он утекает из системы

Последнее утверждение функции, `delete drip2;`, освобождает память, в которой хранится значение 100, что позволяет избежать еще одной утечки памяти. Но помните, фрагмент памяти кучи, в котором хранится значение 50, все еще считается утекшим из системы. Также я не волнуюсь насчет переменной `drip2`, которая технически становится висящим указателем, поскольку она перестанет существовать по завершении функции.

Работа с членами данных и кучей

Вы увидели, как можно использовать агрегирование для того, чтобы объявить члены данных, которые хранят объекты, но вы также можете объявить члены данных, которые являются указателями на значения в куче. Вы можете использовать член данных, который указывает на значение в куче, для того же, что и обычные указатели. Например, вы можете объявить член данных для объемной трехмерной сцены, но доступ к этой сцене можно получить только через указатель. К сожалению, когда вы используете член данных, который указывает на значение в куче, могут возникнуть проблемы из-за поведения объекта по умолчанию. Но вы можете избежать этих проблем, написав соответствующие функции-члены, позволяющие изменить поведение объекта по умолчанию.

Знакомство с программой Heap Data Member

В программе Heap Data Member определяется новый тип тамагочи, имеющий в качестве одного из членов данных указатель, который указывает на объект, хранящийся в куче. В классе определены несколько новых функций-членов, предназначенных для ситуаций, когда объект уничтожается, копируется или присваивается другому объекту. Программа уничтожает, копирует и присваивает объекты для того, чтобы показать, что они ведут себя в соответствии с вашими ожиданиями,

даже если их члены данных указывают на значения в куче. На рис. 9.6 показан результат работы программы Heap Data Member.

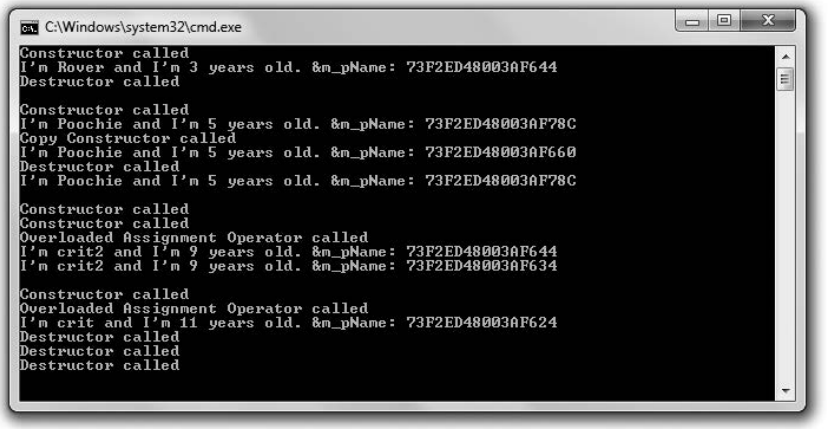


Рис. 9.6. Объекты, содержащие член данных, который указывает на значение в куче, создаются, уничтожаются и копируются (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 9, файл называется `heap_data_member.cpp`.

```
// Heap Data Member
// Демонстрация поведения объекта, имеющего член данных,
// память для которого выделяется динамически
#include <iostream>
#include <string>
using namespace std;
class Critter
{
public:
    Critter(const string& name = "", int age = 0);
    ~Critter(); // прототип деструктора
    Critter(const Critter& c); // прототип конструктора копирования
    Critter& Critter::operator=(const Critter& c);
    // перегруженная операция присваивания
    void Greet() const;
private:
    string* m_pName;
    int m_Age;
};
Critter::Critter(const string& name, int age)
{
    cout << "Constructor called\n";
    m_pName = new string(name);
    m_Age = age;
}
Critter::~Critter() //определение деструктора
```

```
{
    cout << "Destructor called\n";
    delete m_pName;
}
Critter::Critter(const Critter& c) // определение конструктора копирования
{
    cout << "Copy Constructor called\n";
    m_pName = new string(*(c.m_pName));
    m_Age = c.m_Age;
}
Critter& Critter::operator=(const Critter& c)
// определение перегруженной операции присваивания
{
    cout << "Overloaded Assignment Operator called\n";
    if (this != &c)
    {
        delete m_pName;
        m_pName = new string(*(c.m_pName));
        m_Age = c.m_Age;
    }
    return *this;
}
void Critter::Greet() const
{
    cout << "I'm " << *m_pName << " and I'm " << m_Age << " years old. ";
    cout << "&m_pName: " << &m_pName << endl;
}
void testDestructor();
void testCopyConstructor(Critter aCopy);
void testAssignmentOp();
int main()
{
    testDestructor();
    cout << endl;

    Critter crit("Poochie", 5);
    crit.Greet();
    testCopyConstructor(crit);
    crit.Greet();
    cout << endl;

    testAssignmentOp();
    return 0;
}

void testDestructor()
{
    Critter toDestroy("Rover", 3);
    toDestroy.Greet();
```

```
}

void testCopyConstructor(Critter aCopy)
{
    aCopy.Greet();
}

void testAssignmentOp()
{
    Critter crit1("crit1", 7);
    Critter crit2("crit2", 9);
    crit1 = crit2;
    crit1.Greet();
    crit2.Greet();
    cout << endl;

    Critter crit3("crit", 11);
    crit3 = crit3;
    crit3.Greet();
}
```

Объявление членов данных, которые указывают на значения в куче

Для того чтобы объявить член данных, который указывает на значение в куче, вам необходимо объявить член данных, который является указателем. Я делаю это в классе `Critter` с помощью следующей строки, в которой определяется переменная `m_pName` как указатель на объект типа `string`:

```
string* m_pName;
```

В конструкторе класса вы можете выделить память в куче, присвоить значение этому фрагменту и затем заставить член данных, являющийся указателем, указывать на этот участок памяти. Я делаю это в определении конструктора с помощью следующей строки:

```
m_pName = new string(name);
```

Сначала в ней выделяется память для объекта типа `string`, затем ей присваивается значение переменной `name`, а затем указателю `m_pName` присваиваивается адрес этого фрагмента памяти.

Также я объявляю член данных, который не является указателем:

```
int m_Age;
```

Этот член данных получает свое значение в конструкторе способом, который вы уже видели раньше, — с помощью простого утверждения присваивания:

```
m_Age = age;
```

Далее вы увидите, что каждый из этих членов данных обрабатывается по-разному по мере того, как уничтожаются, копируются и присваиваются объекты класса Critter.

Первый объект, имеющий член данных, указывающий на фрагмент памяти в куче, создается, когда из функции main() вызывается функция testDestructor(). Объект с именем toDestroy() имеет член данных m_pName, который указывает на объект типа string, имеющий значение Rover и расположенный в куче. На рис. 9.7 показано визуальное представление объекта типа Critter. Обратите внимание на то, что изображение абстрактно, поскольку имя тамагочи на самом деле хранится как объект типа string, а не как строковый литерал.

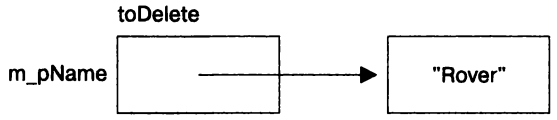


Рис. 9.7. Представление объекта класса Critter. Объект типа string, имеющий значение Rover, хранится в куче

Объявление и определение деструкторов

В случае, когда указатель является членом данных, может возникнуть утечка памяти. Это происходит потому, что при удалении объекта удаляется и указатель. Если значение, на которое он указывал, еще находится в памяти, возникает утечка. Для того чтобы избежать утечки памяти, объект должен прибрать за собой перед своим уничтожением, удалив связанное с ним значение кучи. К счастью, существует функция-член, которая называется *деструктором*. Она вызывается непосредственно перед уничтожением объекта, что можно использовать для того, чтобы выполнить необходимую уборку.

Деструктор по умолчанию, который создается компилятором в том случае, если вы не предоставите свою версию, не пытается очистить памяти кучи, на которую могут указывать члены данных. Это поведение обычно годится для простых классов, но если ваш класс имеет члены данных, которые указывают на фрагмент памяти в куче, следует написать собственный деструктор, чтобы освобождать память, связанную с объектом, перед тем как объект исчезнет. Я делаю это в классе Critter. Сначала я объявляю деструктор в описании класса. Обратите внимание на то, что деструктор имеет такое же имя, как и класс, только оно предваряется символом ~ (тильда). Деструктор не имеет никаких параметров или возвращаемого значения:

```
Critter::~Critter() // определение деструктора
{
    cout << "Destructor called\n";
    delete m_pName;
}
```

В функции main() я проверяю работу объявленного деструктора, вызвав функцию testDestructor(). Функция создает объект класса Critter с именем toDestroy

и вызывает его метод Greet(), который отображает на экране строку I'm Rover and I'm 3 years old. &m_pName: 73F2ED48003AF644. Это сообщение позволяет увидеть значение члена данных m_Age и строку, на которую указывает член данных m_pName. Также на экране отображается адрес строки в куче, который хранится в переменной m_pName. Важно обратить внимание на то, что после того, как это сообщение отображается, функция заканчивает свою работу и объект toDestroy готов к уничтожению. К счастью, деструктор этого объекта вызывается перед его уничтожением автоматически. Деструктор отображает сообщение Destructor called и удаляет объект типа string, имеющий значение Rover, из кучи, что позволяет избежать утечки памяти. Деструктор ничего не делает с членом данных m_Age. Это правильно, поскольку переменная m_Age не находится в куче. Однако она является частью объекта toDestroy и будет удалена вместе с остальной частью объекта класса Crtitter.

ПОДСКАЗКА

Если вы пишете класс, который будет выделять память в куче, напишите для него деструктор, который будет освобождать и очищать эту память.

Объявление и определение конструкторов копирования

Иногда объект копируется автоматически. Это происходит, когда объект:

- передается в функцию по значению;
- возвращается из функции;
- инициализируется с помощью другого объекта;
- предоставляется как единственный аргумент конструктора объекта.

Копирование выполняется с помощью специальной функции-члена, которая называется *конструктором копирования*. Как и в случае с конструктором и деструктором, конструктор копирования по умолчанию создается автоматически, если вы не предоставили свой. Конструктор копирования по умолчанию просто копирует значение каждого члена данных в члены данных другого объекта с таким же именем — это называется *почленным копированием*.

Для простых классов конструктор копирования по умолчанию вполне подходит. Однако, если вы пишете класс, имеющий члены данных, указывающие на значение в куче, рекомендуется создавать собственный конструктор копирования. Почему? Представьте себе объект класса Critter, имеющий член данных, который указывает на объект типа string, расположенный в куче. Конструктор копирования по умолчанию автоматически скопирует объект так, что член данных нового объекта будет указывать на ту же самую строку, поскольку в указатель на объект типа string будет банально скопирован адрес, хранящийся в указателе оригинального объекта. Такой способ копирования создает *поверхностную копию* объекта, указатели которой указывают на те же фрагменты памяти, что и указатели оригинального объекта.

Рассмотрим конкретный пример. Если бы я не написал собственный конструктор копирования в программе Heap Data Member, то в момент, когда я передал по

значению объект класса Critter с помощью следующего вызова функции, программа автоматически создала бы поверхностную копию объекта crit с именем aCopy, которая существовала бы в рамках функции testCopyConstructor():

```
testCopyConstructor(crit);
```

Член данных m_pName объекта aCopy указывал бы на тот же самый объект типа string, что и член данных m_pName объекта crit. На рис. 9.8 продемонстрировано то, что я имею в виду. Обратите внимание на то, что изображение абстрактно, поскольку имя тамагочи хранится как объект типа string, а не как строковый литерал.

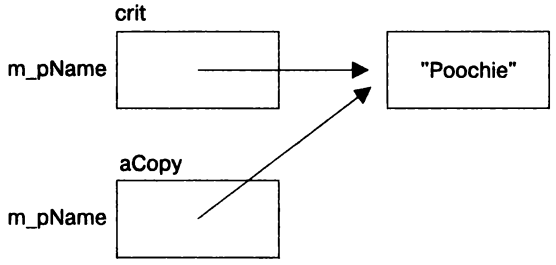


Рис. 9.8. Если бы была создана поверхностная копия объекта crit, объекты aCrit и crit имели бы члены данных, указывающие на один фрагмент памяти

Почему это может быть проблемой? Как только функция testCopyConstructor() завершит свою работу, будет вызван деструктор объекта aCopy, который освободит фрагмент памяти, на который указывает член данных m_pName. Из-за этого член данных m_pName объекта crit будет указывать на фрагмент, который уже был освобожден. Это означает, что член данных m_pName объекта crit является висящим указателем! На рис. 9.9 показано визуальное представление этого примера. Обратите внимание на то, что изображение абстрактно, поскольку имя тамагочи хранится как объект типа string, а не как строковый литерал.

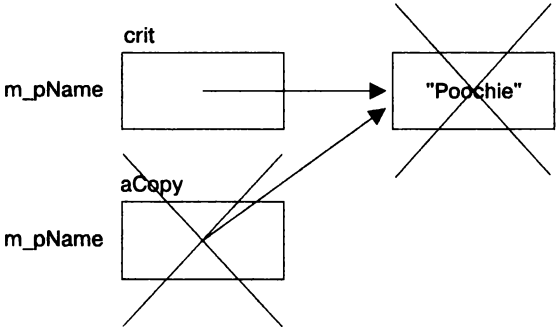


Рис. 9.9. Если поверхностная копия объекта класса Critter была уничтожена, фрагмент памяти в куче, являющийся общим для копии и оригинального объекта, будет освобожден. В результате оригинальный объект будет иметь висящий указатель

В этом случае вам необходимо создать конструктор копирования, который создает новый объект, для каждого из указателей которого выделяется собственный фрагмент памяти. Это называется *глубоким копированием*. Я делаю это при определении конструктора копирования для класса, который заменяет конструктор копирования по умолчанию, предоставляемый компилятором. Сначала я объявляю конструктор копирования внутри определения класса:

```
Critter(const Critter& c); // прототип конструктора копирования
```

Далее за пределами определения класса определяю конструктор копирования:

```
Critter::Critter(const Critter& c) // определение конструктора копирования
{
    cout << "Copy Constructor called\n";
    m_pName = new string(*(c.m_pName));
    m_Age = c.m_Age;
}
```

Конструктор копирования должен иметь то же имя, что и класс, как показано в этом примере. Он не возвращает никаких значений, но принимает как аргумент ссылку на объект класса, который должен быть скопирован. Ссылка должна быть постоянной, чтобы защитить оригинальный объект от изменения во время процесса копирования.

Задача конструктора копирования заключается в том, чтобы скопировать все члены данных из оригинального объекта в объект-копию. Если член данных оригинального объекта является указателем на значение, расположенное в куче, конструктор копирования должен запросить память из кучи, скопировать оригинальное значение в новый фрагмент, а затем заставить указывать член данных объекта-копии на этот новый фрагмент памяти.

Когда я вызываю функцию `testCopyConstructor()` путем передачи объекта `crit` в функцию по значению, конструктор копирования, который я написал, будет вызван автоматически. Это можно определить по появившемуся сообщению `Copy Constructor called`. Мой конструктор копирования создает новый объект класса `Critter` (копию) и принимает ссылку на оригинальный объект, к которому он будет обращаться по имени `c`. С помощью строки `m_pName = new string(*c.m_pName);` мой конструктор копирования выделяет новый фрагмент памяти в куче, получает копию объекта типа `string`, на который указывает оригинальный объект, копирует его в новый фрагмент памяти и заставляет член данных копии `m_pName` указывать на этот фрагмент памяти. В следующей строке, `m_Age = c.m_Age`, копируется значение оригинального члена данных `m_Age` в член данных копии с таким же именем. В результате создается глубокая копия объекта `crit`, которая используется во время работы функции `testCopyConstructor()` под именем `aCopy`.

Вы можете видеть, что конструктор копирования сработал, когда я вызвал функцию-член `Greet()` объекта `aCopy`. В моем примере эта функция-член отобразила сообщение, частью которого является строка `I'm Poochie and I'm 5 years old`. Эта часть сообщения показывает, что объект `aCopy` был скопирован из оригинального объекта `crit`. Вторая часть сообщения, `&m_pName: 73F2ED48003AF660`, показывает, что объект типа `string`, на который указывает член данных `m_pName` объекта `aCopy`,

располагается в другом фрагменте памяти, нежели строка, располагающаяся по адресу 73F2ED48003AF78C, на которую указывает член данных m_pName объекта crit. Это доказывает, что было произведено глубокое копирование. Помните, что адреса памяти, показанные в этом примере, будут отличаться от адресов, которые отобразятся при повторном запуске программы. Однако основной идеей является тот факт, что адреса, хранящиеся в членах данных m_pName объектов crit и aCopy, отличаются друг от друга.

Когда функция testCopyConstructor() заканчивает свою работу, копия объекта типа Critter, использованная в функции и хранящаяся в переменной aCopy, будет уничтожена. Деструктор освобождает фрагмент памяти кучи, связанный с копией, не затрагивая объект типа Critter, созданный в функции main(). Результат показан на рис. 9.10. Обратите внимание на то, что изображение абстрактно, поскольку имя тамагочи хранится как объект типа string, а не как строковый литерал.

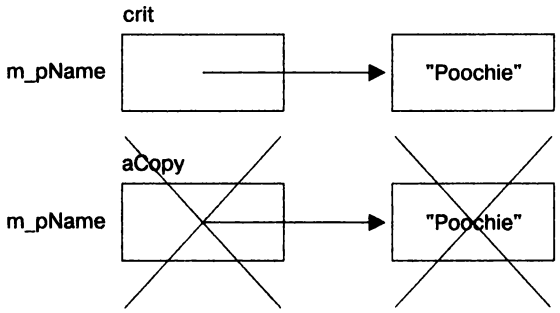


Рис. 9.10. Благодаря конструктору копирования оригинальный объект и его копия указывают на разные фрагменты памяти кучи. При уничтожении копии оригинал остается неизменным

ПОДСКАЗКА

Если вы пишете класс, члены данных которого указывают на память в куче, вам следует написать конструктор копирования, который выделяет память для нового объекта и создает глубокую копию.

Перегрузка оператора присваивания

Когда аргументы, расположенные с обеих сторон оператора присваивания, являются объектами одного и того же класса, вызывается функция-член оператора присваивания этого класса. Как и конструктор копирования по умолчанию, подобная функция-член создается автоматически, если вы не предоставите собственный вариант. Помимо этого, стандартный оператор присваивания, как и конструктор копирования по умолчанию, выполняет лишь почленное копирование.

Для простых классов подобный оператор присваивания вполне годится. Однако, когда вы пишете класс, который имеет члены данных, указывающие на значение в куче, вам следует написать собственный перегруженный оператор присваивания. Если вы не сделаете этого, при присвоении одного объекта другому будут создаваться поверхностные копии объектов. Чтобы избежать этой проблемы, я перегру-

зил оператор присваивания для класса Critter. Сначала в описании класса я написал его объявление:

```
Critter& Critter::operator=(const Critter& c);
// перегруженный оператор присваивания
```

Далее за пределами определения класса написал определение функции-члена:

```
Critter& Critter::operator=(const Critter& c)
// определение перегруженного оператора присваивания
{
    cout << "Overloaded Assignment Operator called\n";
    if (this != &c)
    {
        delete m_pName;
        m_pName = new string(*(c.m_pName));
        m_Age = c.m_Age;
    }
    return *this;
}
```

Обратите внимание на то, что функция-член возвращает ссылку на объект класса Critter. Для того чтобы оператор присваивания был более надежным, верните ссылку из перегруженной функции оператора присваивания.

В функции main() я вызываю функцию, которая тестирует перегруженный оператор присваивания для этого класса:

```
testAssignmentOp();
```

Функция testAssignmentOp() создает два объекта и присваивает один из них другому:

```
Critter crit1("crit1", 7);
Critter crit2("crit2", 9);
crit1 = crit2;
```

Утверждение присваивания crit1 = crit2; вызывает функцию-член оператора присваивания operator=() объекта crit1. В функции operator=() переменная c является постоянной ссылкой на объект crit2. Задача этой функции-члена заключается в присваивании значений всех членов данных объекта crit2 членам данных объекта crit1, при этом убеждаясь в том, что для каждого объекта класса Critter выделены свои фрагменты памяти.

После того как функция operator=() отобразит сообщение о том, что был вызван перегруженный оператор присваивания, она использует указатель this. Что такое указатель this? Это указатель, который имеют по умолчанию все нестатические функции. Он указывает на объект, который был использован для вызова функции. В этом случае this указывает на объект crit1, которому присваиваются данные.

В следующей строке, if(this != &c), выполняется проверка того, совпадает ли адрес объекта crit1 с адресом объекта crit2, то есть не присваивается ли объект сам себе. Поскольку в нашем случае это не так, начинает выполняться блок, соответствующий данной конструкции if.

Внутри блока `if` инструкция `delete m_pName;` освобождает память в куче, на которую указывал член данных `m_pName` объекта `crit1`. Строка `m_pName = new string(*(c.m_pName));` выделяет новый фрагмент памяти в куче, получает копию объекта типа `string`, на который указывает член данных `m_pName` объекта `crit2`, копирует этот объект в новый фрагмент памяти и заставляет член данных `m_pName` объекта `m_pName` указывать на этот фрагмент памяти. Вам следует пользоваться этой логикой для всех членов данных, которые указывают на память в куче.

В последней строке блока, `m_Age = c.m_Age;`, выполняется копирование значения члена данных `m_Age` объекта `crit2` в член данных `m_Age` объекта `crit1`. Вам следует выполнить подобное простое почленное копирование для всех членов данных, не являющихся указателями на память в куче.

Наконец, функция-член возвращает копию объекта `crit1`, возвращая значение `*this`. Вам следует поступить так в каждой функции-члене для перегруженных операторов.

После этого в функции `testAssignmentOp()` я убеждаюсь, что присваивание сработало правильно, вызвав методы `Greet()` для каждого из объектов. Объект `crit1` отобразит сообщение `I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF644`, а объект `crit2` — `I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF634`. Первая часть обоих сообщений, `I'm crit2 and I'm 9 years old.`, совпадает, что позволяет убедиться в том, что копирование значений сработало как полагается. Вторые части сообщений различаются, это говорит о том, что каждый объект указывает на разные фрагменты памяти в куче. Мне удалось избежать поверхностного копирования, а объекты после присваивания по-настоящему независимы.

В последнем тесте перегруженного оператора присваивания я демонстрирую, что случится, если присвоить объект самому себе. Следующие строки выполняются далее в функции:

```
Critter crit3("crit", 11);
crit3 = crit3;
```

Оператор присваивания `crit3 = crit3;` вызывает функцию-член оператора присваивания `operator=()` для объекта `crit3`. Утверждение `if` проверяет, присваивается ли объект `crit3` сам себе. Поскольку это так, функция-член возвращает ссылку на объект с помощью конструкции `return *this`. Вам следует пользоваться этой логикой при написании перегруженных операторов присваивания, поскольку присваивание объекта самому себе может вызвать потенциальные проблемы.

ПОДСКАЗКА

Если вы пишете класс, имеющий член данных, который указывает на память в куче, следует написать перегруженный оператор присваивания для класса.

Знакомство с программой Game Lobby

Программа Game Lobby симулирует игровое лобби — зону ожидания для игроков, обычно доступную в онлайн-играх. Однако эта программа работает в офлайн-режиме. Она создает очередь, в которую помещаются ожидающие игроки. Пользо-

ватель программы после запуска симуляции может выбрать один из четырех вариантов: добавить человека в лобби, удалить человека из лобби (удаляется первый человек из очереди), очистить лобби или выйти из симуляции. На рис. 9.11 показана запущенная программа Game Lobby.

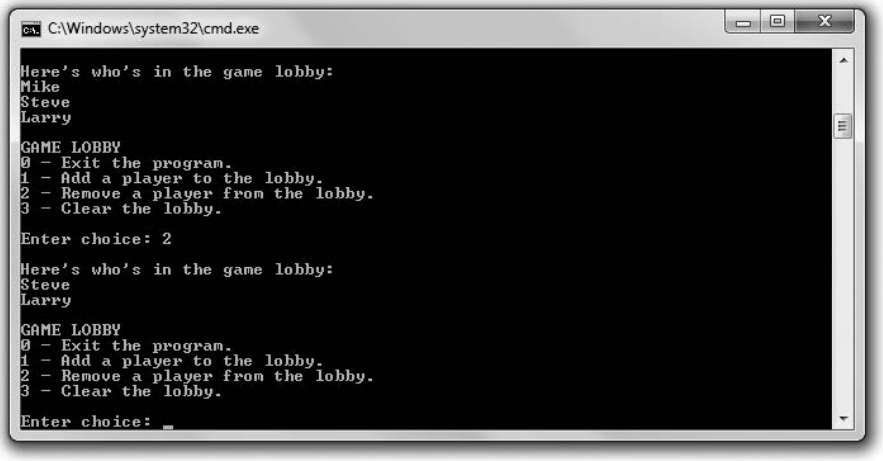


Рис. 9.11. В лобби располагаются игроки, которые удаляются из очереди в порядке их добавления (опубликовано с разрешения компании Microsoft)

Класс Player

Первое, что я делаю, — создаю класс `Player`, представляющий собой игроков, ожидающих в лобби. Поскольку я не знаю, сколько игроков будут находиться в лобби в один момент времени, имеет смысл использовать динамическую структуру данных. В обычной ситуации я бы воспользовался готовыми контейнерами библиотеки STL. Но в этой программе решил пойти другим путем и создать собственный контейнер с использованием динамически выделяемой памяти. Я поступаю так не потому, что это лучший вариант написания (всегда проверяйте, можете ли вы воспользоваться качественным кодом, написанным другими программистами, например библиотекой STL), а потому, что так можно лучше продемонстрировать работу с динамической памятью. Это действительно отличная возможность увидеть динамическую память в действии.

Вы можете загрузить код этой программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 9, файл называется `game_lobby.cpp`. Рассмотрим класс `Player`, расположенный в начале программы:

```
// Game Lobby
// Симуляция игрового лобби, предназначенного для ожидания игроков
#include <iostream>
#include <string>
using namespace std;
```

```
class Player
{
    public:
        Player(const string& name = "");
        string GetName() const;
        Player* GetNext() const;
        void SetNext(Player* next);
    private:
        string m_Name;
        Player* m_pNext; // Указатель на следующего игрока в списке
};

Player::Player(const string& name):
m_Name(name),
m_pNext(0)
{}
string Player::GetName() const
{
    return m_Name;
}
Player* Player::GetNext() const
{
    return m_pNext;
}
void Player::SetNext(Player* next)
{
    m_pNext = next;
}
```

Член данных m_Name хранит имя игрока. Это довольно очевидно, однако вам может быть интересно, за что отвечает следующий член данных, m_pNext. Этот член данных представляет собой указатель на объект класса Player, это означает, что каждый объект класса Player может хранить имя игрока и указывать на другой объект класса Player. Вы поймете смысл этого действия, когда я расскажу о классе Lobby. На рис. 9.12 приведено визуальное представление объекта Player.

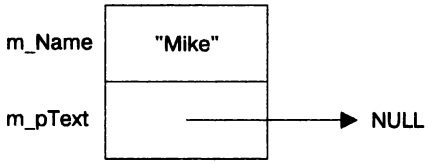


Рис. 9.12. Объект класса Player может хранить имя и указывать на другой объект класса Player

Класс имеет функцию доступа Get для члена данных m_Name, а также функции доступа Get и Set для члена данных m_pNext. Наконец, конструктор этого класса довольно прост. В нем с помощью объекта типа string, переданного в конструктор, инициализируется член данных m_Name. Также значение члена данных m_pNext устанавливается в 0, что делает этот указатель нулевым.

Класс Lobby

Класс Lobby представляет собой лобби или очередь, в которой ждут игроки. Рассмотрим определение этого класса:

```
class Lobby
{
    friend ostream& operator<<(ostream& os, const Lobby& aLobby);
public:
    Lobby();
    ~Lobby();
    void AddPlayer();
    void RemovePlayer();
    void Clear();
private:
    Player* m_pHead;
};
```

Член данных m_pHead является указателем, указывающим на объект класса Player, который представляет собой первого человека в очереди. Член данных m_pHead представляет собой начало очереди.

Поскольку каждый объект класса Player имеет член данных m_pNext, вы можете объединить несколько объектов класса Player в связанный список. Отдельные элементы связанных списков часто называются узлами. На рис. 9.13 показано визуальное представление игрового лобби — группа игроков, связанных с игроком, стоящим в начале очереди.

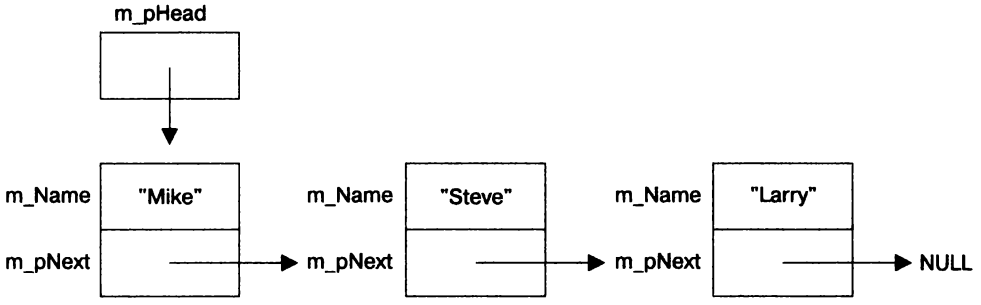


Рис. 9.13. Каждый узел хранит имя и указатель на следующего игрока в списке. Первый игрок в очереди находится в начале списка

Можно представить, что игроки — это группа связанных вагонов, везущих груз. В этом случае грузом является имя игрока, а связываются вагоны с помощью члена данных, являющегося указателем, m_pNext. Класс Lobby выделяет память в куче для каждого объекта класса Player в списке. Член данных m_pHead класса Lobby позволяет получить доступ к первому объекту класса Player, находящегося в начале очереди.

Конструктор этого класса очень прост. В нем член данных m_pHead получает значение 0, становясь нулевым указателем:

```
Lobby::Lobby():
m_pHead(0)
{}
```

Деструктор вызывает функцию Clear(), которая удаляет все объекты класса Player из списка, освобождая выделенную память:

```
Lobby::~Lobby()
{
    Clear();
}
```

Функция AddPlayer() создает объект класса Player в куче и добавляет его в конец списка.

Функция RemovePlayer() удаляет первый объект класса Player из списка и освобождает выделенную память.

Я объявляю функцию operator<<() дружественной классу Lobby, чтобы можно было отправить объект класса Lobby в поток cout с помощью оператора <<.

ОСТОРОЖНО!

Класс Lobby имеет член данных m_pHead, который указывает на объект класса Player, расположенный в куче. Из-за этого я включил в класс деструктор, который освобождает всю память, занятую объектами класса Player, созданными объектом класса Lobby, что позволяет избежать возникновения утечек памяти при уничтожении объекта класса Lobby. Однако я не определил в этом классе ни конструктор копирования, ни перегруженный оператор присваивания. Для программы Game Lobby это не обязательно. Но если бы я хотел создать более надежный класс, мне пришлось бы определить эти функции.

Функция-член Lobby::AddPlayer()

Функция-член Lobby::AddPlayer() добавляет игрока в конец очереди лобби:

```
void Lobby::AddPlayer()
{
    // создать нового игрока
    cout << "Please enter the name of the new player: ";
    string name;
    cin >> name;
    Player* pNewPlayer = new Player(name);
    // если список пуст, поставить в его начало нового игрока
    if (m_pHead == 0)
    {
        m_pHead = pNewPlayer;
    }
    // в противном случае найти конец списка и добавить игрока туда
    else
    {
        Player* pIter = m_pHead;
        while (pIter->GetNext() != 0)
        {
            pIter = pIter->GetNext();
        }
        pIter->SetNext(pNewPlayer);
    }
}
```

Сначала функция получает от пользователя новое имя игрока, которое она использует для создания нового объекта класса `Player` в куче. Далее она устанавливает значение указателя, являющегося членом данных, в `null`. После этого функция проверяет, является ли лобби пустым. Если член данных `m_pHead` объекта класса `Lobby` равен 0, то считается, что в очереди никого нет. Если это так, то новый объект класса `Player` становится в начало очереди и член данных `m_pHead` класса `Lobby` указывает на новый объект класса `Player`, расположенный в куче.

Если же лобби не пустое, игрок добавляется в конец очереди. Это делается путем перемещения по списку с помощью функции-члена `GetNext()` объекта `pIter` до тех пор, пока я не достигну объекта класса `Player`, чья функция `GetNext()` вернет значение `0`, говорящее о том, что я достиг последнего узла в списке. Далее функция заставляет этот узел указывать на новый объект класса `Player`, расположенный в куче, что можно сравнить с добавлением нового игрока в конец списка. Этот процесс проиллюстрирован на рис. 9.14.

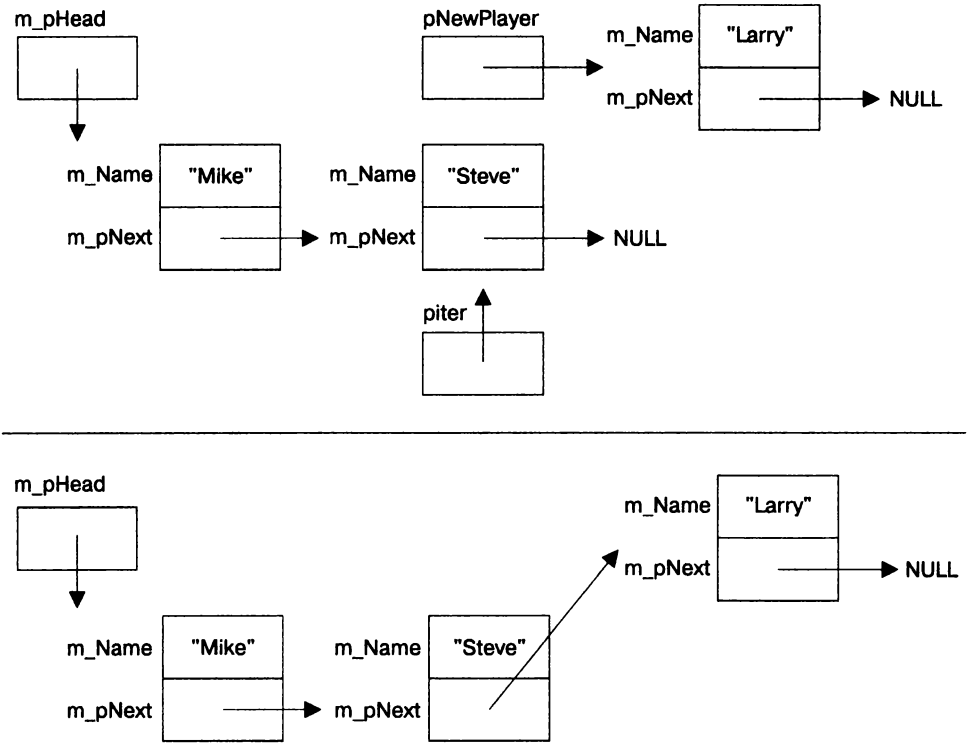


Рис. 9.14. Список игроков до и после момента добавления нового игрока

ОСТОРОЖНО!

Функция Lobby::AddPlayer() проходит по всему списку объектов класса Player при каждом ее вызове. Для маленьких списков это не проблема, но, если список разрастется, этот неэффективный процесс может затянуться. Существуют более эффективные способы решения проблемы. В одном из упражнений, приведенных в конце этой главы, вашей задачей будет реализовать один из этих более эффективных способов.

Функция-член Lobby::RemovePlayer()

Функция-член `Lobby::RemovePlayer()` удаляет игрока из начала очереди:

```
void Lobby::RemovePlayer()
{
    if (m_pHead == 0)
    {
        cout << "The game lobby is empty. No one to remove!\n";
    }
    else
    {
        Player* pTemp = m_pHead;
        m_pHead = m_pHead->GetNext();
        delete pTemp;
    }
}
```

Функция проверяет значение члена данных `m_pHead`. Если оно равно 0, то считается, что лобби пустое и функция отображает соответствующее сообщение. В противном случае удаляется первый объект из списка. Это делается путем создания указателя с именем `pTemp`, который указывает на первый объект списка. Далее функция устанавливает значение члена данных `m_pHead` равным следующему элементу списка — либо объекту класса `Player`, либо `0`. Наконец, функция уничтожает объект класса `Player`, на который указывает `pTemp`. На рис. 9.15 графически представлено то, как это работает.

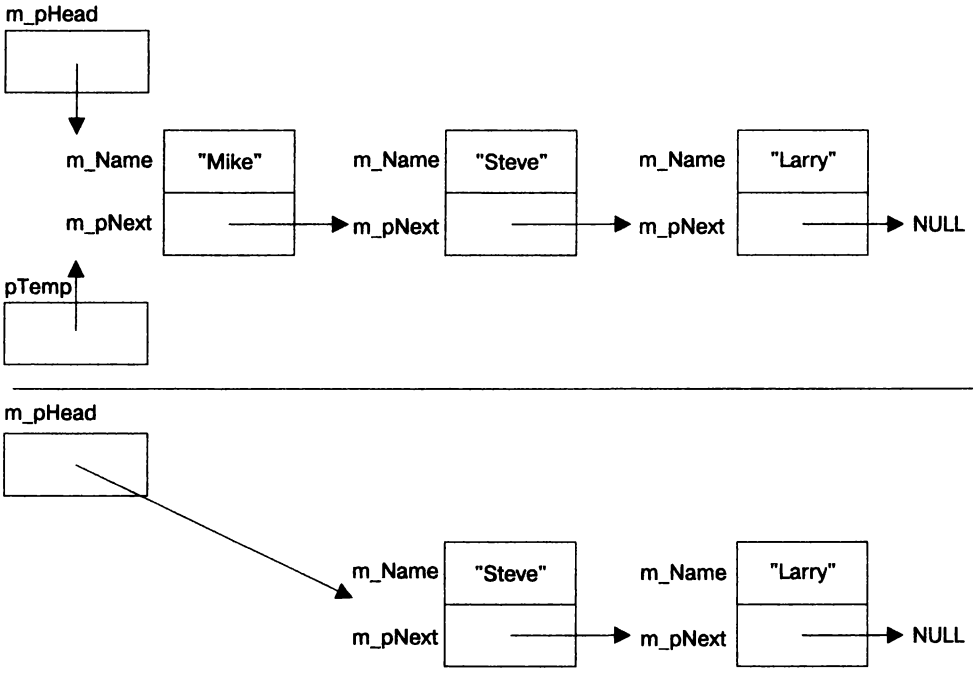


Рис. 9.15. Список игроков до и после того, как игрок был удален

Функция-член Lobby::Clear()

Функция-член `Lobby::Clear()` удаляет всех игроков из лобби:

```
void Lobby::Clear()
{
    while (m_pHead != 0)
    {
        RemovePlayer();
    }
}
```

Если список пуст, цикл не запускается и функция завершается. В противном случае запускается цикл и функция будет удалять первого игрока из списка с помощью функции `RemovePlayer()` до тех пор, пока игроки не закончатся.

Функция-член operator<<()

Функция-член `operator<<()` перегружает оператор << так, что я получаю возможность отобразить объект класса `Lobby`, отправив его в поток `cout`:

```
ostream& operator<<(ostream& os, const Lobby& aLobby)
{
    Player* pIter = aLobby.m_pHead;
    os << "\nHere's who's in the game lobby:\n";
    if (pIter == 0)
    {
        os << "The lobby is empty.\n";
    }
    else
    {
        while (pIter != 0)
        {
            os << pIter->GetName() << endl;
            pIter = pIter->GetNext();
        }
    }
    return os;
}
```

Если лобби пустое, в поток отправляется соответствующее сообщение. В противном случае функция проходит в цикле по всем игрокам списка, отправляя их имена в выходной поток. Она использует объект `pIter` для того, чтобы проходить по списку.

Функция main()

Функция `main()` отображает игроков в лобби, предоставляет пользователю меню с несколькими вариантами и выполняет соответствующее действие:

```
int main()
{
    Lobby myLobby;
```

```
    int choice;
    do
    {
        cout << myLobby;
        cout << "\nGAME LOBBY\n";
        cout << "0 - Exit the program.\n";
        cout << "1 - Add a player to the lobby.\n";
        cout << "2 - Remove a player from the lobby.\n";
        cout << "3 - Clear the lobby.\n";
        cout << endl << "Enter choice: ";
        cin >> choice;
        switch (choice)
        {
            case 0: cout << "Good-bye.\n"; break;
            case 1: myLobby.AddPlayer(); break;
            case 2: myLobby.RemovePlayer(); break;
            case 3: myLobby.Clear(); break;
           default: cout << "That was not a valid choice.\n";
        }
    }
    while (choice != 0);
    return 0;
}
```

Сначала функция создает новый объект класса `Lobby`, а затем входит в цикл, в котором на экран выводится меню и считывается выбор пользователя. Далее она вызывает соответствующую функцию-член объекта класса `Lobby`. Если пользователь выбрал некорректный вариант, ему или ей сообщается об этом. Цикл продолжается до тех пор, пока пользователь не введет значение `0`.

Резюме

Из этой главы вы должны были узнать следующее.

- Агрегация — это объединение объектов таким образом, что один является частью другого.
- Дружественные функции имеют полный доступ к любому члену класса.
- Перегрузка операторов позволяет вам определить новое значение для встроенных операторов, если они используются для объектов ваших классов.
- Стек — это область памяти, которая управляется автоматически и используется для размещения в ней локальных переменных.
- Куча (или свободное хранилище) — это область памяти, которую вы, программист, можете использовать для выделения и освобождения памяти.
- Оператор `new` выделяет память в куче и возвращает ее адрес.
- Оператор `delete` освобождает память в куче, которая была выделена ранее.
- Висящий указатель указывает на некорректную область памяти. Разыменование или освобождение висящего указателя может привести к тому, что программа аварийно завершит работу.

- Утечка памяти — это ошибка, при возникновении которой выделенная память становится недоступной и не может быть освобождена. Если произошла крупная утечка памяти, программа может исчерпать выделенный ей запас памяти и аварийно завершить работу.
- Деструктор — это функция-член, которая вызывается непосредственно перед уничтожением объекта. Если вы не напишете собственный деструктор, компилятор создаст деструктор по умолчанию.
- Конструктор копирования — это функция-член, которая вызывается при выполнении автоматического копирования объекта. Если вы не напишете собственный конструктор копирования, компилятор создаст конструктор копирования по умолчанию.
- Конструктор копирования копирует значения каждого члена данных в члены данных объекта — копии, имеющие такие же имена, выполняя почленное копирование.
- Почленное копирование может привести к созданию поверхностной копии объекта, чьи члены данных, являющиеся указателями, будут указывать на те же фрагменты памяти, что и указатели оригинального объекта.
- Глубокая копия — это копия объекта, не имеющая общих фрагментов памяти с оригинальным объектом.
- Функция-член оператора присваивания, выполняющая только почленное копирование, будет создана компилятором, если вы не напишете собственную.
- Указатель `this` — это указатель, который имеют все функции-члены, не являющиеся статическими. Он указывает на объект, который был использован для вызова функции.

Вопросы и ответы

1. *Зачем использовать агрегирование?*

   Чтобы создавать более сложные объекты на основе других объектов.

2. *Что такое композиция?*

   Это форма агрегирования, при которой создаваемый объект ответственнен за создание и уничтожение своих частей. Композиция часто называется отношениями *uses-a*.

3. *Когда следует использовать дружественные функции?*

   Когда необходимо предоставить функции доступа ко всем членам класса, не являющимся публичными.

4. *Что такое дружественная функция-член?*

   Это функция-член одного класса, которая может получить доступ ко всем членам другого класса.

5. *Что такое дружественный класс?*

   Это класс, который может получить доступ ко всем членам другого класса.

6. *Могут ли перегруженные операторы стать непонятными?*

   Да. Задание слишком большого количества разных значений или неинтуитивных значений для операторов может привести к трудностям в понимании кода.

7. *Что случится, когда я создам новый объект в куче?*

   Все члены данных разместятся в куче, а не в стеке.

8. *Могу ли я получить доступ к объекту с помощью константного указателя?*

   Конечно. Но с помощью такого указателя вы можете получить доступ только к константным функциям-членам.

9. *Что не так с поверхностными копиями?*

   Поскольку поверхностные копии имеют ссылки на одинаковые участки памяти, изменение одного объекта отразится на другом объекте.

10. *Что такое связанный список?*

    Это динамическая структура данных, которая состоит из последовательности связанных узлов.

11. *Чем связанный список отличается от вектора?*

    Связанные списки позволяют добавлять и удалять члены в любом месте списка, но не предоставляют доступа к произвольному объекту, в отличие от векторов. Однако добавление и удаление узлов в середине списка может быть более эффективным, чем добавление и удаление элементов в середине вектора.

12. *Существует ли в библиотеке STL контейнер, который работает как связанный список?*

    Да, это класс `List`.

13. *Является ли структура данных, использованная в программе Game Lobby, связанным списком?*

    Она похожа на связанный список, но по сути является очередью.

14. *Что такое очередь?*

    Это структура данных, элементы которой удаляются в порядке добавления. Этот процесс часто называется «первый вошел — первый вышел» (First in, first out, FIFO).

15. *Существует ли в библиотеке STL контейнер, который работает как очередь?*

    Да, это адаптер контейнера `Queue`.

Вопросы для обсуждения

1. Какие типы игровых сущностей можно создать с помощью агрегирования?
2. Нарушают ли дружественные функции инкапсуляцию в ООП?
3. Какие преимущества предоставляет использование динамической памяти при написании игровых программ?
4. Почему утечки памяти так трудно отследить?
5. Должны ли объекты, выделяющие память в куче, быть обязанными освобождать ее?

Упражнения

1. Улучшите класс Lobby программы Game Lobby, написав функцию, дружественную классу Player, которая позволяет отправить объект класса Player в поток cout. Далее обновите функцию, которая позволяет отправить объект класса Lobby в поток cout, используя при этом функцию отправки в поток cout объекта класса Player, которую вы только что написали.
2. Функция-член Lobby::AddPlayer() из программы Game Lobby неэффективна, поскольку она проходит по всем игрокам, чтобы добавить нового игрока в конец очереди. Добавьте новый член данных program m_pTail, который является указателем на последнего игрока в очереди, и используйте его, чтобы повысить эффективность добавления игроков в очередь.
3. Какая ошибка была допущена в следующем коде?

```
#include <iostream>
using namespace std;
int main()
{
    int* pScore = new int;
    *pScore = 500;
    pScore = new int(1000);
    delete pScore;
    pScore = 0;
    return 0;
}
```

10 Наследование и полиморфизм. Игра Blackjack

С помощью классов вы легко можете представить игровые сущности, имеющие атрибуты и поведение. Но игровые сущности часто связаны друг с другом. Из этой главы вы узнаете о наследовании и полиморфизме, что позволит вам выразить подобные связи и упростить определение и использование классов. В частности, вы научитесь:

- наследовать один класс от другого;
- использовать унаследованные члены данных и функции-члены;
- переопределять функции-члены базовых классов;
- наследовать виртуальные функции, чтобы реализовать полиморфизм;
- объявлять чистые виртуальные функции, чтобы определить абстрактные классы.

Знакомство с наследованием

Наследование — это один из основных элементов ООП. Оно позволяет создать *производный* класс на основе уже существующего класса. Когда такой класс создается, он автоматически *наследует* (или получает) члены данных и функции-члены существующего класса. Это как если бы мы бесплатно получили готовой всю работу, которую пришлось бы выполнить для проектирования базового класса!

Наследование особенно полезно, когда вы хотите создать более специфичную версию существующего класса, поскольку, чтобы расширить новый класс, можете добавлять в него члены данных и функции-члены. Например, представьте, что у вас есть класс `Enemy`, определяющий врага в игре, который имеет функцию-член `Attack()` и член данных `m_Damage`. Вы можете из класса `Enemy` создать производный класс `Boss`, который будет описывать боссов. Это будет означать, что в класс `Boss` автоматически войдут члены `Attack()` и `m_Damage` и вам не придется писать для них никакого кода. Далее, чтобы сделать босса внушительнее, можете добавить в класс `Boss` функцию-член `SpecialAttack()` и член данных `DamageMultiplier`. Обратите внимание на рис. 10.1, на котором показаны отношения между классами `Enemy` и `Boss`.

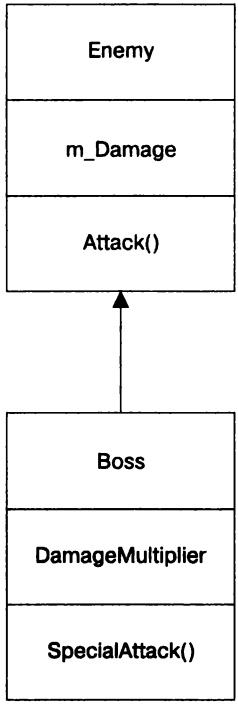


Рис. 10.1. Класс Boss наследует члены Attack() и m_Damage класса Enemy, имея при этом собственные члены SpecialAttack() и m_DamageMultiplier

Одним из многих преимуществ наследования является возможность повторно использовать уже написанные классы. Рассмотрим преимущества такой возможности.

- **Меньше работы.** Нет необходимости повторно определять функциональность, которую вы уже имеете. Когда вы напишете класс, который предоставляет базовую функциональность, вам не придется переписывать весь код еще раз.
- **Меньше ошибок.** Когда вы напишете класс, не имеющий ошибок, сможете повторно использовать его, не беспокоясь о том, что в код может вкрасться ошибка.
- **Чистый код.** Поскольку функциональность базовых классов существует только в одном месте программы, вам не придется постоянно читать один и тот же код, что позволяет упростить понимание и изменение программ.

Наиболее тесно связанные игровые сущности прямо взывают к тому, чтобы вы использовали наследование. Неважно, представляют ли эти сущности разнообразных врагов, которые встречаются персонажу, отряды армейских транспортных средств, которыми управляет игрок, или инвентарь оружия, которым пользуется персонаж, — вы можете использовать наследование, чтобы определить эти группы игровых сущностей одна с помощью другой, что упростит и ускорит выполнение ваших задач.

Знакомство с программой Simple Boss

В программе Simple Boss демонстрируется использование наследования. Я определяю класс Enemy, представляющий более слабых врагов. Из этого класса наследую новый класс Boss, представляющий сильных врагов — боссов, с которыми должен будет встретиться игрок. Далее я создаю объект класса Enemy и вызываю его функцию-класс. Boss наследует член данных m_Damage как защищенный член Attack(). После этого создаю объект класса Boss. Я могу вызвать функцию Attack() этого объекта, поскольку класс Boss наследует ее от класса Enemy. Наконец, я вызываю функцию-член SpecialAttack() класса Boss, чтобы босс произвел специальную атаку. Поскольку я определил функцию-член SpecialAttack() только в классе Boss, только объекты этого класса имеют доступ к ней. Объекты класса Enemy не имеют в своем арсенале специальных атак. На рис. 10.2 показан результат работы программы.

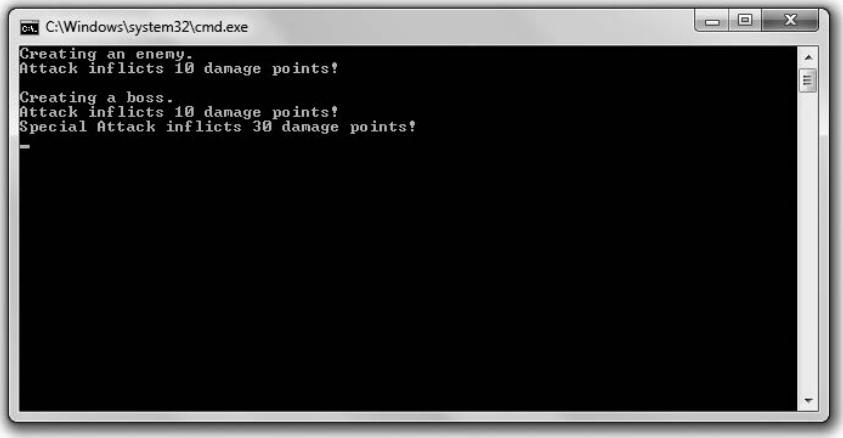


Рис. 10.2. Класс Boss наследует функцию-член Attack(), а затем определяет собственную функцию-член SpecialAttack() (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (www.cengageptr.com/downloads). Программа располагается в каталоге к главе 10, файл называется simple_boss.cpp.

```
// Simple Boss
// Демонстрация наследования
#include <iostream>
using namespace std;
class Enemy
{
    public:
        int m_Damage;
        Enemy();
        void Attack() const;
};
Enemy::Enemy():
m_Damage(10)
{}
```

```
void Enemy::Attack() const
{
    cout << "Attack inflicts " << m_Damage << " damage points!\n";
}
class Boss : public Enemy
{
public:
    int m_DamageMultiplier;
    Boss();
    void SpecialAttack() const;
};
Boss::Boss():
m_DamageMultiplier(3)
{}
void Boss::SpecialAttack() const
{
    cout << "Special Attack inflicts " << (m_DamageMultiplier * m_Damage);
    cout << " damage points!\n";
}
int main()
{
    cout << "Creating an enemy.\n";
    Enemy enemy1;
    enemy1.Attack();
    cout << "\nCreating a boss.\n";
    Boss boss1;
    boss1.Attack();
    boss1.SpecialAttack();
    return 0;
}
```

Наследование от базового класса

Я указываю, что класс `Boss` наследует от класса `Enemy`, с помощью следующей строки:

```
class Boss: public Enemy
```

Класс `Boss` основан на классе `Enemy`. Фактически класс `Enemy` называется *базовым классом* (или *суперклассом*), а класс `Boss` — *производным классом* (или *подклассом*). Это означает, что класс `Boss` наследует члены данных и функции-члены класса `Enemy` в соответствии с модификаторами доступа. В нашем примере класс `Boss` наследует члены `m_Damage` и `Attack()` и может напрямую обращаться к ним точно так же, как если бы я определил оба этих члена в классе `Boss`.

ПОДСКАЗКА

Вы могли заметить, что все члены классов, включая члены данных, являются открытыми. Я создал их такими, потому что хотел показать простейший пример базового и производного классов. Вы также могли заметить слово public, использованное при наследовании классом Boss от класса Enemy. Не обращайте сейчас на него внимания. Мы рассмотрим этот вопрос в следующем примере программы, Simple Boss 2.0.

Для того чтобы наследовать от своих классов, действуйте по моему примеру. В определении класса после его имени поставьте двоеточие, за ним будет стоять модификатор доступа (например, `public`), после которого — имя базового класса. Вполне приемлемо унаследовать новый класс от производного класса, иногда подобное действие будет наиболее логичным. Однако не слишком усложняйте код. В этом примере я работаю только с одним уровнем наследования.

Лишь некоторые функции-члены класса не наследуются производным классом. Перечислим их:

- конструкторы;
- конструкторы копирования;
- деструкторы;
- перегруженные операторы присваивания.

Вы должны написать собственные версии этих функций в производном классе.

Создание объектов производного класса

В функции `main()` я создаю объект класса `Enemy` и вызываю его функцию-член `Attack()`. Это сработает в полном соответствии с вашими ожиданиями. Интересная часть программы начинается позже, когда я создаю объект класса `Boss`:

```
Boss boss1;
```

После этой строки кода у меня появляется объект класса `Boss`, значение члена данных `m_Damage` которого равно `10`, а значение члена данных `m_DamageMultiplier` — `3`. Как это получилось? Хотя конструкторы и деструкторы нельзя унаследовать от базового класса, они все равно вызываются при создании или уничтожении объекта. Фактически конструктор базового класса вызывается перед конструктором производного класса, чтобы создать свою часть итогового объекта.

В нашем примере при создании объекта класса `Boss` автоматически вызывается конструктор по умолчанию объекта `Enemy` и член данных `m_Damage` нового объекта получает значение `10` (как и любой другой объект класса `Enemy`). Далее вызывается конструктор класса `Boss`, который завершает создание объекта, присваивая члену данных `m_DamageMultiplier` значение `3`. При уничтожении объекта класса `Boss` в конце программы происходит обратное: сначала вызывается деструктор класса `Boss`, а затем деструктор класса `Enemy`. Поскольку я не определял деструкторы в этой программе, ничего особенного перед тем, как объект класса `Boss` прекратит свое существование, не произойдет.

ПОДСКАЗКА

Тот факт, что деструкторы базовых классов вызываются при уничтожении объектов производных классов, позволяет гарантировать, что каждый класс получит возможность очистить необходимые части объекта, например память в куче.

Использование унаследованных членов

Далее для объекта класса Boss я вызываю унаследованную функцию-член, которая отображает точно такое же сообщение, как и функция enemy.Attack():

```
boss1.Attack();
```

Это абсолютно логично, поскольку выполняется один и тот же фрагмент кода и член данных m_Damage обоих объектов равен 10. Обратите внимание на то, что вызов функции выглядит так же, как и аналогичный вызов функции объекта enemy1. Факт наследования классом Boss класса Enemy позволяет объектам производного класса вызывать функции базового класса наравне с объектами производного класса.

Далее я заставляю босса произвести специальную атаку, что приводит к появлению на экране сообщения Special Attack inflicts 30 damage points!:

```
boss1.SpecialAttack();
```

Здесь следует обратить внимание на то, что функция-член SpecialAttack(), объявленная как часть класса Boss, использует член данных m_Damage, объявленный в классе Enemy. Это абсолютно приемлемо. Класс Boss наследует член данных m_Damage от класса Enemy, и в этом примере член данных работает так же, как и любой другой член данных класса Boss.

Управление доступом при работе с наследованием

Когда вы наследуете одним классом от другого, то можете управлять доступом производного класса к членам данных базового класса. Вам следует предоставлять доступ произвольного класса только к необходимым членам базового класса по тем же причинам, которые заставляют скрывать реализацию классов для остальной части программы. Для этого используются модификаторы доступа, которые вы уже видели ранее: public, protected и private. (Хорошо-хорошо, вы еще не видели модификатора protected, но я расскажу о нем в подразделе «Использование модификаторов доступа для членов данных».)

Знакомство с программой Simple Boss 2.0

Программа Simple Boss 2.0 — это новая версия программы Simple Boss, показанной ранее в этой главе. Эта версия выглядит для пользователя так же, как и оригинал, но ее код несколько отличается, поскольку я ограничил использование некоторых членов классов. Если вы хотите увидеть результат работы программы, вернитесь к рис. 10.2.

Вы можете загрузить код программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 10, файл называется `simple_boss2.cpp`.

```
// Simple Boss 2.0
// Демонстрация управления доступом при работе с наследованием
#include <iostream>
using namespace std;
class Enemy
{
    public:
        Enemy();
        void Attack() const;
    protected:
        int m_Damage;
};
Enemy::Enemy():
m_Damage(10)
{}
void Enemy::Attack() const
{
    cout << "Attack inflicts " << m_Damage << " damage points!\n";
}
class Boss : public Enemy
{
public:
    Boss();
    void SpecialAttack() const;
private:
    int m_DamageMultiplier;
};
Boss::Boss():
m_DamageMultiplier(3)
{}
void Boss::SpecialAttack() const
{
    cout << "Special Attack inflicts " << (m_DamageMultiplier * m_Damage);
    cout << " damage points!\n";
}
int main()
{
    cout << "Creating an enemy.\n";
    Enemy enemy1;
    enemy1.Attack();
    cout << "\nCreating a boss.\n";
    Boss boss1;
    boss1.Attack();
    boss1.SpecialAttack();
    return 0;
}
```

Использование модификаторов доступа для членов данных

Вы уже встречали модификаторы доступа `public` и `private`, но существует и третий модификатор, который также можно использовать для членов класса, — `protected` (защищенный). Именно его я использую для члена данных класса `Enemy`:

```
protected:
    int m_Damage;
```

Члены данных, имеющие модификатор `protected`, недоступны за пределами класса, за исключением некоторых вариантов наследования. Освежим в памяти три уровня доступа к членам класса.

- Открытые члены доступны во всем коде программы.
- Защищенные члены доступны только внутри своего класса и некоторых производных классов в зависимости от уровня доступа наследования.
- Закрытые члены доступны только внутри своего класса, это означает, что они недоступны непосредственно в производных классах.

Использование модификаторов доступа при создании производных классов

Когда вы создаете класс на основе уже существующего класса, вы можете использовать модификатор доступа, например `public`, что я сделал при создании класса `Boss`:

```
class Boss : public Enemy
```

Использование наследования с модификатором `public` означает, что открытые члены базового класса становятся открытыми членами производного класса, защищенные члены данных базового класса становятся защищенными членами данных производного класса, а закрытые члены данных базового класса недоступны производному классу.

ПРИЕМ

Даже если члены базового класса являются закрытыми, вы все еще можете использовать их опосредованно с помощью функций-членов. Вы даже можете устанавливать и считывать их значения, если в базовом классе имеются функции доступа.

Поскольку класс `Boss` наследует от класса `Enemy`, используя ключевое слово `public`, класс `Boss` наследует открытые функции-члены класса `Enemy` и может пользоваться ими как своими собственными. Это также значит, что класс `Boss` наследует член данных `m_Damage` как защищенный. Новый класс ведет себя точно так же, как если бы я просто скопировал в его определение два члена данных класса `Enemy`. Однако благодаря наследованию я не обязан так делать — класс `Boss` и без этого может использовать члены `Attack()` и `m_Damage`.

ПОДСКАЗКА

Вы можете унаследовать новый класс с помощью ключевых слов protected и private, но подобное наследование встречается редко, поэтому в рамках данной книги оно рассматриваться не будет.

Вызов и переопределение функций-членов базового класса

Вы не обязаны использовать каждую функцию базового класса, унаследованную в производном классе, без изменений. Можете изменить способ работы унаследованных функций в производном классе. Функции базового класса можно переопределить, написав для них новые определения в производном классе. Вы также можете явно вызвать функцию-член базового класса из любой функции-члена производного класса.

Знакомство с программой Overriding Boss

Программа Overriding Boss демонстрирует вызов и переопределение функций базового класса в производном классе. Программа создает врага, который насмехается над игроком, а затем атакует его. Далее программа создает босса с помощью производного класса. Босс также насмехается над игроком и атакует его, но самое интересное заключается в том, что для босса изменились механизмы атаки и насмешки (поскольку он чуть более сильный, чем обычные враги). Эти изменения произошли потому, что были переопределены и вызваны функции-члены базового класса. Результат работы программы показан на рис. 10.3.

Рис. 10.3. Класс Boss наследует и переопределяет функции-члены базового класса Taunt() и Attack(), создавая тем самым новое поведение этих функций для класса Boss (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код программы с веб-сайта Cengage Learning (www.cengageptr.com/downloads). Программа располагается в каталоге к главе 10, файл называется overriding_boss.cpp.

```
// Overriding Boss
// Демонстрация вызова и переопределения функций-членов базового класса
#include <iostream>
```

```
using namespace std;
class Enemy
{
    public:
        Enemy(int damage = 10);
    // функция создается как виртуальная для последующего переопределения
        void virtual Taunt() const;
    // функция создается как виртуальная для последующего переопределения
        void virtual Attack() const;
    private:
        int m_Damage;
};
Enemy::Enemy(int damage):
m_Damage(damage)
{}
void Enemy::Taunt() const
{
    cout << "The enemy says he will fight you.\n";
}
void Enemy::Attack() const
{
    cout << "Attack! Inflicts " << m_Damage << " damage points.";
}
class Boss : public Enemy
{
    public:
        Boss(int damage = 30);
        // использование ключевого слова virtual опционально
        void virtual Taunt() const;
        // использование ключевого слова virtual опционально
        void virtual Attack() const;
};
Boss::Boss(int damage):
Enemy(damage) // вызов конструктора базового класса с аргументом
{}

void Boss::Taunt() const // переопределение функции-члена базового класса
{
    cout << "The boss says he will end your pitiful existence.\n";
}
void Boss::Attack() const // переопределение функции-члена базового класса
{
    Enemy::Attack(); // вызов функции-члена базового класса
    cout << " And laughs heartily at you.\n";
}
int main()
{
    cout << "Enemy object:\n";
```

```
    Enemy anEnemy;
    anEnemy.Taunt();
    anEnemy.Attack();
    cout << "\n\nBoss object:\n";
    Boss aBoss;
    aBoss.Taunt();
    aBoss.Attack();
    return 0;
}
```

Вызов конструкторов базового класса

Как вы уже видели, конструктор базового класса вызывается автоматически, когда создается объект производного класса, но также можно вызвать конструктор базового класса из конструктора производного класса. Синтаксис такого вызова будет выглядеть почти как синтаксис инициализации списка членов. Чтобы вызвать конструктор базового класса из конструктора производного класса, необходимо после списка параметров конструктора производного класса поставить двоеточие, после которого будет идти имя базового класса, за которым — список параметров для конструктора этого класса, размещенный в круглых скобках. Я делаю это в конструкторе класса `Boss`, где явно указывается вызвать конструктор класса `Enemy`, которому передается параметр `damage`:

```
Boss::Boss(int damage):
Enemy(damage) // вызов конструктора базового класса с аргументом
{}
```

Это позволяет мне передать в конструктор класса `Enemy` значение, которое будет присвоено параметру `m_Damage`, вместо того чтобы присваивать ему значение по умолчанию.

Когда я в первый раз создаю объект `aBoss` в функции `main()`, вызывается конструктор класса `Enemy`, в который передается значение `30`, которое присваивается члену данных `m_Damage`. Далее запускается конструктор класса `Boss` (он не делает ничего особенного), после этого создание объекта завершается.

ПОДСКАЗКА

Возможность вызвать конструктор базового класса может пригодиться, когда вам нужно передать туда определенные значения.

Объявление виртуальных функций-членов базового класса

Любая наследуемая функция-член базового класса, которую могут переопределить, должна быть объявлена как виртуальная с помощью ключевого слова `virtual`. Когда вы указываете, что функция-член должна быть виртуальной, вы даете шанс переопределенным функциям работать с указателями и ссылками в соответствии

с вашими ожиданиями. Поскольку я знаю, что буду переопределять функцию Taunt() в производном классе Boss, то объявлю ее виртуальной в классе Enemy:

```
// функция создана виртуальной, чтобы ее можно было переопределить
void virtual Taunt() const;
```

ОСТОРОЖНО!

Несмотря на то что вы можете переопределить и невиртуальные функции-члены, это может привести к тому, что функции будут вести себя непредсказуемо. Рекомендуется использовать ключевое слово virtual по отношению ко всем функциям, которые будут переопределены.

За пределами класса Enemy я определяю функцию Taunt():

```
void Enemy::Taunt() const
{
    cout << "The enemy says he will fight you.\n";
}
```

Обратите внимание на то, что я не использовал ключевое слово virtual в определении функции. Это ключевое слово используется только при объявлении функции.

После того как функция-член объявлена как виртуальная, она будет являться таковой во всех производных классах. Это значит, что использовать ключевое слово virtual в объявлении не обязательно, когда вы будете переопределять виртуальную функцию-член, но вам все равно следует это делать, чтобы не забывать, что эта функция является виртуальной. Поэтому при переопределении функции Taunt() в классе Boss я явно объявляю ее виртуальной, несмотря на то что не обязан это делать:

```
// использование ключевого слова virtual опционально
void virtual Taunt() const;
```

Переопределение виртуальных функций-членов базового класса

Следующим этапом переопределения является указание нового определения функции в производном классе. Я делаю это в классе Boss с помощью следующих строк:

```
void Boss::Taunt() const // переопределение функции-члена базового класса
{
    cout << "The boss says he will end your pitiful existence.\n";
}
```

Это новое определение выполняется всякий раз, когда я вызываю функцию-член для объекта класса Boss. Когда я вызываю эту функцию в функции main() с помощью следующей строки:

```
aBoss.Taunt();
```

на экране появляется сообщение The boss says he will end your pitiful existence.

Переопределение функций-членов может быть полезно в том случае, если вы хотите изменить или расширить поведение функций-членов базового класса в производном классе.

ОСТОРОЖНО!

Не путайте переопределение с перегрузкой. Когда вы переопределяете функцию, вы предоставляете для нее новое определение в производном классе. Когда вы перегружаете функцию, то создаете несколько ее версий с разными сигнатурами.

ОСТОРОЖНО!

Когда вы переопределяете перегруженную функцию базового класса, вы скрываете все остальные перегруженные версии функции-члена базового класса, что означает, что единственный способ получить доступ к другим версиям функции-члена базового класса — явно вызвать функцию-член базового класса. Поэтому, если вы переопределяете перегруженную функцию-член, рекомендуется переопределить каждую версию перегруженной функции.

Вызов функций-членов базового класса

Вы можете вызвать непосредственно функцию-член базового класса из любой функции производного класса. Все, что вам нужно сделать, — добавить имя класса перед именем функции внутри одного оператора разрешения области действия. Я делаю это при определении переопределенной версии функции `Attack()` класса `Boss`:

```
void Boss::Attack() const // переопределение функции-члена базового класса
{
    Enemy::Attack(); // вызов функции-члена базового класса
    cout << " And laughs heartily at you.\n";
}
```

С помощью конструкции `Enemy::Attack();` явно вызывается функция-член `Attack()` класса `Enemy`. Поскольку определение функции `Attack()` класса `Boss` переопределяет унаследованную версию функции, оно работает так, как если бы я расширил определение функции базового класса. Я хочу сказать, что, когда босс атакует, он делает то же самое, что и обычный враг, а затем смеется. Когда я вызываю функцию-член класса `Boss` в функции `main()` с помощью следующей строки:

```
aBoss.Attack();
```

вызывается функция-член `Attack()` класса `Boss`, поскольку я перегрузил функцию `Attack()`.

Первое, что делает функция-член `Attack()` класса `Boss`, — явно вызывает функцию-член `Attack()` класса `Enemy`, которая отображает сообщение `Attack! Inflicts 30 damage points.` Далее функция-член `Attack()` класса `Boss` отображает сообщение `And laughs heartily at you.`

ПРИЕМ

Вы можете расширить возможности функции-члена базового класса, если переопределите метод базового класса, а затем явно вызовете функцию-член из нового определения производного класса, после чего выполнится другая функциональность.

Использование перегруженных операторов присваивания и конструкторов копирования в производных классах

Вы уже знаете, как написать перегруженные операторы присваивания и конструктор копирования для класса. Однако их создание для производного класса потребует чуть больших усилий, поскольку они не наследуются от базового класса.

Когда вы перегружаете оператор присваивания в производном классе, вы обычно хотите вызвать функцию-член оператора присваивания базового класса, что можете сделать явно, использовав как префикс имя базового класса. Если класс Boss наследует от класса Enemy, перегруженная функция-член оператора присваивания может выглядеть следующим образом:

```
Boss& operator=(const Boss& b)
{
Enemy::operator=(b); // работает с членами данных, унаследованными от класса Enemy
// теперь нужно обработать члены данных, определенные в классе Boss
```

Явный вызов функции-члена оператора присваивания класса Enemy работает с членами данных, унаследованными от класса Enemy. Другая часть функции-члена работает с членами данных, определенными в классе Boss.

Что касается конструкторов копирования, вы также обычно хотите вызвать конструктор копирования базового класса, его можно вызвать, как и любой другой конструктор базового класса. Если класс Boss наследует от класса Enemy, конструктор копирования класса Boss может выглядеть следующим образом:

```
Boss (const Boss& b): Enemy(b)
// работает с членами данных, унаследованными от класса Enemy
{
// теперь нужно обработать члены данных, определенные в классе Boss
```

Вызывая конструктор копирования класса Enemy с помощью конструкции Enemy(b), вы копируете члены данных объекта Enemy в новый объект класса Boss. В другой части конструктора копирования класса Boss вы можете позаботиться о копировании членов данных, объявленных в классе Boss, в новый объект.

Знакомство с полиморфизмом

Полиморфизм является одним из столпов ООП. Благодаря полиморфизму функция-член будет возвращать разные результаты в зависимости от того, для объекта какого типа она вызывается. Например, предположим, что у вас есть группа противников, с которыми должен встретиться игрок. Эта группа состоит из объектов различных типов, связанных наследованием, например врагов и боссов. Благодаря магии полиморфизма вы можете вызывать одну и ту же функцию-член для каждого противника в группе для того, чтобы, скажем, атаковать игрока, и тип каждого объекта определит окончательный результат. Вызов функции для обычного врага приведет к одному результату, слабой атаке, а вызов функции для босса — к другому, мощной атаке. Может показаться, что это похоже на переопределение функций,

но полиморфизм от него несколько отличается, поскольку эффект вызова функции определяется динамически в зависимости от типа объекта во время работы программы. Рассмотрим конкретный пример, чтобы лучше понять этот принцип.

Знакомство с программой Polymorphic Bad Guy

В программе Polymorphic Bad Guy демонстрируется полиморфическое поведение. В ней показывается, что случится, когда вы используете указатель на базовый класс для того, чтобы вызвать унаследованные виртуальные функции-члены. В ней также показывается, как, используя виртуальные деструкторы, можно убедиться, что вызывается конкретный деструктор для объектов, на которые указывает указатель базового класса. На рис. 10.4 показан результат работы программы.

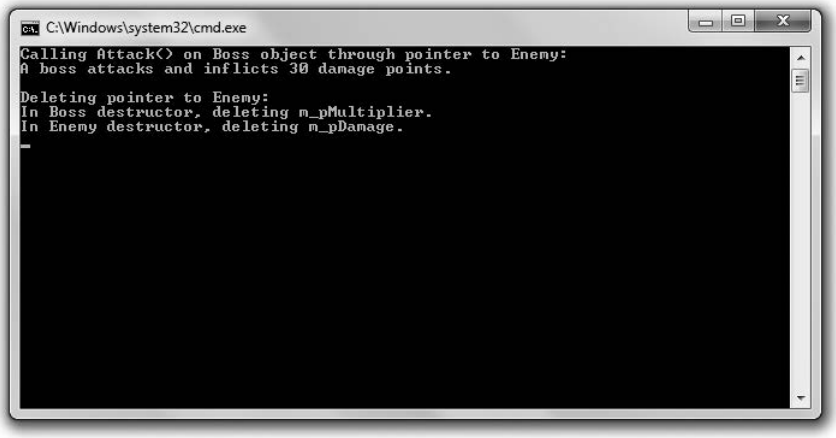


Рис. 10.4. Благодаря полиморфизму для объектов, на которые указывают указатели базового класса, вызываются требуемые функции-члены (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код этой программы с сайта Cengage Learning (www.cengageptr.com/downloads). Программа располагается в каталоге к главе 10, файл называется polymorphic_bad_guy.cpp.

```
// Polymorphic Bad Guy
// Демонстрирует динамический вызов функций-членов
#include <iostream>
using namespace std;
class Enemy
{
    public:
        Enemy(int damage = 10);
        virtual ~Enemy();
        void virtual Attack() const;
    protected:
        int* m_pDamage;
};
Enemy::Enemy(int damage)
{
    m_pDamage = new int(damage);
```

```
}
Enemy::~Enemy()
{
    cout << "In Enemy destructor, deleting m_pDamage.\n";
    delete m_pDamage;
    m_pDamage = 0;
}
void Enemy::Attack() const
{
    cout << "An enemy attacks and inflicts " << *m_pDamage << " damage points.";
}
class Boss : public Enemy
{
public:
    Boss(int multiplier = 3);
    virtual ~Boss();
    void virtual Attack() const;
protected:
    int* m_pMultiplier;
};
Boss::Boss(int multiplier)
{
    m_pMultiplier = new int(multiplier);
}
Boss::~Boss()
{
    cout << "In Boss destructor, deleting m_pMultiplier.\n";
    delete m_pMultiplier;
    m_pMultiplier = 0;
}
void Boss::Attack() const
{
    cout << "A boss attacks and inflicts " << (*m_pDamage) * (*m_pMultiplier)
    << " damage points.";
}
int main()
{
    cout << "Calling Attack() on Boss object through pointer to Enemy:\n";
    Enemy* pBadGuy = new Boss();
    pBadGuy->Attack();
    cout << "\n\nDeleting pointer to Enemy:\n";
    delete pBadGuy;
    pBadGuy = 0;
    return 0;
}
```

Использование указателей базового класса для объектов производного класса

Объект производного класса является также членом базового класса. Например, в программе Polymorphic Bad Guy объект класса `Boss` является также объектом класса `Enemy`. Это логично, поскольку босс на самом деле лишь особенный тип врага.

Это имеет смысл еще и потому, что объект класса Boss имеет все члены объекта класса Enemy. Что из этого следует? Поскольку объект производного класса является членом базового класса, вы можете использовать указатель на базовый класс, чтобы указать на объект производного класса. Я делаю это в функции main() с помощью следующей строки, в которой в куче создаются объект класса Boss, а также указатель на класс Enemy, pBadGuy, который указывает на объект класса Boss:

```
Enemy* pBadGuy = new Boss();
```

Зачем вам вообще делать это? Это полезно, поскольку у вас появляется возможность работать с объектами, не зная их точного типа. Например, вы можете написать функцию, которая принимает в качестве параметра указатель на тип Enemy и может работать с объектами как класса Enemy, так и класса Boss. Эта функция не обязана знать точный тип переданного ей объекта, при этом она работает с разными объектами и получает разные результаты в зависимости от их типа, если функции-члены производных классов были объявлены виртуальными. Поскольку функция Attack() объявлена виртуальной, будет вызвана правильная версия функции-члена (в зависимости от типа объекта), она не фиксируется типом указателя.

Я докажу, что поведение будет полиморфным, в функции main(). Помните, что pBadGuy — это указатель типа Enemy, который указывает на объект класса Boss. Поэтому строка:

```
pBadGuy->Attack();
```

вызывает функцию-член Attack() объекта класса Boss с помощью указателя на тип Enemy, что приводит к вызову функции-члена Attack(), определенной в классе Boss, в результате чего на экран будет выведен текст A boss attacks and inflicts 30 damage points.

ПОДСКАЗКА

Виртуальные функции гарантируют полиморфное поведение при работе как со ссылками, так и с указателями.

ОСТОРОЖНО!

Если вы переопределите невиртуальную функцию-член в производном классе и затем вызовете эту функцию-член с помощью указателя на объект базового класса, будет вызвана функция-член базового класса, а не производного. Это лучше проиллюстрировать на примере. Если бы я не объявил функцию Attack() как виртуальную в программе Polymorphic Bad Guy, а затем вызвал эту функцию-член через указатель на класс Enemy для объекта класса Boss с помощью конструкции pBadGuy->Attack();, на экране появилось бы сообщение An enemy attacks and inflicts 10 damage points. Это происходит в результате раннего связывания, благодаря которому точная функция-член определяется на основании типа указателя — в данном случае Enemy. Но, поскольку функция Attack() объявлена как виртуальная, вызываемая функция будет определена на этапе выполнения программы в зависимости от типа объекта, на который указывает pBadGuy, — в данном случае Boss. Такого результата можно достичь благодаря позднему связыванию, поскольку функция Attack() является виртуальной. Мораль заключается в том, что переопределять следует только виртуальные функции-члены.

Применение виртуальных функций, помимо преимуществ, имеет и недостатки — требования к производительности будут повышены. Поэтому использовать виртуальные функции следует только в том случае, если это вам действительно нужно.

Определение виртуальных деструкторов

Если вы используете указатель типа базового класса, чтобы указать на объект производного класса, у вас потенциально может возникнуть проблема. Когда вы удалите указатель, будет вызван только деструктор базового класса. Это может привести к катастрофическим результатам, поскольку может возникнуть необходимость вызвать деструктор базового класса, чтобы освободить память (что делает деструктор класса `Boss`). Решением, как вы могли догадаться, является объявление деструктора базового класса виртуальным. Таким образом, вызов деструктора производного класса, что (как всегда) приведет к вызову деструктора базового класса, дает каждому классу возможность прибрать за собой. Я пользуюсь этой теорией на практике, объявив деструктор класса `Enemy` виртуальным:

```
virtual ~Enemy();
```

В функции `main()`, когда я с помощью строки:

```
delete pBadGuy;
```

удаляю указатель, указывающий на объект класса `Boss`, вызывается деструктор класса `Boss`, что освобождает память в куче, на которую указывает переменная `m_pDamageMultiplier`, и отображает сообщение `In Boss destructor`, удаляя член данных `m_pMultiplier`.

Далее вызывается конструктор класса `Enemy`, что освобождает память в куче, на которую указывал член данных `m_pDamage`, и выводит сообщение `In Enemy destructor`, удаляя переменную `m_pDamage`. Объект уничтожается, и вся память, связанная с объектом, освобождается.

ПРИЕМ

Если вы пишете класс, имеющий виртуальные функции, возьмите за правило объявлять конструктор такого класса виртуальным.

Использование абстрактных классов

Иногда вам может понадобиться определить класс, который будет являться базовым для других классов, но объекты этого класса создавать не нужно, поскольку он слишком общий. Например, предположим, что вы пишете игру, в которой присутствуют существа нескольких типов. Несмотря на то что все они разные, они имеют две общие черты: значение здоровья и приветствие. Поэтому вы можете определить класс `Creature` как базовый, из которого будут унаследованы другие классы, например `Pixie`, `Dragon`, `Orc` и т. д. Несмотря на то что класс `Creature` полезен, не имеет смысла создавать его объекты. Было бы неплохо иметь способ указать, что класс `Creature` является базовым и не предназначен для того, чтобы создавались его объекты. Язык C++ позволяет вам определить такой класс, он называется абстрактным.

Знакомство с программой Abstract Creature

В программе Abstract Creature демонстрируется использование абстрактных классов. В этой программе я определяю абстрактный класс `Creature`, который может быть использован как базовый для создания определенных классов существ. Я определяю один такой класс — `Orc`. Далее создаю объект класса `Orc` и вызываю одну функцию-член, чтобы заставить орка проворчать приветствие, а затем другую функцию-член, чтобы отобразить уровень его здоровья. Результат работы программы показан на рис. 10.5.

Рис. 10.5. Орк — это объект, созданный с помощью класса, являющегося абстрактным классом для всех существ (опубликовано с разрешения компании Microsoft)

Вы можете загрузить код программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 10, файл называется `abstract_creature.cpp`.

```
// Abstract Creature
// Демонстрация использования абстрактных классов
#include <iostream>
using namespace std;
class Creature // абстрактный класс
{
    public:
        Creature(int health = 100);
        virtual void Greet() const = 0; // чистая виртуальная функция-член
        virtual void DisplayHealth() const;
    protected:
        int m_Health;
};
Creature::Creature(int health):
m_Health(health)
{}
void Creature::DisplayHealth() const
{
```

```
    cout << "Health: " << m_Health << endl;
}
class Orc : public Creature
{
public:
    Orc(int health = 120);
    virtual void Greet() const;
};
Orc::Orc(int health):
Creature(health)
{}
void Orc::Greet() const
{
    cout << "The orc grunts hello.\n";
}

int main()
{
    Creature* pCreature = new Orc();
    pCreature->Greet();
    pCreature->DisplayHealth();
    return 0;
}
```

Объявление чистых виртуальных функций

Чистая виртуальная функция — это такая функция, которой не нужно давать определение. Логика заключается в том, что для функции в рамках этого класса нет хорошего определения. Например, я не думаю, что есть смысл определять функцию `Greet()` в рамках класса `Creature`, поскольку приветствие зависит от определенного типа существа — пикси мерцают, дракон выдыхает клубы дыма, а орк ворчит.

Вы указываете, что функция является чистой виртуальной, разместив после объявления функции знак «равно» и `0`. Я сделал это в классе `Creature` с помощью следующей строки:

```
virtual void Greet() const = 0; // чистая виртуальная функция-член
```

Если класс содержит хотя бы одну чистую виртуальную функцию, он считается абстрактным. Поэтому класс `Creature` является абстрактным. Я могу использовать его как базовый класс для других классов, но не могу создавать его объекты.

Абстрактные классы могут иметь члены данных и виртуальные функции, которые не являются чистыми виртуальными. В классе `Creature` я объявляю член данных `m_Health` и виртуальную функцию-член `DisplayHealth()`.

Наследование от абстрактного класса

Когда класс наследует от абстрактного класса, вы можете переопределить его чистые виртуальные функции. Если вы переопределите все чистые виртуальные функции, класс перестанет быть абстрактным и вы получите возможность создавать его

объекты. Когда я создаю класс Orc на основе класса Creature, я переопределяю единственную виртуальную функцию класс Creature с помощью следующих строк:

```
void Orc::Greet() const
{
    cout << "The orc grunts hello.\n";
}
```

Это значит, что я могу создавать объекты класса Orc, что я и делаю в функции main() с помощью следующей строки:

```
Creature* pCreature = new Orc();
```

Этот код создает в куче новый объект класса Orc и присваивает фрагмент памяти, занятый этим объектом, pCreature, указателю типа Creature. Несмотря на то что я не могу создавать объекты класса Creature, с использованием этого класса можно создать указатель. Как и все указатели типа базового класса, указатель типа Creature может указывать на любой объект производного класса, например класса Orc. Далее я вызываю Greet() — чистую виртуальную функцию, которую переопределил в классе Orc, с помощью следующей строки:

```
pCreature->Greet();
```

На экране отображается правильное приветствие: The orc grunts hello.

Наконец, я вызываю функцию DisplayHealth(), которую определил в классе Creature:

```
pCreature->DisplayHealth();
```

Она также отображает правильное сообщение: Health: 120.

Знакомство с игрой Blackjack

Финальный проект этой главы — упрощенная версия азартной карточной игры Blackjack (сукно для стола поставляется отдельно). Игра работает следующим образом: игрокам раздаются карты, за каждую из которых начисляются очки. Каждый игрок пытается заработать 21 очко, не больше. За каждую карту с числом дается столько очков, сколько на ней указано. За туз дается либо 1 очко, либо 11 (в зависимости от того, что подходит игроку больше), а за валета, даму и короля — по 10 очков.

Компьютер является дилером (казино) и играет против группы от одного до семи игроков. В начале кона все участники (включая дилера) получают две карты. Игроки могут видеть все свои карты, а также сумму очков. Однако одна из карт дилера скрыта на протяжении всего кона.

Далее каждый игрок получает возможность брать дополнительные карты до тех пор, пока ему не надоест. Если сумма очков игрока превысит 21 (эта ситуация известна как перебор), игрок проигрывает. После того как все игроки получили возможность набрать дополнительные карты, дилер открывает скрытую карту. Далее он обязан брать новые карты до тех пор, пока сумма его очков менее или равна 16.

Если у дилера перебор, все игроки, не имеющие перебора, побеждают. В противном случае сумма очков каждого из оставшихся игроков сравнивается с суммой очков дилера. Если сумма очков игрока больше суммы очков дилера, он выигрывает. В противном случае игрок проигрывает. Если суммы очков игрока и дилера одинаковы, засчитывается ничья (такая ситуация называется пуш). Игра показана на рис. 10.6.

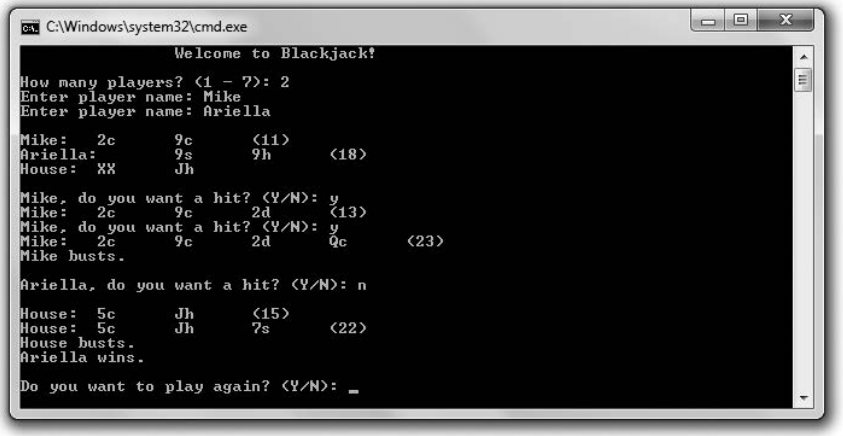


Рис. 10.6. Один игрок побеждает, другому не так повезло (опубликовано с разрешения компании Microsoft)

Разработка классов

Перед тем как начать программировать проект, имеющий несколько классов, полезно перечислить эти классы на бумаге. Вам следует создать список и включить туда краткое описание каждого класса. В табл. 10.1 показана моя первая попытка создать такой список для игры Blackjack.

Таблица 10.1. Классы игры Blackjack

Класс	Базовый класс	Описание
Card	Нет	Карта для игры в Blackjack
Hand	Нет	Набор карт для игры в Blackjack. Коллекция объектов класса Card
Deck	Hand	Имеет дополнительную функциональность, которая отсутствует в классе Hand, в частности тасование и раздачу
GenericPlayer	Hand	Обобщенно описывает игрока в Blackjack. Не является полноценным игроком, а лишь содержит элементы, характерные как для игрока-человека, так и для игрока-компьютера
Player	GenericPlayer	Человек — игрок в Blackjack
House	GenericPlayer	Компьютер — игрок в Blackjack (дилер)
Game	Нет	Игра в Blackjack

Для простоты все функции-члены будут открытыми, а все члены данных — защищенными. Я также стану использовать только открытое наследование, что означает, что каждый производный класс будет наследовать все члены базового класса.

В дополнение к описанию классов словами довольно полезно нарисовать что-то похожее на генеалогическое древо, чтобы визуализировать связь классов. Я сделал это на рис. 10.7.

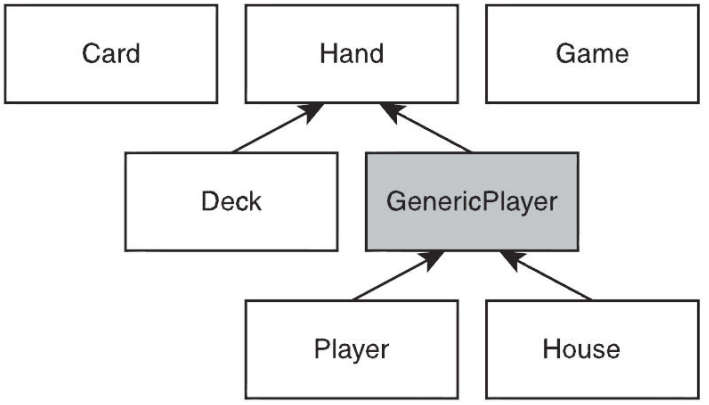


Рис. 10.7. Иерархрия наследования классов игры Blackjack. Класс GenericPlayer затенен, поскольку он будет абстрактным

Далее неплохо расписать классы более подробно. Спросите себя о классах. Что конкретно они будут представлять? Что они будут делать? Как они будут работать с другими классами? Я вижу объекты класса `Card` как реальные карты. При раздаче карты из колоды вы не копируете ее, а перемещаете. Для меня это значит, что класс `Hand` будет иметь в качестве члена данных вектор, где станут храниться указатели на объекты типа `Card`, которые разместятся в куче. Когда карта раздается, указатели будут копироваться и удаляться.

Игроков в Blackjack (как людей, так и компьютер) я рассматриваю как руки, имеющие имена. Именно поэтому я создаю классы `Player` и `House` (опосредованно) на базе класса `Hand`. (Согласно другому подходу каждый игрок имеет руку. Если следовать в этом направлении, классы `Player` и `House` будут иметь член данных типа `Hand` вместо того, чтобы наследовать от класса `Hand`.)

Я определяю класс `GenericPlayer`, в котором будет размещаться общая функциональность классов `Player` и `House`, чтобы не дублировать ее в обоих классах.

Помимо этого, я рассматриваю колоду отдельно от дилера. Карты колоды будут раздаваться игрокам-людям и игроку-компьютеру на равных. Это значит, что у класса `Deck` будет функция-член, предназначенная для раздачи карт. Эта функция будет полиморфной — она будет работать как для объекта типа `Player`, так и для объекта типа `House`.

Для того чтобы все выглядело красиво, вы можете перечислить все члены данных и функции-члены, которые, по вашему мнению, будут иметь классы, а также

привести их краткое описание. Я делаю это в табл. 10.2–10.8. Для каждого класса я перечислил члены, которые будут в нем описаны. Конечно, некоторые классы будут наследовать члены других классов.

Таблица 10.2. Класс Card

Член	Описание
rank m_Rank	Значение карты (туз, двойка, тройка и т. д.). rank — это перечисление, куда входят все 13 значений
suit m_Suit	Масть карты (трефы, бубны, черви или пики). suit — это перечисление, содержащее все четыре возможные масти
bool m_IsFaceUp	Указывает, как расположена карта — лицом вверх или лицом вниз. Влияет на то, отображается она или нет
int GetValue()	Возвращает значение карты
void Flip()	Переворачивает карту. Может использоваться для того, чтобы перевернуть карту лицом вверх или вниз

Таблица 10.3. Класс Hand

Член	Описание
vector<Card*> m_Cards	Коллекция карт. Хранит указатели на объекты типа Card
void Add(Card* pCard)	Добавляет карту в руку. Добавляет указатель на объект типа Card в вектор m_Cards
void Clear()	Очищает руку от карт. Удаляет все указатели из вектора m_Cards, удаляя все связанные с ними объекты в куче
int GetTotal()	Возвращает сумму очков карт руки

Таблица 10.4. Класс GenericPlayer (абстрактный)

Член	Описание
string m_Name	Имя игрока
virtual bool IsHitting() const = 0	Указывает, нужна ли игроку еще одна карта. Чистая виртуальная функция
bool IsBusted() const	Указывает, что у игрока перебор
void Bust() const	Объявляет, что у игрока перебор

Таблица 10.5. Класс Player

Член	Описание
virtual bool IsHitting() const	Указывает, нужна ли игроку еще одна карта
void Win() const	Объявляет, что игрок выиграл
void Lose() const	Объявляет, что игрок проиграл
void Push() const	Объявляет, что игрок сыграл вничью

Таблица 10.6. Класс House

Член	Описание
virtual bool IsHitting() const	Указывает, нужна ли игроку еще одна карта
void FlipFirstCard()	Переворачивает первую карту

Таблица 10.7. Класс Deck

Член	Описание
void Populate()	Создает стандартную колоду из 52 карт
void Shuffle()	Тасует карты
void Deal(Hand& aHand)	Раздает в руку одну карту
void AdditionalCards (GenericPlayer& aGenericPlayer)	Раздает игроку дополнительные карты до тех пор, пока игрок может и хочет их получать

Таблица 10.8. Класс Game

Член	Описание
Deck m_Deck	Колода карт
House m_House	Рука дилера
vector<Player> m_Players	Группа игроков-людей. Вектор, содержащий объекты типа Player
void Play()	Проводит кон игры Blackjack

Планирование логики игры

Последняя часть планирования программы заключается в том, что я обрисовываю базовый поток одного кона игры. Я написал псевдокод функции-члена `Play()` класса `Game`. Рассмотрим его.

```
Раздать игрокам и дилеру две начальные карты
Спрятать первую карту дилера
Отобразить руки игроков и дилера
Раздать игрокам дополнительные карты
Показать первую карту дилера
Раздать дополнительные карты дилеру
Если у дилера перебор
Все игроки без перебора выигрывают
Иначе
Для каждого игрока
Если у игрока нет перебора
Если сумма очков игрока больше, чем у дилера
Игрок побеждает
Иначе если сумма очков игрока меньше, чем у дилера
```

```
Игрок проигрывает
Иначе
Объявляется ничья
Удалить карты всех игроков
```

К этому моменту вы должны были ознакомиться с программой Blackjack, не увидев при этом ни одной строки кода! Планирование может быть так же важно, как и кодирование (если не больше). Поскольку я потратил много времени на описание классов, не буду описывать каждый фрагмент кода, только выделю основные или новые идеи.

Вы можете загрузить код программы с сайта Cengage Learning (**www.cengageptr.com/downloads**). Программа располагается в каталоге к главе 10, файл называется `blackjack.cpp`.

ПОДСКАЗКА

Файл blackjack.cpp содержит семь классов. В программировании на языке C++ можно разбивать подобные файлы на несколько файлов в соответствии с количеством классов. Однако тема написания программы, использующей несколько файлов, лежит за пределами этой книги.

Класс Card

После начальных утверждений я определяю класс `Card`, представляющий отдельную игральную карту.

```
// Blackjack
// Играет упрощенную версию игры Blackjack: от одного до семи игроков
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
using namespace std;
class Card
{
    public:
        enum rank {ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
        JACK, QUEEN, KING};
        enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
        // перегружаем оператор <<, чтобы можно было отправить объект
        // типа Card в стандартный поток вывода
        friend ostream& operator<<(ostream& os, const Card& aCard);
        Card(rank r = ACE, suit s = SPADES, bool ifu = true);
        // возвращает значение карты, от 1 до 11
        int GetValue() const;
        // переворачивает карту: карта, лежащая лицом вверх,
        // переворачивается лицом вниз и наоборот
        void Flip();
    private:
        rank m_Rank;
```

```
        suit m_Suit;
        bool m_IsFaceUp;
};
Card::Card(rank r, suit s, bool ifu): m_Rank(r), m_Suit(s), m_IsFaceUp(ifu)
{}
int Card::GetValue() const
{
    // если карта перевернута лицом вниз, ее значение равно 0
    int value = 0;
    if (m_IsFaceUp)
    {
        // значение - это число, указанное на карте
        value = m_Rank;
        // значение равно 10 для открытых карт
        if (value > 10)
        {
            value = 10;
        }
    }
    return value;
}
void Card::Flip()
{
    m_IsFaceUp = !(m_IsFaceUp);
}
```

Я определил два перечисления, `rank` и `suit`, чтобы использовать их как типы для членов данных `m_Rank` и `m_Suit`. Это позволяет получить два преимущества. Во-первых, код становится более читаемым. Член данных, отвечающий за масть, будет иметь значение `CLUBS` или `HEARTS` вместо значений 0 или 2. Во-вторых, это ограничивает возможные значения этих двух членов данных. Член данных `m_Suit` может хранить лишь значения перечисления `suit`, а член данных `m_Rank` — только значения перечисления `rank`.

Далее я объявляю перегруженную дружественную функцию `operator<<()`, чтобы можно было отобразить объект карты на экране.

Функция `GetValue()` возвращает значение объекта `Card`, которое может лежать в диапазоне от 0 до 11. Тузы всегда имеют значение 11 (я присваиваю им значение 1 при необходимости в классе `Hand`). Карта, лежащая лицом вниз, имеет значение 0.

Класс Hand

Я определяю класс `Hand`, который представляет собой коллекцию карт.

```
class Hand
{
    public:
        Hand();
        virtual ~Hand();
```

```
        // добавляет карту в руку
        void Add(Card* pCard);
        // очищает руку от карт
        void Clear();
        // получает сумму очков карт в руке, присваивая тузу
        // значение 1 или 11 в зависимости от ситуации
        int GetTotal() const;
    protected:
        vector<Card*> m_Cards;
};
Hand::Hand()
{
    m_Cards.reserve(7);
}
Hand::~Hand()
{
    Clear();
}
void Hand::Add(Card* pCard)
{
    m_Cards.push_back(pCard);
}
void Hand::Clear()
{
    // проходит по вектору, освобождая всю память в куче
    vector<Card*>::iterator iter = m_Cards.begin();
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        delete *iter;
        *iter = 0;
    }
    // очищает вектор указателей
    m_Cards.clear();
}
int Hand::GetTotal() const
{
    // если карт в руке нет, возвращает значение 0
    if (m_Cards.empty())
    {
        return 0;
    }
    // если первая карта имеет значение 0, то она лежит рубашкой вверх;
    // вернуть значение 0
    if (m_Cards[0]->GetValue() == 0)
    {
        return 0;
    }
    // находит сумму очков всех карт, каждый туз дает 1 очко
    int total = 0;
    vector<Card*>::const_iterator iter;
```

```
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        total += (*iter)->GetValue();
    }
    // определяет, держит ли рука туз
    bool containsAce = false;
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        if ((*iter)->GetValue() == Card::ACE)
        {
            containsAce = true;
        }
    }
    // если рука держит туз и сумма довольно маленькая, туз дает 11 очков
    if (containsAce && total <= 11)
    {
        // добавляем только 10 очков, поскольку мы уже добавили
        // за каждый туз по одному очку
        total += 10;
    }
    return total;
}
```

ОСТОРОЖНО!

Деструктор этого класса виртуальный, но обратите внимание на то, что я не использую ключевое слово virtual за пределами класса, когда определяю деструктор. Ключевое слово virtual используется только внутри определения класса. Не волнуйтесь, деструктор останется виртуальным.

Несмотря на то что я уже рассмотрел этот вопрос, хочу снова обратить внимание на него. Любая коллекция карт, например объект `Hand`, будет содержать вектор указателей на группу этих объектов, размещенных в куче.

Функция-член `Clear()` решает важную задачу. Она не только удаляет все указатели из вектора `m_Cards`, но и удаляет связанные объекты типа `Card` и освобождает занятую ими память в куче. Это работает так же, как и в реальном мире, когда в конце кона карты сбрасываются. Виртуальный деструктор вызывает метод `Clear()`.

Функция-член `GetTotal()` возвращает сумму очков для карт в руке. Если рука содержит туз, он считается за 1 или 11 очков в зависимости от остальных карт. Количество очков, которое дает туз, определяется так: если в руке есть туз, он дает 11 очков, затем выполняется проверка, превышает ли сумма очков карт в руке число 21. Если нет, то количество очков, которое дает туз, не изменяется. В противном случае туз даст 1 очко.

Класс GenericPlayer

Я определяю класс `GenericPlayer`, обобщенно представляющий игрока в Blackjack. Он представляет не полноценного игрока, а общие элементы игрока-человека и игрока-компьютера.

```
class GenericPlayer : public Hand
{
    friend ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);
    public:
        GenericPlayer(const string& name = "");
        virtual ~GenericPlayer();
        // показывает, хочет ли игрок продолжать брать карты
        virtual bool IsHitting() const = 0;
        // возвращает значение, если игрок имеет перебор -
        // сумму очков, большую 21
        bool IsBusted() const;
        // объявляет, что игрок имеет перебор
        void Bust() const;
    protected:
        string m_Name;
};
GenericPlayer::GenericPlayer(const string& name):
m_Name(name)
{}
GenericPlayer::~GenericPlayer()
{}
bool GenericPlayer::IsBusted() const
{
    return (GetTotal() > 21);
}
void GenericPlayer::Bust() const
{
    cout << m_Name << " busts.\n";
}
```

Я объявил перегруженную функцию `operator<<()` дружественной классу, чтобы иметь возможность отображать на экране объект класса `GenericPlayer`. Эта функция принимает ссылку на объект типа `GenericPlayer`, что значит, что она также может принимать ссылки на объекты типа `Player` или `House`.

Конструктор принимает строку, представляющую собой имя игрока. Деструктор автоматически становится виртуальным, поскольку он наследует эту черту от класса `Hand`.

Функция-член `IsHitting()` показывает, хочет ли игрок взять еще одну карту. Поскольку эта функция-член не имеет реального значения для обобщенного представления игрока, я сделал ее чисто виртуальной. Благодаря этому класс `GenericPlayer` становится абстрактным. Это также означает, что в классах `Player` и `House` должны быть реализованы собственные версии этой функции.

Функция-член `IsBusted()` показывает, есть ли у игрока перебор. Поскольку перебор у игроков и дилера одинаков — сумма очков их карт превосходит 21, — я разместил определение этой функции внутри данного класса.

Функция-член `Bust()` объявляет, что у игрока перебор. Поскольку перебор для игроков и дилера объявляется одинаковым образом, я разместил определение этой функции внутри данного класса.

Класс Player

Класс Player представляет игрока-человека. Он наследует от класса GenericPlayer.

```
class Player : public GenericPlayer
{
    public:
        Player(const string& name = "");
        virtual ~Player();
        // показывает, хочет ли игрок продолжать брать карты
        virtual bool IsHitting() const;
        // объявляет, что игрок победил
        void Win() const;
        // объявляет, что игрок проиграл
        void Lose() const;
        // объявляет ничью
        void Push() const;
    };
Player::Player(const string& name):
GenericPlayer(name)
{}
Player::~Player()
{}
bool Player::IsHitting() const
{
    cout << m_Name << ", do you want a hit? (Y/N): ";
    char response;
    cin >> response;
    return (response == 'y' || response == 'Y');
}
void Player::Win() const
{
    cout << m_Name << " wins.\n";
}
void Player::Lose() const
{
    cout << m_Name << " loses.\n";
}
void Player::Push() const
{
    cout << m_Name << " pushes.\n";
}
```

Этот класс реализует функцию-член IsHitting(), которая унаследована от класса GenericPlayer. Поэтому класс Player не является абстрактным. Класс реализует функцию-член, спрашивая человека, хочет ли он взять еще одну карту. Если игрок-человек вводит символ y или Y в ответ на этот вопрос, функция-член возвращает значение true, что показывает: игрок хочет взять еще одну карту. Если же игрок-человек вводит любой другой символ, эта функция возвращает значение false, что показывает: игрок больше не хочет брать карту.

Функции-члены Win(), Lose() и Push() просто объявляют, что игрок выиграл, проиграл или сыграл вничью соответственно.

Класс House

Класс House представляет дилера. Он наследует от класса GenericPlayer.

```
class House : public GenericPlayer
{
    public:
        House(const string& name = "House");
        virtual ~House();
        // показывает, хочет ли игрок продолжать брать карты
        virtual bool IsHitting() const;
        // переворачивает первую карту
        void FlipFirstCard();
};
House::House(const string& name):
GenericPlayer(name)
{}
House::~House()
{}
bool House::IsHitting() const
{
    return (GetTotal() <= 16);
}
void House::FlipFirstCard()
{
    if (!(m_Cards.empty()))
    {
        m_Cards[0]->Flip();
    }
    else
    {
        cout << "No card to flip!\n";
    }
}
```

Этот класс реализует функцию-член IsHitting(), которая унаследована от класса GenericPlayer. Поэтому класс House не является абстрактным. Класс реализует эту функцию-член, вызывая функцию GetTotal(). Если возвращенное значение меньше или равно 16, функция-член возвращает значение true, что показывает: дилер хочет взять еще одну карту. В противном случае функция возвращает значение false, что показывает: дилеру карты больше не нужны.

Функция FlipFirstCard() переворачивает первую карту дилера. Эта функция-член является необходимой, поскольку дилер скрывает свою первую карту в начале кона, а затем показывает ее после того, как все игроки взяли дополнительные карты.

Класс Deck

Класс Deck представляет колоду карт. Он наследует от класса Hand.

```
class Deck : public Hand
{
    public:
```

```
        Deck();
        virtual ~Deck();
        // создает стандартную колоду из 52 карт
        void Populate();
        // тасует карты
        void Shuffle();
        // раздает одну карту в руку
        void Deal(Hand& aHand);
        // дает дополнительные карты игроку
        void AdditionalCards(GenericPlayer& aGenericPlayer);
};
Deck::Deck()
{
    m_Cards.reserve(52);
    Populate();
}
Deck::~Deck()
{}
void Deck::Populate()
{
    Clear();
    // создает стандартную колоду
    for (int s = Card::CLUBS; s <= Card::SPADES; ++s)
    {
        for (int r = Card::ACE; r <= Card::KING; ++r)
        {
            Add(new Card(static_cast<Card::rank>(r), static_cast<Card::suit>(s)));
        }
    }
}
void Deck::Shuffle()
{
    random_shuffle(m_Cards.begin(), m_Cards.end());
}
void Deck::Deal(Hand& aHand)
{
    if (!m_Cards.empty())
    {
        aHand.Add(m_Cards.back());
        m_Cards.pop_back();
    }
    else
    {
        cout << "Out of cards. Unable to deal.";
    }
}
void Deck::AdditionalCards(GenericPlayer& aGenericPlayer)
{
    cout << endl;
    // продолжает раздавать карты до тех пор, пока у игрока не случается
    // перебор или пока он хочет взять еще одну карту
    while (!(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting() )
```

```
    {
        Deal(aGenericPlayer);
        cout << aGenericPlayer << endl;
        if (aGenericPlayer.IsBusted())
        {
            aGenericPlayer.Bust();
        }
    }
}
```

ПОДСКАЗКА

Преобразование типов — это способ преобразования значения одного типа в значение другого типа. Один из способов выполнения преобразования — применение static_cast. Вы можете использовать static_cast, чтобы вернуть значение нового типа, образованное из значения другого типа, определив желаемый тип между символами < и >, за которыми в скобках будет стоять преобразуемое значение. В этом примере конструкция вернет значение 5.0, которое будет иметь тип double:

```
static_cast<double>(5);
```

Функция `Populate()` создает стандартную колоду из 52 карт. Функция-член проходит по всем возможным комбинациям значений перечислений `Card::suit` и `Card::rank values`. Она использует `static_cast`, чтобы преобразовать целочисленные переменные в значения перечислений, определенных в классе `Card`.

Функция `Shuffle()` тасует колоду карт. Она в случайном порядке переставляет указатели, расположенные в векторе `m_Cards` с помощью функции `random_shuffle()` стандартной библиотеки шаблонов (Standard Template Library). Именно поэтому я включил заголовочный файл `<algorithm>`.

Функция `Deal()` выдает одну карту из колоды в руку. Она добавляет копию указателя в конец вектора `m_Cards` с помощью функции-члена `Add()`. Далее она удаляет указатель из конца вектора `m_Cards`, что, по сути, является перемещением карты. Функция `Deal()` довольно мощная, поскольку она принимает ссылку на объект типа `Hand`, что означает: она может работать также с объектами классов `Player` и `House`. Благодаря магии полиморфизма функция `Deal()` может вызывать функцию-член `Add()` объекта любого из этих классов, не зная его конкретный тип.

Функция `AdditionalCards()` дает дополнительные карты игроку до тех пор, пока он этого хочет или пока у него не образуется перебор. Функция-член принимает ссылку на объект типа `GenericPlayer`, поэтому вы можете передать ей объект типа `Player` или `House`. Опять же благодаря магии полиморфизма функция `AdditionalCards()` не обязана знать, с объектом какого именно типа она работает. Она может вызывать функции-члены `IsBusted()` и `IsHitting()`, не зная типа объекта, при этом будет выполнен корректный код.

Класс Game

Класс `Game` представляет игру Blackjack.

```
class Game
{
    public:
        Game(const vector<string>& names);
```

```
        ~Game();
        // проводит игру в Blackjack
        void Play();
    private:
        Deck m_Deck;
        House m_House;
        vector<Player> m_Players;
};
Game::Game(const vector<string>& names)
{
    // создает вектор игроков из вектора с именами
    vector<string>::const_iterator pName;
    for (pName = names.begin(); pName != names.end(); ++pName)
    {
        m_Players.push_back(Player(*pName));
    }
    // засевает генератор случайных чисел
    srand(static_cast<unsigned int>(time(0)));
    m_Deck.Populate();
    m_Deck.Shuffle();
}
Game::~Game()
{}
void Game::Play()
{
    // раздает каждому по две стартовые карты
    vector<Player>::iterator pPlayer;
    for (int i = 0; i < 2; ++i)
    {
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        {
            m_Deck.Deal(*pPlayer);
        }
        m_Deck.Deal(m_House);
    }
    // прячет первую карту дилера
    m_House.FlipFirstCard();
    // открывает руки всех игроков
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        cout << *pPlayer << endl;
    }
    cout << m_House << endl;
    // раздает игрокам дополнительные карты
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        m_Deck.AdditionalCards(*pPlayer);
    }
    // показывает первую карту дилера
    m_House.FlipFirstCard();
    cout << endl << m_House;
```

```
    // раздает дилеру дополнительные карты
    m_Deck.AdditionalCards(m_House);
    if (m_House.IsBusted())
    {
        // все, кто остался в игре, побеждают
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        {
            if ( !(pPlayer->IsBusted()) )
            {
                pPlayer->Win();
            }
        }
    }
    else
    {
        // сравнивает суммы очков всех оставшихся игроков с суммой очков дилера
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        {
            if ( !(pPlayer->IsBusted()) )
            {
                if (pPlayer->GetTotal() > m_House.GetTotal())
                {
                    pPlayer->Win();
                }
                else if (pPlayer->GetTotal() < m_House.GetTotal())
                {
                    pPlayer->Lose();
                }
                else
                {
                    pPlayer->Push();
                }
            }
        }
    }
    // очищает руки всех игроков
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        pPlayer->Clear();
    }
    m_House.Clear();
}
```

Конструктор этого класса принимает ссылку на вектор строк, представляющих имена игроков-людей. Конструктор создает объект класса `Player` для каждого имени.

Далее засевается генератор случайных чисел, а затем создается и тасуется колода карт.

Функция-член `Play()` реализует тот псевдокод, который я написал ранее. Псевдокод описывает порядок разыгрывания кона.

Функция main()

После объявления функций перегруженного оператора operator<<() я создаю функцию main().

```
// прототипы функций
ostream& operator<<(ostream& os, const Card& aCard);
ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);
int main()
{
    cout << "\t\tWelcome to Blackjack!\n\n";
    int numPlayers = 0;
    while (numPlayers < 1 || numPlayers > 7)
    {
        cout << "How many players? (1 - 7): ";
        cin >> numPlayers;
    }
    vector<string> names;
    string name;
    for (int i = 0; i < numPlayers; ++i)
    {
        cout << "Enter player name: ";
        cin >> name;
        names.push_back(name);
    }
    cout << endl;
    // игровой цикл
    Game aGame(names);
    char again = 'y';
    while (again != 'n' && again != 'N')
    {
        aGame.Play();
        cout << "\nDo you want to play again? (Y/N): ";
        cin >> again;
    }
    return 0;
}
```

Функция main() получает имена всех игроков и помещает их в вектор строк, а затем создает объект класса Game и передает в него ссылку на вектор. Функция main() вызывает функцию Play() объекта класса Game до тех пор, пока игроки не покажут, что больше не хотят играть.

Перегрузка функции operator<<()

Следующее определение функции перегружает оператор <<, что позволяет мне отправить объект типа Card в стандартный поток вывода.

```
// перегружает оператор <<, чтобы получить возможность отправить
// объект типа Card в поток cout
```

```
ostream& operator<<(ostream& os, const Card& aCard)
{
    const string RANKS[] = {"0", "A", "2", "3", "4", "5", "6", "7", "8", "9",
    "10", "J", "Q", "K"};
    const string SUITS[] = {"c", "d", "h", "s"};
    if (aCard.m_IsFaceUp)
    {
        os << RANKS[aCard.m_Rank] << SUITS[aCard.m_Suit];
    }
    else
    {
        os << "XX";
    }
    return os;
}
```

Функция использует значения членов данных rank и suit объекта как индексы массива. Я начинаю массив RANKS со значения 0, чтобы компенсировать тот факт, что значения перечисления мастей, определенного в классе Card, начинаются с 1.

Последнее описание функции перегружает оператор << так, что я могу отправить объект класса GenericPlayer в стандартный поток ввода-вывода.

```
// перегружает оператор <<, чтобы получить возможность отправить
// объект типа GenericPlayer в поток cout
ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer)
{
    os << aGenericPlayer.m_Name << ":\t";
    vector<Card*>::const_iterator pCard;
    if (!aGenericPlayer.m_Cards.empty())
    {
        for (pCard = aGenericPlayer.m_Cards.begin();
            pCard != aGenericPlayer.m_Cards.end();
            ++pCard)
        {
            os << *(*pCard) << "\t";
        }
        if (aGenericPlayer.GetTotal() != 0)
        {
            cout << "(" << aGenericPlayer.GetTotal() << ")";
        }
    }
    else
    {
        os << "<empty>";
    }
    return os;
}
```

Функция отображает имя игрока и его карты, а также общую сумму очков его карт.

Резюме

В этой главе вы должны были выучить следующий материал.

- Один из ключевых элементов ООП — это наследование, оно позволяет вам создавать классы на основе уже существующих. Новый класс автоматически наследует члены данных и функции-члены базового класса.
- Производный класс не наследует конструкторы, конструкторы копирования, деструкторы и перегруженные операторы присваивания.
- Когда создается объект производного класса, конструкторы базового класса вызываются автоматически перед конструкторами производного класса.
- Когда объект производного класса уничтожается, деструкторы базового класса вызываются автоматически после деструкторов производного класса.
- Защищенные члены доступны внутри их классов, а также некоторых производных классов в зависимости от уровня доступа, указанного при наследовании.
- Использование открытого наследования означает, что открытые члены базового класса станут открытыми членами данных производного класса, защищенные члены базового класса станут защищенными членами производного класса, а закрытые члены будут (как всегда) недоступны.
- Вы можете переопределить функции-члены, указав для них новые определения в производном классе.
- Вы можете явно вызвать функцию-член базового класса из производного класса.
- Вы можете явно вызвать конструктор базового класса из конструктора производного класса.
- Полиморфизм — это возможность функции-члена производить разные результаты в зависимости от типа вызвавшего ее объекта.
- Виртуальные функции позволяют использовать полиморфическое поведение. Если функция объявлена виртуальной, она будет оставаться таковой и в производных классах.
- Чистая виртуальная функция — это функция, для которой не нужно давать определение. Вы указываете, что функция является чистой виртуальной, поместив знак = и 0 после заголовка функции.
- Абстрактный класс имеет как минимум одну виртуальную функцию.
- Нельзя создать объект абстрактного класса.

Вопросы и ответы

1. *Как много уровней наследования можно иметь в программе?*

   Теоретически столько, сколько вам хочется. Но поскольку вы только начинаете программировать, старайтесь не усложнять код и не выходить за рамки нескольких уровней.

2. *Наследуется ли дружественность? То есть, если функция является дружественной базовому классу, будет ли она автоматически считаться дружественной производному классу?*

   Нет.

3. *Может ли класс иметь более одного непосредственного базового класса?*

   Да. Это называется множественным наследованием. Эта особенность довольно мощная, но создает определенные сложности.

4. *Зачем вызывать конструктор базового класса из конструктора производного класса?*

   Это нужно для того, чтобы управлять вызовом конструктора базового класса. Например, вам может понадобиться передать определенные значения в конструктор базового класса.

5. *Есть ли какие-то подводные камни, проявляющиеся при переопределении функций базового класса?*

   Да. Переопределение функции-члена базового класса скрывает все перегруженные версии функции базового класса. Однако вы всегда можете вызвать скрытые функции-члены базового класса явно, используя имя этого класса и оператор разрешения области действия.

6. *Как можно решить проблему сокрытия функций базового класса?*

   Одно из возможных решений — переопределить все перегруженные версии функции базового класса.

7. *Зачем нужно вызывать функцию-член оператора присваивания базового класса из функции-члена оператора присваивания производного класса?*

   Это позволяет корректно выполнить присваивание членов базового класса.

8. *Зачем нужно вызывать конструктор копирования базового класса из конструктора производного класса?*

   Это позволяет корректно выполнить копирование членов базового класса.

9. *Почему можно потерять доступ к функции-члену объекта, если указать на него с помощью указателя типа базового класса?*

   Потому что невиртуальные функции вызываются на основе типа указателя и типа объекта.

10. *Почему бы не сделать виртуальными все функции-члены на случай, если когда-нибудь понадобится использовать полиморфное поведение?*

    Потому что виртуальные функции предъявляют повышенные требования к производительности.

11. *В таком случае когда стоит объявлять функцию виртуальной?*

    Когда она может быть унаследована из базового класса.

12. *Когда стоит объявлять деструктор виртуальным?*

    Если в вашем классе имеются виртуальные функции, деструктор также следует объявить виртуальным. Однако некоторые программисты с целью повышения безопасности рекомендуют делать все деструкторы виртуальными.

13. *Могут ли конструкторы быть виртуальными?*

    Нет. Это также значит, что и конструкторы копирования не могут быть виртуальными.

14. *Что такое отсечение с точки зрения ООП?*

    Отсечение — это отрезание части объекта. Присваивание объекта производного класса переменной базового класса — это корректно, но объект придется обрезать так, чтобы он разместился в меньшем объеме памяти, что приведет к потере членов данных, объявленных в производном классе, а также потере доступа к функциям-членам производного класса.

15. *Что хорошего в абстрактных классах, если нельзя создавать их объекты?*

    Абстрактные классы могут быть очень полезными. Они могут содержать множество общих членов данных, которые унаследуют остальные классы, что позволит не объявлять их в производных классах снова и снова.

Вопросы для обсуждения

1. Какие преимущества дает наследование с точки зрения программирования игр?
2. Как полиморфизм расширяет возможности наследования?
3. Какие типы игровых сущностей имеет смысл моделировать с помощью наследования?
4. Какие типы игровых классов можно реализовать как абстрактные?
5. Почему удобно иметь возможность указать на объект производного класса с помощью указателя на объект типа базового класса?

Упражнения

1. Улучшите программу Simple Boss 2.0, добавив в нее новый класс, `FinalBoss`, который наследует от класса `Boss`. В классе `FinalBoss` должен быть определен новый метод `MegaAttack()`, который наносит в десять раз больший урон, чем метод `SpecialAttack()`.
2. Улучшите игру Blackjack, заставив ее формировать колоду заново, если перед коном в ней осталось мало карт.
3. Улучшите программу Abstract Creature, добавив в нее новый класс `OrcBoss`, который наследует от класса `Orc`. Объект класса `AnOrcBoss` должен иметь значение здоровья, равное 180. Также необходимо переопределить виртуальную функцию-член `Greet()`, чтобы она отображала следующее сообщение: `The orc boss growls hello.`

Приложение 1
Создание первой программы на языке C++

Следуйте этим шагам, чтобы написать, скомпилировать и запустить свою первую программу на языке C++ с помощью Microsoft Visual Studio Express 2013 for Windows Desktop — популярной бесплатной интегрированной среды разработки (Integrated Development Environment, IDE), предназначенной для платформы Windows.

1. Загрузите среду разработки Visual Studio Express 2013 for Windows Desktop с сайта **www.visualstudio.com/downloads/download-visual-studio-vs**.

ПОДСКАЗКА

Убедитесь, что загружаете среду разработки Visual Studio Express 2013 for Windows Desktop, а не Visual Studio Express 2013 for Windows — это разные продукты.

2. Установите среду разработки Visual Studio Express 2013 for Windows Desktop, приняв все параметры по умолчанию.
3. Запустите среду разработки Visual Studio Express 2013 for Windows Desktop. Вы должны увидеть диалоговое окно **Welcome**, показанное на рис. П1.1.
4. Создайте профиль и авторизуйтесь. Приложение откроется, и вы увидите стартовую страницу (рис. П1.2).
5. В меню **Application** выберите пункты **File ▸ New Project**. На левой панели появившегося диалогового меню **New Project** выберите **Visual C++**. На средней панели выберите пункт **Win32 Console Application**. В поле **Name** введите `game_over`. В поле **Location** выберите место, куда следует сохранить ваш проект, нажав кнопку **Browse**. (Я храню свой проект по адресу **C:\Users\Mike\Desktop**.) Наконец, убедитесь, что установлен флажок **Create directory for solution**. Диалоговое меню **New Project** должно выглядеть так, как показано на рис. П1.3.
6. Когда вы заполните диалоговое меню **New Project**, нажмите кнопку **OK**. Это приведет к появлению диалогового окна **Win32 Application Wizard — Overview**. Нажмите кнопку **Next**. Это приведет к появлению диалогового окна **Win32 Application Wizard — Application Settings**. В разделе **Additional options** установите флажок **Empty project**. Экран должен выглядеть так, как показано на рис. П1.4.

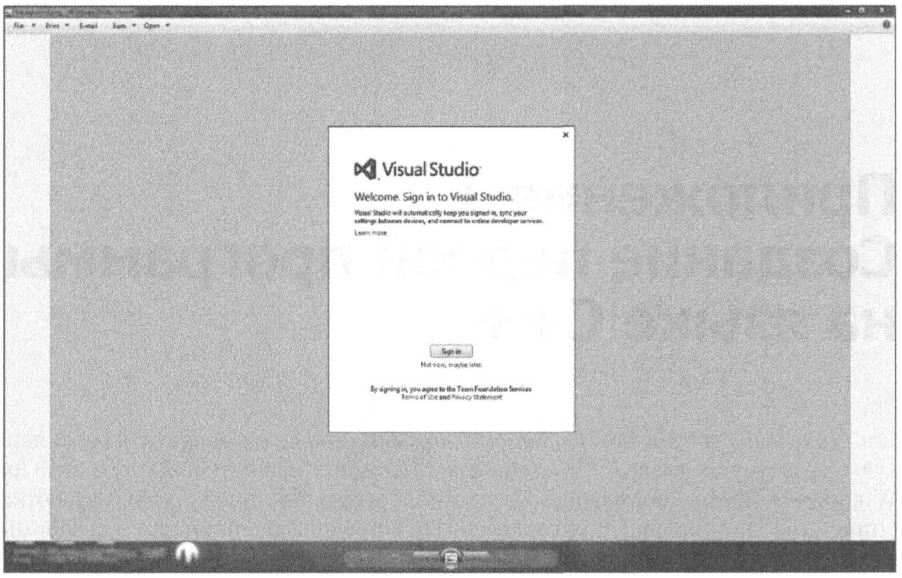


Рис. П1.1. Диалоговое окно Welcome среды разработки The Visual Studio Express 2013 приглашает вас авторизоваться (опубликовано с разрешения компании Microsoft)

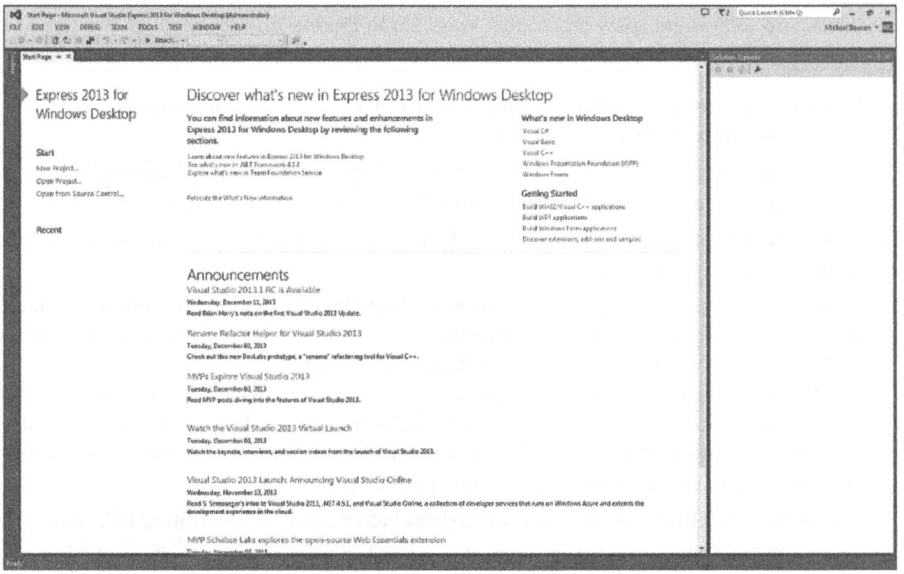


Рис. П1.2. Среда разработки Visual Studio Express 2013 после запуска (опубликовано с разрешения компании Microsoft)

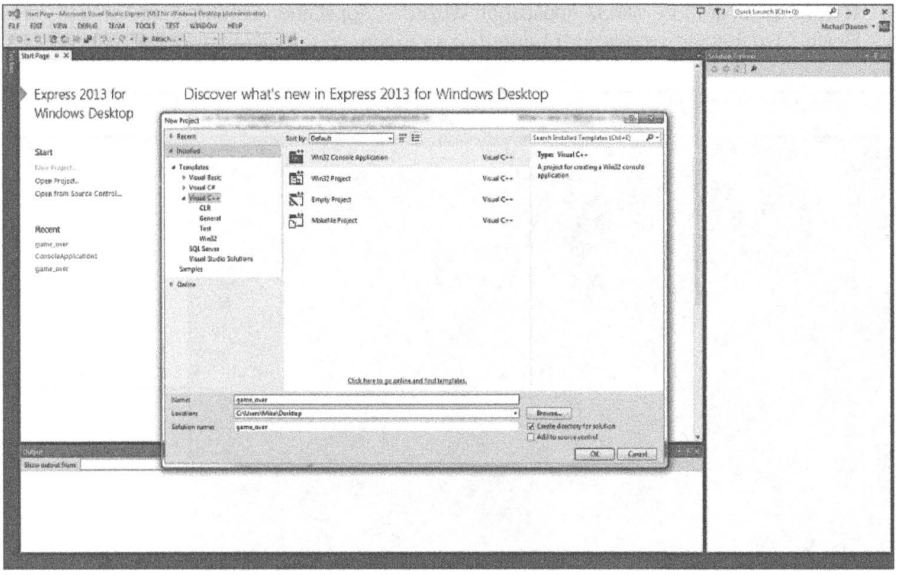


Рис. П1.3. Заполненное диалоговое меню New Project (опубликовано с разрешения компании Microsoft)

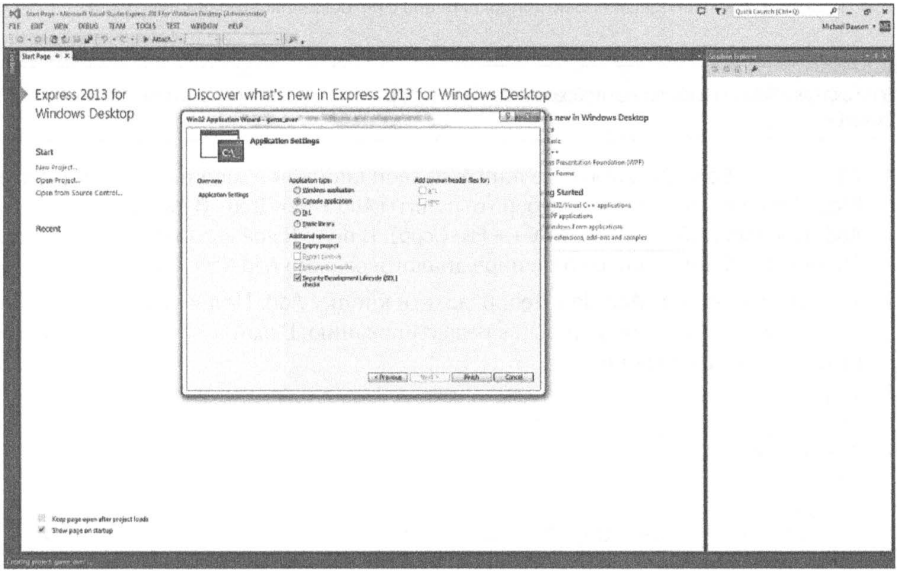


Рис. П1.4. Диалоговое окно Win32 Application Wizard — Application Settings, создание пустого проекта (опубликовано с разрешения компании Microsoft)

7. В диалоговом окне **Win32 Application Wizard — Application Settings** нажмите кнопку **Finish**. Это создаст и откроет решение для вашего проекта (рис. П1.5).

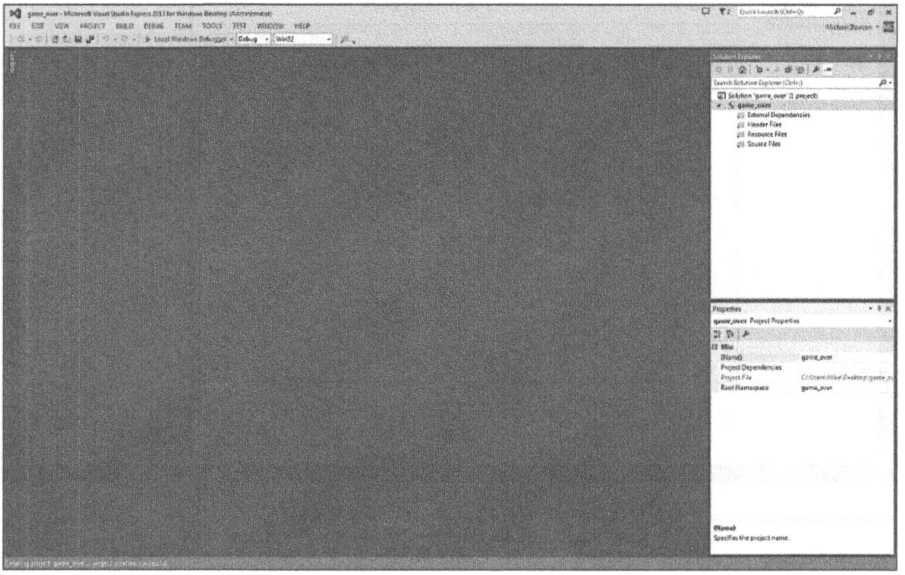


Рис. П1.5. Ваш новый проект (опубликовано с разрешения компании Microsoft)

ПОДСКАЗКА

Если вкладка Solution Explorer не отображается, в меню Application следует выбрать пункт View ▸ Solution Explorer.

8. На вкладке **Solution Explorer** щелкните правой кнопкой мыши на каталоге **Source Files**. В появившемся меню выберите пункты **Add ▸ New Item**. В диалоговом меню **Add New Item** выберите пункт **C++ File (.cpp)**. В поле **Name** введите `game_over.cpp`. На рис. П1.6 показано заполненное диалоговое окно **Add New Item**.
9. В диалоговом окне **Add New Item** нажмите кнопку **Add**. Появится пустой файл с именем `game_over.cpp`, готовый к редактированию. В файле `game_over.cpp` наберите следующие строки:

```
// Game Over
// Первая программа на языке C++
#include <iostream>
int main()
{
    std::cout << "Game Over!" << std::endl;
    return 0;
}
```

Экран должен выглядеть так, как показано на рис. П1.7.

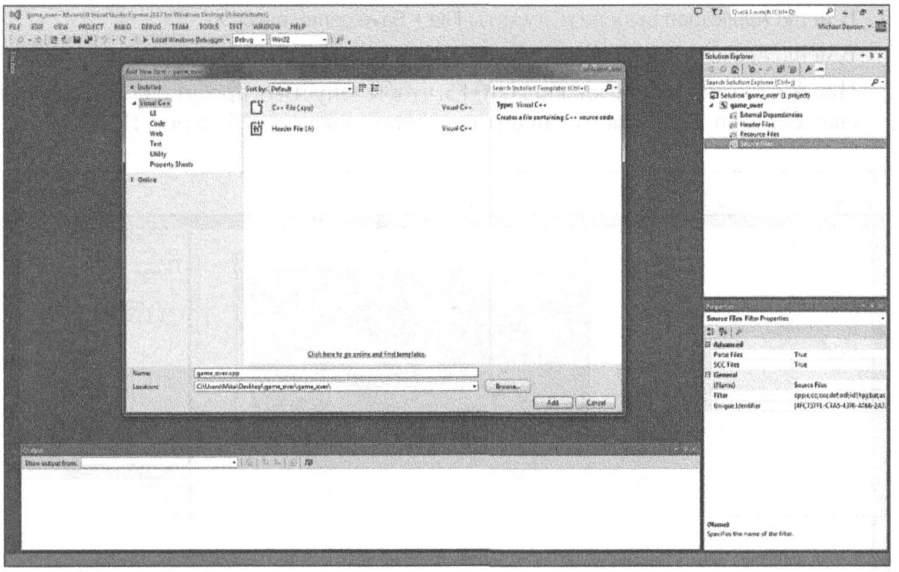


Рис. П1.6. Заполненное диалоговое окно Add New Item
(опубликовано с разрешения компании Microsoft)

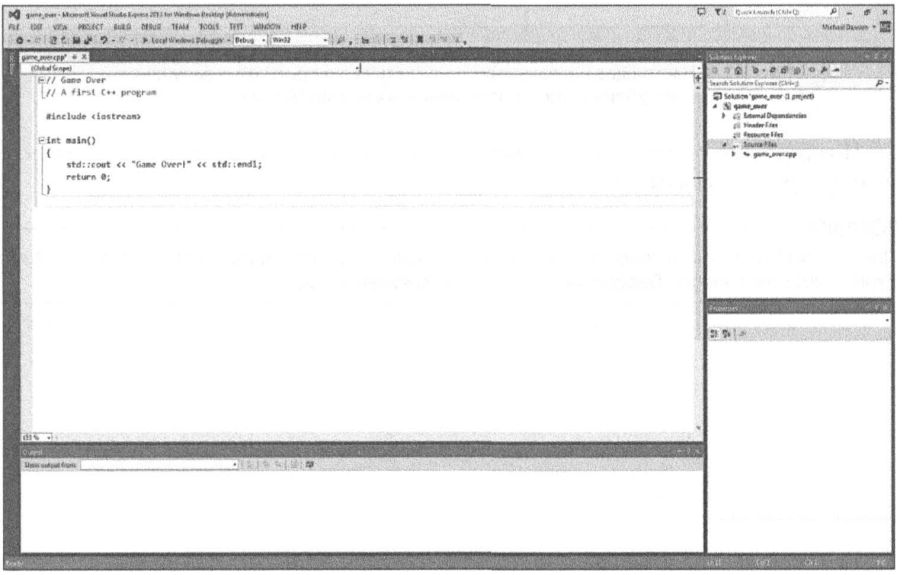


Рис. П1.7. Новый файл со свежими изменениями
(опубликовано с разрешения компании Microsoft)

10. В меню **Application** выберите пункты **File ▸ Save game_over.cpp**.
11. В меню **Application** выберите пункты **Build ▸ Build Solution**.
12. Нажмите комбинацию клавиш **Ctrl+F5**, чтобы запустить проект и пожать плоды своих трудов. Вы должны увидеть результат, показанный на рис. П1.8.

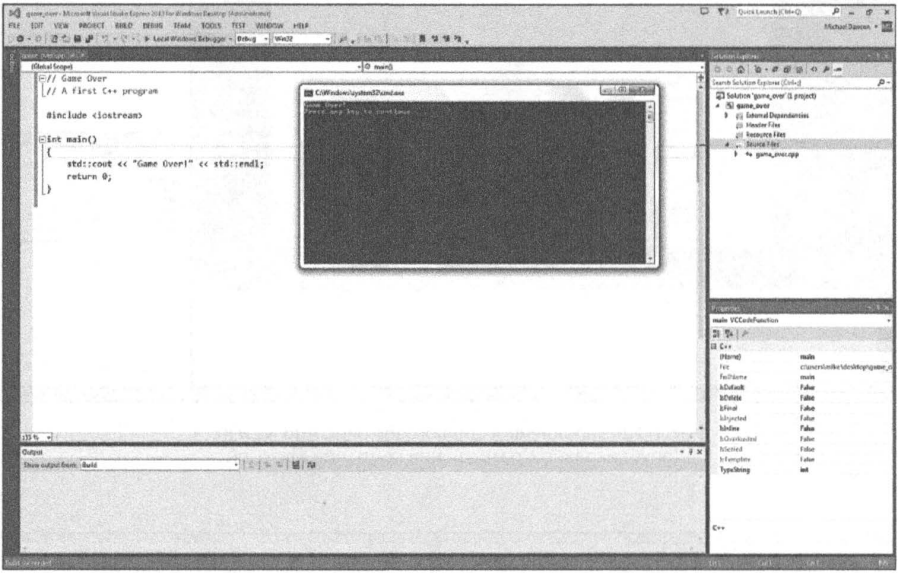


Рис. П1.8. Настоящее наслаждение — видеть свою программу запущенной (опубликовано с разрешения компании Microsoft)

Поздравляю! Вы написали, сохранили, скомпилировали и запустили свою первую программу на языке C++.

ПОДСКАЗКА

Для того чтобы получить более детальную информацию о среде разработки Microsoft Visual Studio Express 2013 for Windows Desktop, обратитесь к ее документации.

Приложение 2
Приоритет операторов языка C++

Уровень приоритета	Оператор	Описание
17	::	Разрешение области действия
16	->	Опосредованный выбор члена
16	.	Выбор члена
16	[]	Индекс массива
16	()	Вызов функции
16	()	Создание типа
16	sizeof	Размер в байтах
15	++	Постфиксный инкремент
15	—	Постфиксный декремент
15	~	Битовое «НЕ»
15	!	Логическое «НЕ»
15	+	Унарный плюс
15	–	Унарный минус
15	*	Разыменование
15	&	Взятие адреса
15	()	Преобразование
15	new	Получение памяти кучи
15	delete	Освобождение памяти кучи
15	++	Префиксный инкремент
15	—	Префиксный декремент
14	–>*	Опосредованный оператор указателя на член
14	.*	Оператор указателя на член
13	*	Умножение

Продолжение ⇩

(продолжение)

Уровень приоритета	Оператор	Описание
13	/	Деление
13	%	Остаток от целочисленного деления
12	+	Сложение
12	-	Вычитание
11	<<	Побитовый сдвиг влево
11	>>	Побитовый сдвиг вправо
10	<	Меньше
10	<=	Меньше или равно
10	>	Больше
10	>=	Больше или равно
9	==	Равенство
9	!=	Неравенство
8	&	Битовое «И»
7	^	Битовое «ИСКЛЮЧАЮЩЕЕ ИЛИ»
6	|	Битовое «ИЛИ»
5	&&	Логическое «И»
4	||	Логическое «ИЛИ»
3	?:	Условный оператор
2	=	Присваивание
2	*=	Умножение и присваивание
2	/=	Деление и присваивание
2	%=	Целочисленное деление и присваивание
2	+=	Сложение и присваивание
2	−=	Вычитание и присваивание
2	<<=	Битовый сдвиг влево и присваивание
2	>>=	Битовый сдвиг вправо и присваивание
2	&=	Битовое «И» и присваивание
2	|=	Битовое «ИЛИ» и присваивание
2	^=	Битовое «ИСКЛЮЧАЮЩЕЕ ИЛИ» и присваивание
1	,	Запятая

Приложение 3
Ключевые слова языка C++

alignas	char32_t	enum
alignof	class	explicit
and	compl	export
and_eq	const	extern
asm	constexpr	false
auto	const_cast	float
bitand	continue	for
bitor	decltype	friend
bool	default	goto
break	delete	if
case	do	inline
catch	double	int
char	dynamic_cast	long
char16_t	else	mutable
namespace	return	try
new	short	typedef
noexcept	signed	typeid
not	sizeof	typename
not_eq	static	union
nullptr	static_assert	unsigned
operator	static_cast	using
or	struct	virtual
or_eq	switch	void
private	template	volatile
protected	this	wchar_t
public	thread_local	while
register	throw	xor
reinterpret_cast	true	xor_eq

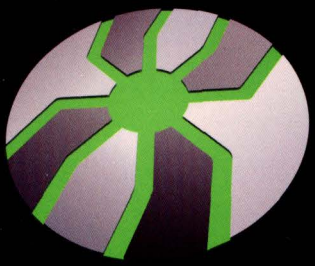
Приложение 4
Таблица символов ASCII

Десятичное представление	Шестнадцатеричное представление	Символ	Десятичное представление	Шестнадцатеричное представление	Символ
0	00	NUL	27	1B	ESC
1	01	SOH	28	1C	FS
2	02	STX	29	1D	GS
3	03	ETX	30	1E	RS
4	04	EOT	31	1F	US
5	05	ENQ	32	20	SP
6	06	ACK	33	21	!
7	07	BEL	34	22	"
8	08	BS	35	23	#
9	09	HT	36	24	$
10	0A	LF	37	25	%
11	0B	VT	38	26	&
12	0C	FF	39	27	'
13	0D	CR	40	28	(
14	0E	SO	41	29	)
15	0F	SI	42	2A	*
16	10	DLE	43	2B	+
17	11	DC1	44	2C	,
18	12	DC2	45	2D	-
19	13	DC3	46	2E	
20	14	DC4	47	2F	/
21	15	NAK	48	30	0
22	16	SYM	49	31	1
23	17	ETD	50	32	2
24	18	CAN	51	33	3
25	19	EM	52	34	4
26	1A	SUB	53	35	5

Десятичное представление	Шестнадцатеричное представление	Символ	Десятичное представление	Шестнадцатеричное представление	Символ
54	36	6	91	5B	[
55	37	7	92	5C	\
56	38	8	93	5D	]
57	39	9	94	5E	^
58	3A	:	95	5F	_
59	3B	;	96	60	`
60	3C	<	97	61	a
61	3D	=	98	62	b
62	3E	>	99	63	c
63	3F	?	100	64	d
64	40	@	101	65	e
65	41	A	102	66	f
66	42	B	103	67	g
67	43	C	104	68	h
68	44	D	105	69	i
69	45	E	106	6A	j
70	46	F	107	6B	k
71	47	G	108	6C	l
72	48	H	109	6D	m
73	49	I	110	6E	n
74	4A	J	111	6F	o
75	4B	K	112	70	p
76	4C	L	113	71	q
77	4D	M	114	72	r
78	4E	N	115	73	s
79	4F	O	116	74	t
80	50	P	117	75	u
81	51	Q	118	76	v
82	52	R	119	77	w
83	53	S	120	78	x
84	54	T	121	79	y
85	55	U	122	7A	z
86	56	V	123	7B	{
87	57	W	124	7C	|
88	58	X	125	7D	}
89	59	Y	126	7E	~
90	5A	Z	127	7F	DEL

Приложение 5
Управляющие последовательности

Управляющая последовательность	Описание
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратный слеш
\0	Пустой символ
\a	Звуковой сигнал
\b	Возврат на шаг
\f	Прогон страницы
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\x	Шестнадцатеричное число



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ АНТИВИРУСНАЯ ЛАБОРАТОРИЯ ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные программные продукты

Если вы хотите научиться программировать первоклассные игры, вам просто необходимо изучить язык C++. Эта книга поможет вам освоить разработку игр с самых азов, независимо от того, есть ли у вас опыт программирования. Гораздо интересней учиться, когда обучение превращается в игру.

Каждая глава книги описывает самостоятельный игровой проект. В заключительной главе вам предстоит написать сложную игру, которая объединяет все приемы программирования, рассмотренные в предыдущих главах.

Книга, которую вы держите в руках, идеально подойдет для начинающего программиста, планирующего не только как следует освоить непростой язык C++, но и поупражняться в программировании игр.

Тема: Разработка игр	Уровень читателя: начинающий

ПИТЕР®

CENGAGE Learning®

Заказ книг:
Санкт-Петербург
тел.: (812) 703-73-74, postbook@piter.com

ISBN:978-5-496-01629-2

9 785496 016292

www.piter.com — каталог книг и интернет-магазин