Herbert Schildt
C++ from the Ground Up Third Edition

Герберт Шилдт: C++ базовый курс



В этой книге описаны все основные средства языка C++ - от элементарных понятий до супервозможностей. После рассмотрения основ программирования на C++ (переменных, операторов, инструкций управления, функций, классов и объектов) читатель освоит такие более сложные средства языка, как механизм обработки исключительных ситуаций (исключений), шаблоны, пространства имен, динамическая идентификация типов, стандартная библиотека шаблонов (STL), а также познакомится с расширенным набором ключевых слов, используемых в .NET-программировании. Автор справочника - общепризнанный авторитет в области программирования на языках C и C++, Java и C# - включил в текст своей книги и советы программистам, которые позволят повысить эффективность их работы.

Книга рассчитана на широкий круг читателей, желающих изучить язык программирования C++.

Примеры программ работают со всеми компиляторами C++, включая Visual C++

Оглавление

Глава 1. Из истории создания C++

Глава 2. Обзор элементов языка C++

Глава 3. Основные типы данных

Глава 4. Инструкции управления

Глава 5. Массивы и строки

Глава 6. Указатели

Глава 7. Функции, часть первая: основы

Глава 8. Функции, часть вторая: ссылки, перегрузка и использование аргументов по умолчанию

Глава 9. Еще о типах данных и операторах

Глава 10. Структуры и объединения

Глава 11. Введение в классы

Глава 12. О классах подробнее

Глава 13. Перегрузка операторов

Глава 14. Наследование

Глава 15. Виртуальные функции и полиморфизм

Глава 16. Шаблоны

Глава 17. Обработка исключительных ситуаций

Глава 18. C++ - система ввода-вывода

Глава 19. Динамическая идентификация типов и операторы приведения типа

Глава 20. Пространства имен и другие темы

Глава 21. Введение в стандартную библиотеку шаблонов

Глава 22. Препроцессор C++

Приложение А. С-ориентнрованная система ввода-вывода

Приложение Б. Использование устаревшего C++-компилятора

Приложение В. .NET-расширения для C++

Предметный указатель

Об авторе

Герберт Шилдт (Herbert Schildt) — признанный авторитет в области программирования на языках C, C++ Java и C#, профессиональный Windows-программист, член комитетов ANSI/ISO, принимавших стандарт для языков C и C++. Продано свыше 3 миллионов экземпляров его книг. Они переведены на все самые распространенные языки мира. Шилдт — автор таких бестселлеров, как *Полный справочник по C*, *Полный справочник по C++*, *Полный справочник по C#*, *Полный справочник по Java 2*, и многих других книг, включая: *Руководство для начинающих по C++*, *C#: A Beginner's Guide* и *Java 2: A Beginner's Guide*. Шилдт— обладатель степени магистра в области вычислительной техники (университет шт. Иллинойс). Его контактный телефон (в консультационном отделе): (217) 586-4683.

Введение

Цель этой книги — научить писать программы на C++ — самом мощном языке программирования наших дней. Для освоения представленного здесь материала никакого предыдущего опыта в области программирования не требуется. Мы начнем с азов, знание которых позволит читателю осилить сначала фундаментальные понятия языка, а затем и его ядро. Изучив базовый курс, вы справитесь и с более сложными темами, освоение которых даст вам право считать себя вполне сложившимся программистом на C++.

Язык C++ — это ключ к современному объектно-ориентированному программированию. Он создан для разработки высокопроизводительного программного обеспечения и чрезвычайно популярен среди программистов. Сегодня быть профессиональным программистом высокого класса означает быть компетентным в C++.

Этот язык не просто популярен. Он обеспечивает концептуальный фундамент, на который опираются другие языки программирования и многие современные средства обработки данных. Не случайно ведь потомками C++ стали такие почитаемые языки, как C# и Java.

Поскольку язык C++ предназначен для профессионального программирования, для изучения он не самый простой; тем не менее, C++ — самый лучший язык для изучения. Освоив C++, вы сможете писать профессиональные высокопроизводительные программы. Кроме того, вы сможете легко изучить такие языки программирования, как C# и Java, поскольку они используют тот же базовый синтаксис и те же принципы разработки.

Что нового в третьем издании

За время, прошедшее с момента выхода предыдущего издания этой книги, язык C++ не претерпел никаких изменений. Однако изменилась вычислительная среда. Например, доминирующее положение в Web-программировании занял язык Java, появилась система .NET Framework и язык C#. Но мощь C++ за прошедшие несколько лет ничуть не убавилась. C++ был, есть и еще долго будет основным языком "классных" программистов.

Общая структура и организация третьего издания практически повторяют второе. Большинство изменений связано с обновлением текста или его дополнением. В одних случаях лучше раскрыта тема, а в других добавлено описание современной среды программирования. Книга расширена также за счет нескольких новых разделов.

Кроме того, добавлено два приложения. В одном описаны определенные компанией Microsoft ключевые слова, которые используются для создания управляемого кода, предназначенного для среды .NET Framework. В другом разъясняется, как адаптировать код, приведенный в этой книге, к более старым и нестандартным компиляторам.

Наконец, все примеры программ были протестированы с использованием таких компиляторов, как Visual Studio .Net (Microsoft) и C++ Builder (Borland).

О версии C++

Материал этой книги описывает *Standard C++*. Эта версия C++ определена Американским национальным институтом стандартов (American National Standards Institute — ANSI) и Международной организацией по стандартизации (International Standards Organization — ISO) в качестве стандарта для C++, который поддерживается практически всеми известными компиляторами. Поэтому, используя эту книгу, вы можете быть уверены в том, что освоенное вами сегодня непременно будет применено завтра.

Как работать с этой книгой

Изучайте любой язык программирования (в том числе и C++), программируя. Это — лучший способ. Поэтому, прочитав очередной раздел, закрепите материал на практике. Прежде чем переходить к следующему разделу, убедитесь в том, что вы понимаете, почему примеры программ делают то, что они делают. Полезно также экспериментировать с программами, изменяя одну или две строки и анализируя влияние этих изменений на результаты. Чем больше вы будете программировать, тем выше будет ваш уровень как программиста.

Если вы работаете под управлением Windows

Если вы работаете под управлением Windows

Если на вашем компьютере установлена система Windows, и ваша цель — создание Windows-ориентированных программ, вы сделали правильный выбор, решив изучать C++, поскольку C++ — это родной язык для Windows. Однако ни в одном из примеров этой книги не используется Windows-интерфейс GUI (graphical user interface — графический интерфейс пользователя). Все приведенные здесь примеры являются консольными, т.е. приложениями, запускаемыми из командной строки (поскольку не имеют графического интерфейса). Дело в том, что GUI-программы отличаются высокой сложностью и большим размером. Кроме того, они используют множество методов, которые напрямую не связаны с языком C++. Поэтому они не совсем подходят для обучения языку программирования. Однако это не мешает использовать Windows-ориентированный компилятор для компиляции программ из этой книги, поскольку он автоматически создаст консольный сеанс, позволяющий выполнить программу.

Освоив C++, вы сможете применить свои знания к Windows-программированию. Windows-программирование на основе C++ позволяет использовать библиотеки классов, например *MFC* или *.NET Framework*, которые значительно упрощают разработку Windows-программ.

Программный код — из Web-пространства

Помните, что исходный код всех программ, приведенных в этой книге, можно загрузить с Web-сайта с адресом: http://www.osborne.com. Загрузка кода избавит вас от необходимости вводить текст программ вручную.

Что еще почитать

Книга C++: базовый курс — это ваш "ключ" к серии книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены те из них, которые могут представлять для вас интерес.

Те, кто желает подробнее изучить язык C++, могут обратиться к следующим книгам.

- Полный справочник по C++
- Руководство для начинающих по C++
- Освой самостоятельно C++ за 21 день
- STL Programming from the Ground Up
- Справочник программиста по C/C++

Тем, кого интересует программирование на языке Java, мы рекомендуем такие издания.

- Java 2: A Beginner's Guide
- Полный справочник по Java 2
- Java 2: Programmer's Reference

Если вы желаете научиться программировать на C#, обратитесь к следующим книгам.

- C#: A Beginner's Guide
- Полный справочник по C#

Тем, кто интересуется Windows-программированием, мы можем предложить такие книги Шилдта.

- Windows 98 Programming from the Ground Up
- Windows 2000 Programming from the Ground Up
- MFC Programming from the Ground Up
- The Windows Annotated Archives

Если вы хотите поближе познакомиться с языком C, который является фундаментом всех современных языков программирования, обратитесь к следующим книгам.

- Полный справочник по C
- Освой самостоятельно C за 21 день

■ Освой самостоятельно C за 21 день

Если вам нужны четкие ответы, обращайтесь к Герберту Шилдту, общепризнанному авторитету в области программирования.
Ждем ваших отзывов!

Вы, уважаемый читатель, и есть главный критик и комментатор этой книги. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

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

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: http: //www.williamspublishing. com

Адреса для писем:

из России: 115419, Москва, а/я 783

из Украины: 03150, Киев, а/я 152

от редактора FB2

Так как эта электронная книга будет отображаться на устройствах с разными размерами экрана, то расположение текста книги будет разным. Так же в книге используется форматирование кода (сдвиг вправо пробелами), а в некоторых читалках пять пробелов отображается как один (AlReader2). Поэтому я сделал пробел(сдвиг) в виде юникода если ваше устройство и/или читалка поддерживают юникод, то у вас будет отображаться пробел в коде, если неподдерживают то будет отображаться квадрат (вместо него в программе компиляторе пишите пару пробелов, или один TAB)

Пример:

<<<тут должно быть 3 квадрата... если их нет то код будет сдвинут вправо в нужных местах (ваше устройство поддерживает юникод). Если вы их видите то в местах сдвига (там где они отображаются) пишите пару пробелов (или игнорируйте их)

Так же обращайте внимание на построчный коментрарий: **//** на маленьком экране он может разбиться на 2 строки!

```
код программы //коментарий
```

коментарий может разбиться на две строки

1-я строка-**ком**
2-я строка-**ентарий**
в этом случае компилятор будет ругаться на команду **ентарий** в компиляторе он будет отображаться другим цветом нежели
//ком
в итоге программа небудет запускаться. Лечится это перемещением **ентарий** на строку **// ком**
-> 1-я строка **//коментарий**

Настройка отображения кода: (для того чтобы было удобее читать код программ)

AlReader2:

меню=> настройки=> стили текста=> код=> настройте: выравнивание=*К левому краю*, цвет=

(выберите цвет), отступ слева=*нет отступа*, отступ справа=*нет отступа*, отступ начала абзаца= *(убрать галочку)*, остальные настройки на ваш выбор. В других программах настройки аналогичны...

Так как каждый компилятор и среда разработки "по своймy нарушает стандарт C++" (что-то убирает или добавляет), а так же из за развития C++ с момента его создания, некоторые программы этой книги нужно писать по другому. По этой причине я привожу список сайтов на которых вы сможете найти ответы на интересующие вас вопросы:

http://codenet.ru/

http://hashcode.ru

http://rsdn.ru/

http://ci-plus-plus.blogspot.com/

http://programmersclub.ru/

http://cyberforum.ru/

Глава 1: Из истории создания C++

Язык C++ — единственный (из самых значительных) язык программирования, который может освоить любой программист. Это может показаться очень серьезным заявлением, но оно — не преувеличение. C++ — это центр притяжения, вокруг которого "вращается" всё современное программирование. Его синтаксис и принципы разработки определяют суть объектно-ориентированного программирования. Более того, C++ проложил "лыжню" для разработки языков будущего. Например, как Java, так и C# — прямые потомки языка C++. C++ также можно назвать универсальным языком программирования, поскольку он позволяет программистам обмениваться идеями. Сегодня быть профессиональным программистом высокого класса означает быть компетентным в C++. C++ — это ключ к современному программированию.

Приступая к изучению C++, важно знать, как он вписывается в исторический контекст языков программирования. Понимая, что привело к его созданию, какие принципы разработки он представляет и что он унаследовал от своих предшественников, вам будет легче оценить суть новаторства и уникальность средств C++. Именно поэтому в данной главе вам предлагается сделать краткий экскурс в историю создания языка программирования C++, заглянуть в его истоки, проанализировать его взаимоотношения с непосредственным предшественником (C), рассмотреть его возможности (области применения) и принципы программирования, которые он поддерживает. Здесь также вы узнаете, какое место занимает C++ среди других языков программирования.

Истоки C++

История создания C++ начинается с языка C. И немудрено: C++ построен на фундаменте C. C++ и в самом деле представляет собой супермножество языка C. (Все компиляторы C++ можно использовать для компиляции C-программ.) C++ можно назвать расширенной и улучшенной версией языка C, в которой реализованы принципы объектно-ориентированного программирования. C++ также включает ряд других усовершенствований языка C, например расширенный набор библиотечных функций. При этом "вкус и запах" C++ унаследовал непосредственно из языка C. Чтобы до конца понять и оценить достоинства C++, необходимо понять все "как" и "почему" в отношении языка C.

Создание языка C

Появление языка C потрясло компьютерный мир. Его влияние нельзя было переоценить, поскольку он коренным образом изменил подход к программированию и его восприятие. Язык C стал считаться первым современным "языком программиста", поскольку до его изобретения компьютерные языки в основном разрабатывались либо как учебные упражнения, либо как результат деятельности бюрократических структур. С языком C все обстояло иначе. Он был задуман и разработан реальными, практикующими программистами и отражал их подход к программированию. Его средства были многократно обдуманы, отточены и протестированы людьми, которые действительно работали с этим языком. В результате этого процесса появился язык, который понравился многим программистам-практикам.

Язык C быстро завоевал умы и сердца многочисленных приверженцев, у которых возникло к новому языку почти религиозное чувство, что способствовало быстрому и широкому его распространению в сообществе программистов. Короче говоря, C — это язык, который разработан программистами для программистов. Именно это и обусловило его бешеный успех.

Язык C изобрел Дэнис Ритчи (Dennis Ritchie) для компьютера PDP-11 (разработка компании DEC — Digital Equipment Corporation), который работал под управлением операционной системы (ОС) UNIX. Язык C — это результат процесса разработки, который сначала был связан с другим языком — BCPL, созданным Мартином Ричардсом (Martin Richards). Язык BCPL индуцировал появление языка, получившего название B (его автор — Кен Томпсон (Ken Thompson)), который в свою очередь привел к разработке языка C. Это случилось в начале 70-х годов.

На протяжении многих лет стандартом для языка C де-факто служил язык, поддерживаемый ОС UNIX и описанный в книге Брайана Кернигана (Brian Kernighan) и Дэниса Ритчи The C Programming Language (Prentice-Hall, 1978). Однако формальное отсутствие стандарта стало причиной расхождений между различными реализациями языка C. Чтобы изменить ситуацию, в начале лета 1983 г. был учрежден комитет по созданию ANSI-стандарта, призванного — раз и навсегда — определить язык C. Конечная версия этого стандарта была принята в декабре 1989, а его первая копия стала доступной для желающих в начале 1900. Эта версия языка C получила название C89, и именно она явилась фундаментом, на котором был построен язык C++.

На заметку: *Стандарт С был обновлен в 1999 году, и эта версия языка С получила название С99. Новый стандарт содержит ряд новых средств, причем некоторые из них позаимствованы из С++, тем не менее они полностью совместимы с оригинальным стандартом С89. Насколько мне известно, на данный момент ни один из широко доступных компиляторов не поддерживает версию С99, и по-прежнему версия С89 определяет то, что обычно подразумевается под языком С. Более того, именно стандарт С89 послужил основой для создания языка С++. Вполне возможно, что будущий стандарт языка С++ будет включать средства, добавленные версией С99, но пока они не являются частью С++.*

Язык С часто называют компьютерным языком "среднего уровня". Применительно к С это определение не имеет негативного оттенка, поскольку оно отнюдь не означает, что язык С менее мощный и развитый (по сравнению с языками "высокого уровня") или его сложно использовать (подобно ассемблеру). (Для *языка ассемблера* характерно символическое представление реального машинного кода, который может выполнять компьютер.) С называют языком среднего уровня, поскольку он сочетает элементы языков высокого уровня (например Pascal, Modula-2 или Visual Basic) с функциональностью ассемблера.

С точки зрения теории в *язык высокого уровня* заложено стремление дать программисту все, что он может захотеть, в виде встроенных средств. *Язык низкого уровня* не обеспечивает программиста ничем, кроме доступа к реальным машинным командам. *Язык среднего уровня* предоставляет программистам некоторый (небольшой) набор инструментов, позволяя им самим разрабатывать конструкции более высокого уровня. Другими словами, язык среднего уровня предлагает программисту встроенную мощь в сочетании с гибкостью.

Будучи языком среднего уровня, С позволяет манипулировать битами, байтами и адресами, т.е. базовыми элементами, с которыми работает компьютер. Таким образом, в С не предусмотрена попытка отделить аппаратные средства компьютера от программных. Например, размер целочисленного значения в С напрямую связан с размером слова центрального процессора (ЦП). В большинстве языков высокого уровня существуют встроенные инструкции, предназначенные для чтения и записи дисковых файлов. В языке С все эти процедуры выполняются посредством вызова библиотечных функций, а не с помощью ключевых слов, определенных в самом языке. Такой подход повышает гибкость языка С.

Язык С позволяет программисту (правильнее сказать, стимулирует его) определять подпрограммы для выполнения операций высокого уровня. Эти подпрограммы называются *функциями*. Функции имеют очень большое значение для языка С. Их можно назвать строительными блоками С. Программист может без особых усилий создать библиотеку функций, предназначенную для выполнения различных задач, которые используются его программой. В этом смысле программист может персонализировать С в соответствии со своими потребностями.

Необходимо упомянуть еще об одном аспекте языка С, который очень важен для С++: С — *структурированный язык*. Самой характерной особенностью структурированного языка является использование блоков. *Блок* — это набор инструкций, которые логически связаны между собой. Например, представьте себе инструкцию *IF*, которая при успешной проверке своего выражения должна выполнить пять отдельных инструкций. А если эти инструкции (специальным образом) сгруппировать и обращаться к ним как к единому целому, то такая группа образует блок.

Структурированный язык поддерживает концепцию подпрограмм и локальных переменных. Локальная переменная — это обычная переменная, которая известна только подпрограмме, в которой она определена. Структурированный язык также поддерживает ряд таких циклических конструкций, как *while*, *do-while* и *for*. Однако использование инструкции *goto* либо запрещается, либо не рекомендуется, и не несет в себе той смысловой нагрузки для передачи управления, какая присуща ей в таких языках программирования, как *BASIC* или *FORTRAN*. Структурированный язык позволяет структурировать текст программы, т.е. делать отступы при написании инструкций, и не требует строгой привязки к полям, как это реализовано в ранних версиях языка *FORTRAN*.

Наконец (и это, возможно, самое важное), язык С не несет ответственности за действия программиста. Основной принцип С состоит в том, чтобы позволить программисту делать все, что он хочет, но за последствия, т.е. за все, что делает программа (пусть даже очень необычное, что-то из ряда вон или даже подозрительное), отвечает не язык, а программист. Язык С предоставляет программисту практически полную власть над компьютером, но эта власть ложится на его плечи тяжким бременем ответственности.

Предпосылки возникновения языка С++

Приведенная выше характеристика языка С может вызвать справедливое недоумение: зачем же тогда, мол, был изобретен язык С++? Если С — такой хороший и полезный язык, то почему возникла необходимость в чем-то еще? Оказывается, все дело в сложности. На протяжении всей истории программирования усложнение программ заставляло программистов искать пути, которые бы позволили справиться со сложностью. С++ можно считать одним из способов ее преодоления. Попробуем лучше раскрыть эту взаимосвязь.

Отношение к программированию резко изменилось с момента изобретения компьютера. Основная причина — стремление "укротить" всевозрастающую сложность программ. Например, программирование для первых вычислительных машин заключалось в переключении тумблеров на их передней панели таким образом, чтобы их положение соответствовало двоичным кодам машинных команд. Пока длины программ не превышали нескольких сотен команд, такой метод еще имел право на существование. Но по мере их дальнейшего роста был изобретен язык ассемблер, чтобы программисты могли использовать символическое представление машинных команд. Поскольку программы продолжали расти в размерах, желание справиться с более высоким уровнем сложности вызвало появление языков высокого уровня, разработка которых дала программистам больше инструментов (новых и разных).

Первым широко распространенным языком программирования был, конечно же, *FORTRAN*. Несмотря на то что это был очень значительный шаг на пути прогресса в области программирования, *FORTRAN* все же трудно назвать языком, который способствовал написанию ясных и простых для понимания программ. Шестидесятые годы двадцатого столетия считаются периодом появления структурированного программирования. Именно такой метод программирования и был реализован в языке С. С помощью структурированных языков программирования можно было писать программы средней сложности, причем без особых героических усилий со стороны программиста. Но если программный проект достигал определенного размера, то даже с использованием упомянутых структурированных методов его сложность резко возрастала и оказывалась непреодолимой для возможностей программиста. Когда (к концу 70-х) к "критической" точке подошло довольно много проектов, стали рождаться новые технологии программирования. Одна из них получила название *объектно-ориентированного программирования* (ООП). Вооружившись методами ООП, программист мог справляться с программами гораздо большего размера, чем прежде. Но язык С не поддерживал методов ООП. Стремление получить объектно-ориентированную версию языка С в конце концов и привело к созданию С++.

Несмотря на то что язык C был одним из самых любимых и распространенных профессиональных языков программирования, настало время, когда его возможности по написанию сложных программ достигли своего предела. Желание преодолеть этот барьер и помочь программисту легко справляться с еще более сложными программами — вот что стало основной причиной создания C++.

Рождение C++

Итак, C++ появился как ответ на необходимость преодолеть еще большую сложность программ. Он был создан Бьерном Страуструпом (Bjarne Stroustrup) в 1979 году в компании Bell Laboratories (г. Муррей-Хилл, шт. Нью-Джерси). Сначала новый язык получил имя "C с классами" (C with Classes), но в 1983 году он стал называться C++.

C++ полностью включает язык C. Как упоминалось выше, C — это фундамент, на котором был построен C++. Язык C++ содержит все средства и атрибуты C и обладает всеми его достоинствами. Для него также остается в силе принцип C, согласно которому программист, а не язык, несет ответственность за результаты работы своей программы. Именно этот момент позволяет понять, что изобретение C++ не было попыткой создать новый язык программирования. Это было скорее усовершенствование уже существующего (и при этом весьма успешного) языка.

Большинство новшеств, которыми Страуструп обогатил язык C, было предназначено для поддержки объектно-ориентированного программирования. По сути, C++ стал объектно-ориентированной версией языка C. Взяв язык C за основу, Страуструп подготовил плавный переход к ООП. Теперь, вместо того, чтобы изучать совершенно новый язык, C-программисту достаточно было освоить только ряд новых средств, и он мог пожинать плоды использования объектно-ориентированной технологии программирования.

Однако в основу C++ лег не только язык C. Страуструп утверждает, что некоторые объектно-ориентированные средства были инспирированы другим объектно-ориентированным языком, а именно Simula67. Таким образом, C++ представляет собой симбиоз двух мощных методологий программирования.

Создавая C++, Страуструп понимал, насколько важно, сохранить изначальную суть языка C, т.е. его эффективность, гибкость и принципы разработки, внести в него поддержку объектно-ориентированного программирования. К счастью, эта цель была достигнута. C++ по-прежнему предоставляет программисту свободу действий и власть над компьютером (которые были присущи языку C), расширяя при этом его (программиста) возможности за счет использования объектов.

Несмотря на то что C++ изначально был нацелен на поддержку очень больших программ, этим, конечно же, его использование не ограничивалось. И в самом деле, объектно-ориентированные средства C++ можно эффективно применять практически к любой задаче программирования. Неудивительно, что C++ используется для создания компиляторов, редакторов, компьютерных игр и программ сетевого обслуживания. Поскольку C++ обладает эффективностью языка C, то программное обеспечение многих высокоэффективных систем построено с использованием C++. Кроме того, C++ — это язык, который чаще всего выбирается для Windows-программирования.

Важно также помнить следующее. Поскольку C++ является супермножеством языка C, то, научившись программировать на C++, вы сможете также программировать и на C! Таким образом, приложив усилия к изучению только одного языка программирования, вы в действительности изучите сразу два.

Эволюция C++

С момента изобретения C++ претерпел три крупных переработки, причем каждый раз язык как дополнялся новыми средствами, так и в чем-то изменялся. Первой ревизии он был подвергнут в 1985 году, а второй — в 1990. Третья ревизия имела место в процессе стандартизации, который активизировался в начале 1990-х. Специально для этого был сформирован объединенный ANSI/ISO-комитет (я был его членом), который 25 января 1994 года принял первый проект предложенного на рассмотрение стандарта. В этот проект были включены все средства, впервые определенные Страуструпом, и добавлены новые. Но в целом он отражал состояние C++ на тот момент времени.

Вскоре после завершения работы над первым проектом стандарта C++ произошло событие, которое заставило значительно расширить существующий стандарт. Речь идет о создании Александром Степановым стандартной библиотеки шаблонов (Standard Template Library — STL). Как вы узнаете позже, STL — это набор обобщенных функций, которые можно использовать для обработки данных. Он довольно большой по размеру. Комитет ANSI/ISO проголосовал за включение STL в спецификацию C++. Добавление STL расширило сферу рассмотрения средств C++ далеко за пределы исходного определения языка.

Однако включение STL, помимо прочего, замедлило процесс стандартизации C++, причем довольно существенно. Помимо STL, в сам язык было добавлено несколько новых средств и внесено множество мелких изменений. Поэтому версия C++ после рассмотрения комитетом по стандартизации стала намного больше и сложнее по сравнению с исходным вариантом Страуструпа. Конечный результат работы комитета датируется 14 ноября 1997 года, а реально ANSI/ISO-стандарт языка C++ увидел свет в 1998 году. Именно эта спецификация C++ обычно называется Standard C++. И именно она описана в этой книге. Эта версия C++ поддерживается всеми основными C++-компиляторами, включая Visual C++ (Microsoft) и C++ Builder (Borland). Поэтому код программ, приведенных в этой книге, полностью применим ко всем современным C++-средам.

Что такое объектно-ориентированное программирование

Поскольку именно принципы объектно-ориентированного программирования были основополагающими для разработки C++, важно точно определить, что они собой представляют. Объектно-ориентированное программирование объединило лучшие идеи структурированного с рядом мощных концепций, которые способствуют более эффективной организации программ. Объектно-ориентированный подход к программированию позволяет разложить задачу на составные части таким образом, что каждая составная часть будет представлять собой самостоятельный объект, который содержит собственные инструкции и данные. При таком подходе существенно понижается общий уровень сложности программ, что позволяет программисту справляться с более сложными программами, чем раньше (т.е. написанными при использовании структурированного программирования).

Все языки объектно-ориентированного программирования характеризуются тремя общими признаками: инкапсуляцией, полиморфизмом и наследованием. Рассмотрим кратко каждый из них (подробно они будут описаны ниже в этой книге).

Инкапсуляция

Ни для кого не секрет, что все программы, как правило, состоят из двух основных элементов: инструкций (кода) и данных. *Код* — это часть программы, которая выполняет действия, а данные представляют собой информацию, на которую направлены эти действия. *Инкапсуляция* — это такой механизм программирования, который связывает воедино код и данные, которые он обрабатывает, чтобы обезопасить их как от внешнего вмешательства, так и от неправильного использования.

В объектно-ориентированном языке код и данные могут быть связаны способом, при котором создается самостоятельный черный ящик. В этом "ящике" содержатся все необходимые (для обеспечения самостоятельности) данные и код. При таком связывании кода и данных создается объект, т.е. *объект* — это конструкция, которая поддерживает инкапсуляцию.

Внутри объекта, код, данные или обе эти составляющие могут быть закрытыми в "рамках" этого объекта или открытыми. *Закрытый* код (или данные) известен и доступен только другим частям того же объекта. Другими словами, к закрытому коду или данным не может получить доступ та часть программы, которая существует вне этого объекта. *Открытый* код (или данные) доступен любым другим частям программы, даже если они определены в других объектах. Обычно открытые части объекта используются для предоставления управляемого интерфейса с закрытыми элементами объекта.

Полиморфизм

Полиморфизм (от греческого слова *polymorphism*, означающего "много форм") — это свойство, позволяющее использовать один интерфейс для целого класса действий. Конкретное действие определяется характерными признаками ситуации. В качестве простого примера полиморфизма можно привести руль автомобиля. Для руля (т.е. интерфейса) безразлично, какой тип рулевого механизма используется в автомобиле. Другим словами, руль работает одинаково, независимо от того, оснащен ли автомобиль рулевым управлением прямого действия (без усилителя), рулевым управлением с усилителем или механизмом реечной передачи. Если вы знаете, как обращаться с рулем, вы сможете вести автомобиль любого типа. Тот же принцип можно применить к программированию. Рассмотрим, например, *стек*, или *список*, добавление и удаление элементов к которому осуществляется по принципу "последним прибыл — первым обслужен". У вас может быть программа, в которой используются три различных типа стека. Один стек предназначен для целочисленных значений, второй — для значений с плавающей точкой и третий — для символов. Алгоритм реализации всех стеков — один и тот же, несмотря на то, что в них хранятся данные различных типов. В необъектно-ориентированном языке программисту пришлось бы создать три различных набора подпрограмм обслуживания стека, причем подпрограммы должны были бы иметь различные имена, а каждый набор — собственный интерфейс. Но благодаря полиморфизму в C++ можно создать один общий набор подпрограмм (один интерфейс), который подходит для всех трех конкретных ситуаций. Таким образом, зная, как использовать один стек, вы можете использовать все остальные.

В более общем виде концепция полиморфизма выражается фразой "один интерфейс — много методов". Это означает, что для группы связанных действий можно использовать один обобщенный интерфейс. Полиморфизм позволяет понизить уровень сложности за счет возможности применения одного и того же интерфейса для задания целого класса действий. Выбор же *конкретного действия* (т.е. функции) применительно к той или иной ситуации ложится "на плечи" компилятора. Вам, как программисту, не нужно делать этот выбор вручную. Ваша задача — использовать общий интерфейс.

Первые языки объектно-ориентированного программирования были реализованы в виде интерпретаторов, поэтому полиморфизм поддерживался во время выполнения программ. Однако C++ — это транслируемый язык (в отличие от интерпретируемого). Следовательно, в C++ полиморфизм поддерживается на уровне как компиляции программы, так и ее выполнения.

Наследование

Наследование — это процесс, благодаря которому один объект может приобретать свойства другого. Благодаря наследованию поддерживается концепция иерархической классификации. В виде управляемой иерархической (нисходящей) классификации организуется большинство областей знаний. Например, яблоки *Красный Делишес* являются частью классификации *яблоки*, которая в свою очередь является частью класса *фрукты*, а тот — частью еще большего класса *пища*. Таким образом, класс *пища* обладает определенными качествами (съедобность, питательность и пр.), которые применимы и к подклассу *фрукты*. Помимо этих качеств, класс *фрукты* имеет специфические характеристики (сочность, сладость и пр.), которые отличают их от других пищевых продуктов. В классе *яблоки* определяются качества, специфичные для яблок (растут на деревьях, не тропические и пр.). Класс *Красный Делишес* наследует качества всех предыдущих классов и при этом определяет качества, которые являются уникальными для этого сорта яблок.

Если не использовать иерархическое представление признаков, для каждого объекта пришлось бы в явной форме определить все присущие ему характеристики. Но благодаря наследованию объекту нужно доопределить только те качества, которые делают его уникальным внутри его класса, поскольку он (объект) наследует общие атрибуты своего родителя. Следовательно, именно механизм наследования позволяет одному объекту представлять конкретный экземпляр более общего класса.

C++ и реализация ООП

В этой книге показано, что многие средства C++ предназначены для поддержки инкапсуляции, полиморфизма и наследования. Однако следует помнить, что язык C++ можно использовать для написания программ любого типа. Тот факт, что C++ поддерживает объектно-ориентированное программирование, не означает, что C++-программист может писать только объектно-ориентированные программы. Одним из самых важных достоинств языка C++ (как и его предшественника, языка C) является гибкость.

Связь C++ с языками Java и C#

Вероятно, многие читатели знают о существовании таких языков программирования, как Java и C#. Язык Java разработан в компании Sun Microsystems, а C# — в компании Microsoft. Поскольку иногда возникает путаница относительно того, какое отношение эти два языка имеют к C++, попробуем внести ясность в этот вопрос.

C++ является родительским языком для Java и C#. И хотя разработчики Java и C# добавили к первоисточнику, удалили из него или модифицировали различные средства, в целом синтаксис этих трех языков практически идентичен. Более того, объектная модель, используемая C++, подобна объектным моделям языков Java и C#. Наконец, очень сходно общее впечатление и ощущение от использования всех этих языков. Это значит, что, зная C++, вы можете легко изучить Java или C#. Схожесть синтаксисов и объектных моделей — одна из причин быстрого освоения (и одобрения) этих двух языков многими опытными C++-программистами. Обратная ситуация также имеет место: если вы знаете Java или C#, изучение C++ не доставит вам хлопот.

Основное различие между C++, Java и C# заключается в типе вычислительной среды, для которой разрабатывался каждый из этих языков. C++ создавался с целью написания высокоэффективных программ, предназначенных для выполнения под управлением определенной операционной системы и в расчете на ЦП конкретного типа. Например, если вы хотите написать высокоэффективную программу для выполнения на процессоре Intel Pentium под управлением операционной системы Windows, лучше всего использовать для этого язык C++.

Языки Java и C# разработаны в ответ на уникальные потребности сильно распределенной

сетевой среды, которая может служить типичным примером современных вычислительных сред. Java позволяет создавать межплатформенный (совместимый с несколькими операционными средами) переносимый программный код для Internet. Используя Java, можно написать программу, которая будет выполняться в различных вычислительных средах, т.е. в широком диапазоне операционных систем и типов ЦП. Таким образом, Java-программа может свободно "бороздить просторами" Internet. C# разработан для среды .NET Framework (Microsoft), которая поддерживает *многоязычное программирование* (mixed-language programming) и компонентно-ориентированный код, выполняемый в сетевой среде.

Несмотря на то что Java и C# позволяют создавать переносимый программный код, который работает в сильно распределенной среде, цена этой переносимости — эффективность. Java-программы выполняются медленнее, чем C++-программы. То же справедливо и для C#. Поэтому, если вы хотите создавать высокоэффективные приложения, используйте C++. Если же вам нужны переносимые программы, используйте Java или C#.

И последнее. Языки C++, Java и C# предназначены для решения различных классов задач. Поэтому вопрос "Какой язык лучше?" поставлен некорректно. Уместнее задать вопрос по-другому: "Какой язык наиболее подходит для решения данной задачи?".

Глава 2: Обзор элементов языка C++

Самым трудным в изучении языка программирования, безусловно, является то, что ни один его элемент не существует изолированно от других. Компоненты языка работают вместе, можно сказать, в дружном "коллективе". Такая тесная взаимосвязь усложняет рассмотрение одного аспекта C++ без рассмотрения других. Зачастую обсуждение одного средства предусматривает предварительное знакомство с другим. Для преодоления подобных трудностей в этой главе приводится краткое описание таких элементов C++, как общий формат C++-программы, основные инструкции управления и операторы. При этом мы не будем углубляться в детали, а сосредоточимся на общих концепциях создания C++-программы. Большинство затронутых здесь тем более подробно рассматриваются в остальных главах книги.

Поскольку изучать язык программирования лучше всего путем реального программирования, мы рекомендуем читателю собственноручно выполнять приведенные в этой книге примеры на своем компьютере.

Первая C++-программа

Прежде чем зарываться в теорию, рассмотрим простую C++-программу. Начнем с вывода текста, а затем перейдем к ее компиляции и выполнению.

```
/* Программа №1 - Первая C++-программа.
Введите эту программу, затем скомпилируйте ее и выполните.
*/
#include <iostream>
using namespace std;

// main() - начало выполнения программы.
int main()
{
  cout << "Это моя первая C++-программа.";

  return 0;
}
```

Итак, вы должны выполнить следующие действия.

1. Ввести текст программы.

2. Скомпилировать ее.

3. Выполнить.

Исходный код — *это текстовая форма программы*. **Объектный код** — *это форма программы, которую может выполнить компьютер*.

Прежде чем приступать к выполнению этих действии, необходимо определить два термина: исходный код и объектный код. *Исходный код* — это версия программы, которую может читать человек. Приведенный выше листинг — это пример исходного кода Выполняемая версия программы называется *объектным*, или *выполняемым*, *кодом*.

Ввод текста программы

Программы, представленные в этой книге, можно загрузить с Web-сайта компании Osborne с адресом: www.osborne.com. При желании вы можете ввести текст программ вручную. В этом случае необходимо использовать какой-нибудь текстовый редактор (например WordPad), а не текстовой процессор (word processor). Дело в том, что при вводе текста программ должны быть созданы исключительно текстовые файлы, а не файлы, в которых вместе с текстом сохраняется информация о его форматировании. Помните, что информация о форматировании помешает работе C++-компилятора.

Имя файла, который будет содержать исходный код программы, формально может быть любым. Но C++-программы обычно хранятся в файлах с расширением *.cpp* . Поэтому называйте свои C++-программы любыми именами, но в качестве расширения используйте *.cpp* . Например, назовите нашу первую программу *MyProg.cpp* (это имя будет употребляться в дальнейших инструкциях), а для других программ (если не будет специальных указаний) выбирайте имена по своему усмотрению.

Компилирование программы

Способ компиляции программы *MyProg.cpp* зависит от используемого компилятора и выбранных опций. Более того, многие компиляторы, например Visual C++ (Microsoft) и C++ Builder (Borland), предоставляют два различных способа компиляции программ: с помощью компилятора командной строки и интегрированной среды разработки (Integrated Development Environment — IDE). Поэтому для компилирования C++-программ невозможно дать универсальные инструкции, которые подойдут для всех компиляторов. Это значит, что вы должны следовать инструкциям, приведенным в сопроводительной документации, прилагаемой к вашему компилятору.

Но, как упоминалось выше, самыми популярными компиляторами являются Visual C++ и C++ Builder, поэтому для удобства читателей, которые их используют, мы приведем здесь инструкции по компиляции программ, соответствующие этим компиляторам. Проще всего в обоих случаях компилировать и выполнять программы, приведенные в этой книге, с использованием компиляторов командной строки. Так мы и поступим.

Чтобы скомпилировать программу *MyProg.cpp*, используя Visual C++, введите следующую командную строку:

```
C:\...>cl -GX MyProg.cpp
```

Опция *-GX* предназначена для повышения качества компиляции. Чтобы использовать компилятор командной строки Visual C++, необходимо выполнить пакетный файл VCVARS32.bat, который входит в состав Visual C++.

Чтобы скомпилировать программу *MyProg.cpp*, используя C++ Builder, введите такую командную строку:

```
C: \...>bcc32 MyProg.cpp
```

В результате работы C++-компилятора получается выполняемый объектный код. Для Windows-среды выполняемый файл будет иметь то же имя, что и исходный, но другое расширение, а именно расширение *.exe*. Итак, выполняемая версия программы *MyProg.cpp* будет храниться в файле *MyProg.exe*.

На заметку. *Если при попытке скомпилировать первую программу вы получили сообщение об ошибке, но уверены, что ввели ее текст корректно, то, возможно, вы используете*

старую версию C++-компилятора, который был создан до принятия C++-стандарта ANSI/ISO. В этом случае обратитесь к приложению Б за инструкциями по использованию старых компиляторов.

Выполнение программы

Скомпилированная программа готова к выполнению. Поскольку результатом работы - C++-компилятора является выполняемый объектный код, то для запуска программы в качестве команды достаточно ввести ее имя. Например, чтобы выполнить программу *MyProg.exe*, используйте эту командную строку:

```
C:\...>MyProg.cpp
```

Результаты выполнения этой программы таковы:

```
Это моя первая C++-программа.
```

Если вы используете интегрированную среду разработки, то выполнить программу можно путем выбора из меню команды *Run* (Выполнить). Безусловно, более точные инструкции приведены в сопроводительной документации, прилагаемой к вашему компилятору. Но, как упоминалось выше, проще всего компилировать и выполнять приведенные в этой книге программы с помощью командной строки.

Необходимо отметить, что все эти программы представляют собой консольные приложения, а не приложения, основанные на применении окон, т.е. они выполняются в сеансе приглашения на ввод команды. При этом вам, должно быть, известно, что язык C++ не просто подходит для Windows-программирования, C++ — основной язык, применяемый в разработке Windows-приложений. Однако ни одна из программ, представленных в этой книге, не использует графического интерфейса пользователя (GUI — *graphics use interface*). Дело в том, что Windows — довольно сложная среда для написания программ включающая множество второстепенных тем, не связанных напрямую с языком C++ В то же время консольные приложения гораздо короче графических и лучше подходят для обучения программированию. Освоив C++, вы сможете без проблем применить свои знания в сфере создания Windows-приложений.

Построчный "разбор полетов"

После успешной компиляции и выполнения первого примера программы настало время разобраться в том, как она работает. Поэтому мы подробно рассмотрим каждую её строку. Итак, наша программа начинается с таких строк.

```
/* Программа №1 - Первая C++-программа.
Введите эту программу, затем скомпилируйте ее и выполните.
*/
```

Это — *комментарий*. Подобно большинству других языков программирования, C++ позволяет вводить в исходный код программы комментарии, содержание которых компилятор игнорирует. С помощью комментариев описываются или разъясняются действия, выполняемые в программе, и эти разъяснения предназначаются для тех, кто будет читать исходный код. В данном случае комментарий просто идентифицирует программу и напоминает, что с ней нужно сделать. Конечно, в реальных приложениях комментарии используются для разъяснения особенностей работы отдельных частей программы или конкретных действий программных средств. Другими словами, вы можете использовать комментарии для детального описания всех (или некоторых) ее строк.

Комментарий — *это текст пояснительного содержания, встраиваемый в программу.*

В C++ поддерживается два типа комментариев. Первый, показанный в начале рассматриваемой программы, называется многострочным. Комментарий этого типа должен начинаться символами */** и заканчиваться ими же, но переставленными в обратном порядке (***/**). Все, что находится между этими парами символов, компилятор игнорирует. Комментарий этого типа, как следует из его названия, может занимать несколько строк. Второй тип комментариев мы рассмотрим чуть ниже.

Приведем здесь следующую строку программы.

```
#include <iostream>
```

В языке C++ определен ряд заголовков (header), которые обычно содержат информацию, необходимую для программы. В нашу программу включен заголовок *<iostream>* (он используется для поддержки в C++-системы ввода-вывода), который представляет собой внешний исходный файл, помещаемый компилятором в начало программы с помощью директивы *#include*. Ниже в этой книге мы ближе познакомимся с заголовками и узнаем, почему они так важны.

Рассмотрим следующую строку программы:

```
using namespace std;
```

Эта строка означает, что компилятор должен использовать пространство имен *std*. Пространства имен — относительно недавнее дополнение к языку C++. Подробнее о них мы поговорим позже, а пока ограничимся их кратким определением. *Пространство имен* (namespace) создает декларативную область, в которой могут размещаться различные элементы программы. Пространство имен позволяет хранить одно множество имен отдельно от другого. Другими словами, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом. Пространства имен позволяют упростить организацию больших программ. Ключевое слово *using* информирует компилятор об использовании заявленного пространства имен (в данном случае *std*). Именно в пространстве имен *std* объявлена вся библиотека стандарта C++. Таким образом, используя пространство имен *std*, вы упрощаете доступ к стандартной библиотеке языка.

Очередная строка в нашей программе представляет собой *однострочный комментарий*.

```
// main() - начало выполнения программы.
```

Так выглядит комментарий второго типа, поддерживаемый в C++. Однострочный комментарий начинается с пары символов *//* и заканчивается в конце строки. Как правило, программисты используют многострочные комментарии для подробных и потому более пространных разъяснений, а однострочные — для кратких (построчных) описаний инструкций или назначения переменных. Вообще-то, характер использования комментариев — личное дело программиста.

Перейдем к следующей строке:

```
int main()
```

Как сообщается в только что рассмотренном комментарии, именно с этой строки и начинается выполнение программы.

С функции main() начинается выполнение любой C++-программы.

Все C++-программы состоят из одной или нескольких функций. (Под *функцией* main понимаем подпрограмму.) Каждая C++-функция имеет имя, и только одна из них (её должна включать каждая C++-программа) называется *main()*. Выполнение C++ программы начинается и заканчивается (в большинстве случаев) выполнением функции *main()*. (Точнее, C++-программа начинается с вызова функции *main()* и обычно заканчивается возвратом из функции *main()*.) Открытая фигурная скобка на следующей (после *int main()*) строке указывает на начало кода функции *main()*. Ключевое слово *int* (сокращение от слова *integer*), стоящее перед именем main(), означает тип данных для значения, возвращаемого функцией main(). Как вы скоро узнаете, C++ поддерживает несколько встроенных типов данных, и *int* — один из них.

Рассмотрим очередную строку программы:

```
cout << "Это моя первая C++-программа.";
```

Это инструкция вывода данных на консоль. При ее выполнении на экране компьютера отобразится сообщение *Это моя первая C++-программа.*. В этой инструкции используется оператор вывода *"<<"*. Он обеспечивает вывод выражения, стоящего с правой стороны, на устройство, указанное с левой. Слово *cout* представляет собой встроенный идентификатор (составленный из частей слов *console output*), который в большинстве случаев означает экран компьютера. Итак, рассматриваемая инструкции обеспечивает вывод заданного сообщения на экран. Обратите внимание на то, что эта инструкция завершается точкой с запятой. В действительности все выполняемые C++-инструкции завершаются точкой с запятой.

Сообщение *"Это моя первая C++-программа."* представляет собой *строку* В C++ под строкой понимается последовательность символов, заключенная в двойные кавычки. Как вы увидите, строка в C++ — это один из часто используемых элементов языка.

А этой строкой завершается функция *main()*:

```
return 0;
```

При ее выполнении функция *main()* возвращает вызывающему процессу (в роли которого обычно выступает операционная система) значение *0*. Для большинства операционных систем нулевое значение, которое возвращает эта функция, свидетельствует о нормальном завершении программы. Другие значения могут означать завершение программы в связи с какой-нибудь ошибкой. Слово *return* относится к числу ключевых используется для возврата значения из функции. При нормальном завершении (т.е. без ошибок) все ваши программы должны возвращать значение 0.

Закрывающая фигурная скобка в конце программы формально завершает ее. Хотя фигурная скобка в действительности не является частью объектного кода программы, её "выполнение" (т.е. обработку закрывающей фигурной скобки функции *main()*) мысленно можно считать концом C++-программы. И в самом деле, если в этом примере программы инструкция *return* отсутствовала бы, программа автоматически завершилась бы по достижении этой закрывающей фигурной скобки.

Обработка синтаксических ошибок

Каждому программисту известно, насколько легко при вводе текста программы в компьютер вносятся случайные ошибки (опечатки). К счастью, при попытке скомпилировать такую программу компилятор "просигналит" сообщением о наличии *синтаксических ошибок*. Большинство C++-компиляторов попытаются "увидеть" смысл в исходном коде программы, независимо от того, что вы ввели. Поэтому сообщение об ошибке не всегда отражает истинную причину проблемы. Например, если в предыдущей программе случайно опустить открывающую фигурную скобку после имени функции *main()*, компилятор укажет в качестве источника ошибки инструкцию *cout*. Поэтому при получении сообщения об ошибке просмотрите две-три строки кода, непосредственно предшествующих строке с "обнаруженной" ошибкой. Ведь иногда компилятор начинает "чуять недоброе" только через несколько строк после реального местоположения ошибки.

Многие C++-компиляторы выдают в качестве результатов своей работы не только сообщения об ошибках, но и предупреждения (warning). В язык C++ "от рождения" заложено великодушное отношение к программисту, т.е. он позволяет программисту практически все, что корректно с точки зрения синтаксиса. Однако даже "всепрощающим" C++-компиляторам некоторые синтаксически правильные вещи могут показаться подозрительными. В таких ситуациях и выдается предупреждение. Тогда программист сам должен оценить, насколько справедливы подозрения компилятора. Откровенно говоря, некоторые компиляторы слишком уж бдительны и предупреждают по поводу совершенно корректных инструкций. Кроме того, компиляторы позволяют использовать различные опции, которые могут информировать об интересующих вас вещах. Иногда такая информация имеет форму предупреждающего сообщения даже несмотря на отсутствие "состава" предупреждения. Программы, приведенные в этой книге, написаны в соответствии со стандартом C++ и при корректном вводе не должны генерировать никаких предупреждающих сообщений.

Важно! *Большинство C++-компиляторов предлагают несколько уровней сообщений (и*

предупреждений) об ошибках. В общем случае можно выбрать тип ошибок, о наличии которых вы хотели бы получать сообщения. Например, большинство компиляторов по желанию программиста могут информировать об использовании неэффективных конструкций или устаревших средств. Для примеров этой книги достаточно использовать обычную настройку компилятора. Но вам все же имеет смысл заглянуть в прилагаемую к компилятору документацию и поинтересоваться, какие возможности по управлению процессом компиляции есть в вашем распоряжении. Многие компиляторы довольно "интеллектуальны" и могут помочь в обнаружении неочевидных ошибок еще до того, как они перерастут в большие проблемы. Знание принципов, используемых компилятором при составлении отчета об ошибках, стоит затрат времени и усилий, которые потребуются от программиста на их освоение.

Вторая C++-программа

Возможно, самой важной конструкцией в любом языке программирования является присвоение переменной некоторого значения. *Переменная* — это именованная область памяти, в которой могут храниться различные значения. При этом значение переменной во время выполнения программы можно изменить один или несколько раз. Другими словами, содержимое переменной изменяемо, а не фиксированно.

В следующей программе создается переменная с именем *x*, которой присваивается значение *1023*, а затем на экране отображается сообщение *Эта программа выводит значение переменной x: 1023*.

```
// Программа №2 - Использование переменной.
#include <iostream>
using namespace std;

int main( )
{
  int x; // Здесь объявляется переменная.
  x = 1023; // Здесь переменной x присваивается число 1023.
  cout << "Эта программа выводит значение переменной x: ";
  cout << x; // Отображение числа 1023.

  return 0;
}
```

Что же нового в этой программе? Во-первых, инструкция:

```
int x; // Здесь объявляется переменная.
```

объявляет переменную с именем *x* целочисленного типа. В C++ все переменные должны быть объявлены до их использования. В объявлении переменной помимо ее имени необходимо указать, значения какого типа она может хранить. Тем самым объявляется *тип* переменной. В данном случае переменная *x* может хранить целочисленные значения, т.е. целые числа, лежащие в диапазоне *-32 768--32 767*. В C++ для объявления переменной целочисленного типа достаточно поставить перед ее именем ключевое слово *int*. Таким образом, инструкция *int x;* объявляет переменную *x* типа *int*. Ниже вы узнаете, что C++ поддерживает широкий диапазон встроенных типов переменных. (Более того, C++ позволяет программисту определять собственные типы данных.)

Во-вторых, при выполнении следующей инструкции переменной присваивается конкретное значение:

```
x = 1023; // Здесь переменной x присваивается число 1023.
```

В C++ *оператор присваивания* представляется одиночным знаком равенства (=). Его действие заключается в копировании значения, расположенного справа от оператора, в переменную, указанную слева от него. После выполнения этой инструкции присваивания переменная *x* будет содержать число *1023*.

Результаты, сгенерированные этой программой, отображаются на экране с помощью двух инструкций *cout*. Обратите внимание на использование следующей инструкции для вывода значения переменной *x*:

```
cout << x; // Отображение числа 1023.
```

В общем случае для отображения значения переменной достаточно в инструкции *cout* поместить ее имя справа от оператора *"<<"*. Поскольку в данном конкретном случае переменная *x* содержит число *1023*, то оно и будет отображено на экране. Прежде чем переходить к следующему разделу, попробуйте присвоить переменной *x* другие значения (в исходном коде) и посмотрите на результаты выполнения этой программы после внесения изменений.

Более реальный пример

Первые две программы, кроме демонстрации ряда важных средств языка C++, не делали ничего полезного. В следующей программе решается практическая задача преобразования галлонов в литры. Здесь также показан один из способов ввода данных в программу.

```
// Эта программа преобразует галлоны в литры.
#include <iostream>
using namespace std;

int main( )
{
  int gallons, liters;
  cout << "Введите количество галлонов:";
    cin >> gallons; // Ввод данных от пользователя.
  liters = gallons * 4; // Преобразование в литры.
  cout << "Литров: " << liters;

  return 0;
}
```

Эта программа сначала отображает на экране сообщение, предлагающее пользователю ввести число для преобразования галлонов в литры, а затем ожидает до тех пор, пока оно не будет введено. (Помните, вы должны ввести целое число галлонов, т.е. число, не содержащее дробной части.) Затем программа отобразит значение, приблизительно равное эквивалентному объему, выраженному в литрах. В действительности для получения точного результата необходимо использовать коэффициент *3,7854* (т.е. в одном галлоне помещается *3,7854* литра), но поскольку в этом примере мы работаем с целочисленными переменными, то коэффициент преобразования округлен до *4*.

Обратите внимание на то, что две переменные *gallons* и *liters* объявляются после ключевого слова *int* в форме списка, элементы которого разделяются запятыми. В общем случае можно объявить любое количество переменных одного типа, разделив их запятыми. (В качестве альтернативного варианта можно использовать несколько декларативных *int*-инструкций — результат будет тот же.)

Для приема значения, вводимого пользователем, используется следующая инструкция:

```
cin >> gallons; // Ввод данных от пользователя.
```

Здесь применяется еще один встроенный идентификатор — *cin* — предоставляемый C++-компилятором. Он составлен из частей слов *console input* и в большинстве случаев означает ввод данных с клавиатуры. В качестве оператора ввода используется символ *">>"*. При выполнении этой инструкции значение, введенное пользователем (которое в данном случае должно быть целочисленным), помещается в переменную, указанную с правой стороны от

оператора *">>"* (в данном случае это переменная *gallons*).

В этой программе заслуживает внимания и эта инструкция:

```
cout << "Литров: " << liters;
```

Здесь интересно то, что в одной инструкции использовано сразу два оператора вывода "<<". При ее выполнении сначала будет выведена строка *"Литров: "*, а за ней — значение переменной *liters*. В общем случае в одной инструкции Можно соединять любое количество операторов вывода, предварив каждый элемент вывода *"своим"* оператором

Новый тип данных

Несмотря на то что для приблизительных подсчетов вполне сгодится рассмотренная выше программа преобразования галлонов в литры, для получения более точных результатов ее необходимо переделать. Как отмечено выше, с помощью целочисленных типов данных невозможно представить значения с дробной частью. Для них нужно использовать один из типов данных с плавающей точкой, например *double* (двойной точности). Данные этого типа обычно находятся в диапазоне *1,7E-308--1,7E+308*. Операции, вы-полняемые над числами с плавающей точкой, сохраняют любую дробную часть результата и, следовательно, обеспечивают более точное преобразование.

В следующей версии программы преобразования используются значения с плавающей точкой.

```
/* Эта программа преобразует галлоны в литры с помощью чисел с плавающей точкой.
*/
#include <iostream>
using namespace std;

int main( )
{
  double gallons, liters;
  cout << "Введите количество галлонов: ";
    cin >> gallons; // Ввод данных от пользователя.
  liters = gallons * 3.7854; // Преобразование в литры.
  cout << "Литров: " << liters;

  return 0;
}
```

Для получения этого варианта в предыдущую программу было внесено два изменения. Во-первых, переменные *gallons* и *liters* объявлены на этот раз с использованием типа *double*. Во-вторых, коэффициент преобразования задан в виде числа *3.7854*, что позволяет получить более точные результаты. Если C++-компилятор встречает число, содержащее десятичную точку, он автоматически воспринимает его как константу с плавающей точкой. Обратите также внимание на то, что инструкции *cout* и *cin* остались такими же, как в предыдущем варианте программы, в которой использовались переменные типа *int*. Это очень важный момент: C++-система ввода-вывода автоматически настраивается на тип данных, указанный в программе.

Скомпилируйте и выполните эту программу. На приглашение указать количество галлонов введите число *1*. В качестве результата программа должна отобразить *3,7854* литра.

Повторим пройденное

Итак, подытожим самое важное из уже прочитанного материала.

1. Каждая C++-программа должна иметь функцию *main()*, которая означает начало

выполнения программы.

2. Все переменные должны быть объявлены до их использования.

3. C++ поддерживает различные типы данных, включая целочисленные и с плавающей точкой.

4. Оператор вывода данных обозначается символом *"<<"*, а при использовании в инструкции *cout* он обеспечивает отображение информации на экране компьютера.

5. Оператор ввода данных обозначается символом *">>"*, а при использовании в инструкции *cin* он считывает информацию с клавиатуры.

6. Выполнение программы завершается с окончанием функции *main()*.

Функции

Любая C++-программа составляется из "строительных блоков", именуемых функциями. Функция — это подпрограмма, которая содержит одну или несколько C++-инструкий и выполняет одну или несколько задач. Хороший стиль программирования на C++ предполагает, что каждая функция выполняет только одну задачу.

Каждая функция имеет *имя*, которое используется для ее вызова. Своим функциям программист может давать любые имена за исключением имени *main()*, зарезервированного для функции, с которой начинается выполнение программы.

Функции — *это "строительные блоки" C++-программы*.

В C++ ни одна функция не может быть встроена в другую. В отличие от таких языков программирования, как Pascal, Modula-2 и некоторых других, которые позволяют использование вложенных функций, в C++ все функции рассматриваются как отдельные компоненты. (Безусловно, одна функция может вызывать другую.)

При обозначении функций в тексте этой книги используется соглашение (обычно соблюдаемое в литературе, посвященной языку программирования C++), согласно которому имя функции завершается парой круглых скобок. Например, если функция имеет имя *getval*, то ее упоминание в тексте обозначится как *getval()*. Соблюдение этого соглашения позволит легко отличать имена переменных от имен функций.

В уже рассмотренных примерах программ функция main() была единственной. Как упоминалось выше, функция main() — первая функция, выполняемая при запуске программы. Ее должна содержать каждая C++-программа. Вообще, функции, которые вам предстоит использовать, бывают двух типов. К первому типу относятся функции, написанные программистом (main() — пример функции такого типа). Функции другого типа находятся в *стандартной библиотеке* C++-компилятора. (Стандартная библиотека будет рассмотрена ниже, а пока заметим, что она представляет собой коллекцию встроенных функций.) Как правило, C++-программы содержат как функции, написанные программистом, так и функции, предоставляемые компилятором.

Поскольку функции образуют фундамент C++, займемся ими вплотную.

Программа с двумя функциями

Следующая программа содержит две функции: *main()* и *myfunc()*. Еще до выполнения этой программы (или чтения последующего описания) внимательно изучите ее текст и попытайтесь предугадать, что она должна отобразить на экране.

```
/* Эта программа содержит две функции: main( ) и myfunc( ).
*/
#include <iostream>
using namespace std;

void myfunc( ); // прототип функции myfunc( )
int main( )
```

```
{
  cout << "В функции main().";
  myfunc(); // Вызываем функцию myfunc().
  cout << "Снова в функции main().";
  return 0;
}
void myfunc() {
  cout << " В функции myfunc(). ";
}
```

Программа работает следующим образом. Вызывается функция *main()* и выполняется ее первая *cout*-инструкция. Затем из функции *main()* вызывается функция *myfunc()*. Обратите внимание на то, как этот вызов реализуется в программе: указывается имя функции *myfunc*, за которым следуют пара круглых скобок и точка с запятой. Вызов любой функции представляет собой C++-инструкцию и поэтому должен завершаться точкой с запятой. Затем функция *myfunc()* выполняет свою единственную *cout*-инструкцию и передает управление назад функции *main()*, причем той строке кода, которая расположена непосредственно за вызовом функции. Наконец, функция *main()* выполняет свою вторую *cout*-инструкцию, которая завершает всю программу. Итак, на экране мы должны увидеть такие результаты.

```
В функции main().
В функции myfunc().
Снова в функции main().
```

В этой программе необходимо рассмотреть следующую инструкцию.

```
void myfunc(); // прототип функции myfunc()
```

Прототип объявляет функцию до ее первого использования.

Как отмечено в комментарии, это — *прототип* функции *myfunc()*. Хотя подробнее прототипы будут рассмотрена ниже, все же без кратких пояснений здесь не обойтись. Прототип функции объявляет функцию до ее определения. Прототип позволяет компилятору узнать тип значения, возвращаемого этой функцией, а также количество и тип параметров, которые она может иметь. Компилятору нужно знать эту информацию до первого вызова функции. Поэтому прототип располагается до функции main(). Единственной функцией, которая не требует прототипа, является main(), поскольку она встроена в язык C++.

Как видите, функция *myfunc()* не содержит инструкцию *return*. Ключевое слово *void*, которое предваряет как прототип, так и определение функции *myfunc()*, формально заявляет о том, что функция *myfunc()* не возвращает никакого значения. В C++ функции, не возвращающие значений, объявляются с использованием ключевого слова *void*.

Аргументы функций

Функции можно передать одно или несколько значений. Значение, передаваемое функции, называется *аргументом*. Несмотря на то что в программах, которые мы рассматривали до сих пор, ни одна из функций (ни main(), ни myfunc()) не принимала никаких значений, функции в C++ могут принимать один или несколько аргументов. Верхний предел числа принимаемых аргументов определяется конкретным компилятором. Согласно стандарту C++ он равен 256.

Аргумент — *это значение, передаваемое функции при вызове*.

Рассмотрим короткую программу, которая для отображения абсолютного значения числа использует стандартную библиотечную (т.е. встроенную) функцию *abs()*. Эта функция

принимает один аргумент, преобразует его в абсолютное значение и возвращает результат.

```
// Использование функции abs().
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  cout << abs(-10);

  return 0;
}
```

Здесь функции *abs()* в качестве аргумента передается число *-10*. Функция *abs()* принимает этот аргумент при вызове и возвращает его абсолютное значение, которое в свою очередь передаётся инструкции *cout* для отображения на экране абсолютного значения числа *-10*. Дело в том, что если функция является частью выражения, она автоматически вызывается для получения возвращаемого ею значения. В данном случае значение, возвращаемое функцией *abs()*, оказывается справа от оператора *"<<"* и поэтому законно отображается на экране.

Обратите также внимание на то, что рассматриваемая программа включает заголовок *<cstdlib>*. Этот заголовок необходим для обеспечения возможности вызова функции *abs()*. Каждый раз, когда вы используете библиотечную функцию, в программу необходимо включать соответствующий заголовок. Заголовок, помимо прочей информации, содержит прототип библиотечной функции.

Параметр — *это определяемая функцией переменная*, *которая принимает передаваемый функции аргумент*.

При создании функции, которая принимает один или несколько аргументов, иногда необходимо объявить переменные, которые будут хранить значения аргументов. Эти переменные называются *параметрами* функции. Например, следующая функция выводит произведение двух целочисленных аргументов, передаваемых функции при ее вызове.

```
void mul (int x, int y)
{
  cout << x * y << " ";
}
```

При каждом вызове функции *mul()* выполняется умножение значения, переданного параметру *x*, на значение, переданное параметру *y*. Однако помните, что *x* и *y* — это просто переменные, которые принимают значения, передаваемые при вызове функции.

Рассмотрим следующую короткую программу, которая демонстрирует использование функции *mul()*.

```
// Простая программа, которая демонстрирует использование функции mul().
#include <iostream>
using namespace std;

void mul(int x, int y); // Прототип функции mul().
int main()
{
  mul (10, 20);
  mul (5, 6);
```

```
  mul (8, 9);
  return 0;
}
void mul(int x, int y)
{
  cout << x * y << " ";
}
```

Эта программа выведет на экран числа *200*, *30* и *72*. При вызове функции *mul()* C++-компилятор копирует значение каждого аргумента в соответствующий параметр. В данном случае при первом вызове функции *mul()* число *10* копируется в переменную *x*, а число *20* — в переменную *y*. При втором вызове *5* копируется в *x*, а *6* — в *y*. При третьем вызове *8* копируется в *x*, а *9* — в *y*.

Если вы никогда не работали с языком программирования, в котором разрешены параметризованные функции, описанный процесс может показаться несколько странным. Однако волноваться не стонт: по мере рассмотрения других C++-программ принцип использования функций, их аргументов и параметров станет более понятным.

Узелок на память. *Термин аргумент относится к значению, которое используется при вызове функции. Переменная, которая принимает этот аргумент, называется параметром. Функции, которые принимают аргументы, называются параметризованными функциями.*

Если C++-функции имеют два или больше аргументов, то они разделяются запятыми. В этой книге под термином *список аргументов* следует понимать аргументы, разделенные запятыми. Для рассмотренной выше функции *mul()* список аргументов выражен в виде *x*, *y*.

Функции, возвращающие значения

В C++ многие библиотечные функции возвращают значения. Например, уже знакомая вам функция *abs()* возвращает абсолютное значение своего аргумента. Функции, написанные программистом, также могут возвращать значения. В C++ для возврата значения используется инструкция *return*. Общий формат этой инструкции таков:

```
return значение;
```

Нетрудно догадаться, что здесь элемент *значение* представляет собой значение, возвращаемое функцией.

Чтобы продемонстрировать процесс возврата функциями значений, переделаем предыдущую программу так, как показано ниже. В этой версии функция *mul()* возвращает произведение своих аргументов. Обратите внимание на то, что расположение функции справа от оператора присваивания означает присваивание переменной (расположенной слева) значения, возвращаемого этой функцией.

```
// Демонстрация возврата функциями значений.
#include <iostream>
using namespace std;

int mul (int x, int y); // Прототип функции mul().
int main()
{
  int answer;
   answer = mul (10, 11); // Присваивание значения, возвращаемого функцией.
  cout << "Ответ равен" << answer;
```

```
  cout << "Ответ равен " << answer;

  return 0;
}

// Эта функция возвращает значение.
int mul (int x, int y)
{
  return x * y; // Функция возвращает произведение x и y.
}
```

В этом примере функция *mul()* возвращает результат вычисления выражения *x*y* с помощью инструкции *return*. Затем значение этого результата присваивается переменной *answer*. Таким образом, значение, возвращаемое инструкцией *return*, становится значением функции *mul()* в вызывающей программе.

Поскольку в этой версии программы функция *mul()* возвращает значение, ее имя в определении не предваряется словом *void*. (Вспомните, слово *void* используется только в том случае, когда функция не возвращает никакого значения.) Поскольку существуют различные типы переменных, существуют и различные типы значений, возвращаемых функциями. Здесь функция *mul()* возвращает значение целочисленного типа. Тип значения, возвращаемого функцией, предшествует ее имени как в прототипе, так и в определении.

В более ранних версиях C++ для типов значений, возвращаемых функциями, существовало соглашение, действующее по умолчанию. Если тип возвращаемого функцией значения не указан, предполагалось, что эта функция возвращает целочисленное значение. Например, функция *mul()* согласно тому соглашению могла быть записана так.

```
// Устаревший способ записи функции mul().
mul (int X, int y) /* По умолчанию в качестве типа значения, возвращаемого функцией, используется тип int.*/
{
  return x * y; // Функция возвращает произведение x и y.
}
```

Здесь по умолчанию предполагается целочисленный тип значения, возвращаемого функцией, поскольку не задан никакой другой тип. Однако правило установки целочисленного типа по умолчанию было отвергнуто стандартом C++. Несмотря на то что большинство компиляторов поддерживают это правило ради обратной совместимости, вы должны явно задавать тип значения, возвращаемого каждой функцией, которую пишете. Но если вам придется разбираться в старых версиях C++-программ, это соглашение следует иметь в виду.

При достижении инструкции *return* функция немедленно завершается, а весь остальной код игнорируется. Функция может содержать несколько инструкций *return*. Возврат из функции можно обеспечить с помощью инструкции *return* без указания возвращаемого значения, но такую ее форму допустимо применять только для функций, которые не возвращают никаких значений и объявлены с использованием ключевого слова *void*.

Функция main()

Как вы уже знаете, функция main() — специальная, поскольку это первая функция которая вызывается при выполнении программы. В отличие от некоторых других языков программирования, в которых выполнение всегда начинается "сверху", т.е. с первой строки кода, каждая C++-программа всегда начинается с вызова функции main() независимо от ее расположения в программе. (Все же обычно функцию main() размещают первой, чтобы ее было легко найти.)

В программе может быть только одна функция main(). Если попытаться включить в

В программе может быть только одна функция main(). Если попытаться включить в программу несколько функций main(), она "не будет знать", с какой из них начать работу. В действительности большинство компиляторов легко обнаружат ошибку этого типа и сообщат о ней. Как упоминалось выше, поскольку функция main() встроена в язык C++, она не требует прототипа.

Общий формат C++-функций

В предыдущих примерах были показаны конкретные типы функций. Однако все C++-функции имеют такой общий формат.

```
тип_возвращаемого_значения имя (список_параметров) {

.
.// тело метода

.
}
```

Рассмотрим подробно все элементы, составляющие функцию.

С помощью элемента *тип_возвращаемого_значения* указывается тип значения, возвращаемого функцией. Как будет показано ниже в этой книге, это может быть практически любой тип, включая типы, создаваемые программистом. Если функция не возвращает никакого значения, необходимо указать тип *void*. Если функция действительно возвращает значение, оно должно иметь тип, совместимый с указанным в определении функции.

Каждая функция имеет имя. Оно, как нетрудно догадаться, задается элементом *имя*. После имени функции между круглых скобок указывается список параметров, который представляет собой последовательность пар (состоящих из типа данных и имени), разделенных запятыми. Если функция не имеет параметров, элемент *список_параметров* отсутствует, т.е. круглые скобки остаются пустыми.

В фигурные скобки заключено тело функции. Тело функции составляют C++-инструкции, которые определяют действия функции. Функция завершается (и управление передается вызывающей процедуре) при достижении закрывающей фигурной скобки или инструкции *return*.

Некоторые возможности вывода данных

До сих пор у нас не было потребности при выводе данных обеспечивать переход на следующую строку. Однако такая необходимость может потребоваться очень скоро. В C++ последовательность символов "*возврат каретки/перевод строки*" генерируется с помощью *символа новой строки*. Для вывода этого символа используется такой код: *\n* (символ обратной косой черты и строчная буква *n*). Продемонстрируем использование последовательности символов "*возврат каретки/перевод строки*" на примере следующей программы.

```
/*  Эта программа демонстрирует \n-последовательность, которая
обеспечивает переход на новую строку.
*/
#include <iostream>
using namespace std;

int main( )
{
  cout << "один\n";
  cout << "два\n";
  cout << "три";
```

```
  cout << "четыре";

  return 0;
}
```

При выполнении программа генерирует такие результаты:

```
один
два
тричетыре
```

Символ новой строки можно поместить в любом месте строки, а не только в конце. "Поиграйте" с символом новой строки, чтобы убедиться в том, что вы правильно понимаете его назначение.

Две простые инструкции

Для рассмотрения более реальных примеров программ нам необходимо познакомиться с двумя C++-инструкциями: *if* и *for*. (Более подробное их описание приведено ниже в этой книге.)

Инструкция if

Инструкция if позволяет сделать выбор между двумя выполняемыми ветвями программы.

Инструкция *if* в C++ действует подобно инструкции *IF*, определенной в любом другом языке программирования. Её простейший формат таков:

```
if(условие) инструкция;
```

Здесь элемент *условие* — это выражение, которое при вычислении может оказаться равным значению ИСТИНА или ЛОЖЬ. В C++ ИСТИНА представляется ненулевым значением, а ЛОЖЬ — нулем. Если *условие*, или условное выражение, истинно, элемент инструкция выполнится, в противном случае — нет. При выполнении следующего фрагмента кода на экране отобразится фраза *10 меньше 11*.

```
if(10 < 11) cout << "10 меньше 11";
```

Такие операторы сравнения, как *"<"* (меньше) и *">="* (больше или равно), используются во многих других языках программирования. Но следует помнить, что в C++ в качестве оператора равенства применяется двойной символ "равно" (==). В следующем примере *cout*-инструкция не выполнится, поскольку условное выражение дает значение ЛОЖЬ. Другими словами, поскольку *10* не равно *11*, *cout*-инструкция не отобразит на экране приветствие.

```
if(10==11) cout << "Привет";
```

Безусловно, операнды условного выражения необязательно должны быть константами. Они могут быть переменными и даже содержать обращения к функциям.

В следующей программе показан пример использования *if*-инструкции. При выполнении этой программы пользователю предлагается ввести два числа, а затем сообщается результат их сравнения.

```
// Эта программа демонстрирует использование if-инструкции.
#include <iostream>
using namespace std;

int main()
{
  int a,b;
  cout << "Введите первое число: ";
    cin >> a;
```

```
  cout << "Введите второе число: ";
    cin >> b;
  if(a < b) cout << "Первое число меньше второго.";

  return 0;
}
```

Цикл for

for — *одна из циклических инструкций, определенных в C++.*

Цикл *for* повторяет указанную инструкцию заданное число раз. Инструкция *for* в C++ действует практически так же, как инструкция *FOR*, определенная в таких языках программирования, как Java, C#, Pascal и BASIC. Ее простейший формат таков:

```
for(инициализация; условие; инкремент) инструкция;
```

Здесь элемент *инициализация* представляет собой инструкцию присваивания, которая устанавливает *управляющую переменную цикла* равной начальному значению. Эта переменная действует в качестве счетчика, который управляет работой цикла. Элемент *условие* представляет собой выражение, в котором тестируется значение управляющей переменной цикла. Результат этого тестирования определяет, выполнится цикл *for* еще раз или нет. Элемент *инкремент* — это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации. Цикл *for* будет выполняться до тех пор, пока вычисление элемента *условие* дает истинный результат. Как только *условие* станет ложным, выполнение программы продолжится с инструкции, следующей за циклом *for*.

Например, следующая программа с помощью цикла for выводит на экран числа от *1* до *100*.

```
// Программа демонстрирует использование for-цикла.
#include <iostream>
using namespace std;

int main()
{
  int count;
  for(count=1; count<=100; count=count+1)
    cout << count << " ";

  return 0;
}
```

На рис. 2.1 схематично показано выполнение цикла *for* в этом примере. Как видите, сначала переменная *count* инициализируется числом *1*. При каждом повторении цикла проверяется условие *count<=100*. Если результат проверки оказывается истинным, *cout*-инструкция выводит значение переменной *count*, после чего ее содержимое увеличивается на единицу. Когда значение переменной *count* превысит *100*, проверяемое условие даст в результате *ЛОЖЬ*, и выполнение цикла прекратится.

В профессионально написанном C++-коде редко можно встретить инструкцию *count=count+1*, поскольку для инструкций такого рода в C++ предусмотрена специальная сокращенная форма: *count++*. Оператор *"++"* называется *оператором инкремента*. Он увеличивает операнд на единицу. Оператор *"++"* дополняется оператором *"--"* (*оператором декремента*), который уменьшает операнд на единицу. С помощью оператора инкремента используемую в предыдущей программе инструкцию for можно переписать следующим образом.

```
for(count=1; count<=100; count++) cout << count << " ";
```

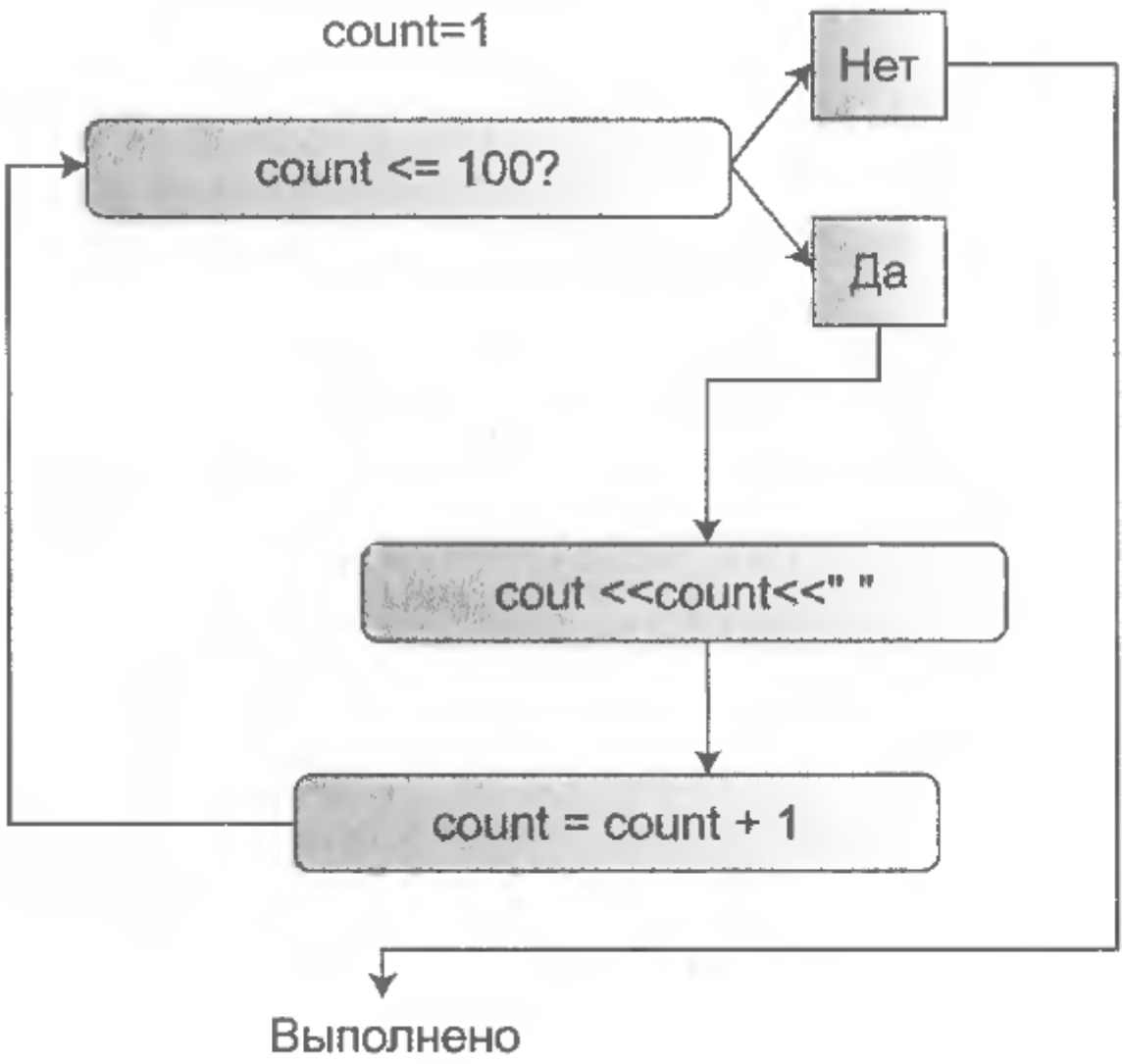


Рис. 2.1. Выполнение цикла `for`

Блоки кода

Поскольку C++ — структурированный (а также объектно-ориентированный) язык, он поддерживает создание блоков кода. *Блок* — это логически связанная группа программных инструкций, которые обрабатываются как единое целое. В C++ программный блок создается путем размещения последовательности инструкций между фигурными (открывающей и закрывающей) скобками. В следующем примере

```
if(x<10) {
  cout << "Слишком мало, попытайтесь еще раз.";
    cin >> x;
}
```

обе инструкции, расположенные после if-инструкции (между фигурными скобками) выполнятся только в том случае, если значение переменной *x* меньше *10*. Эти две инструкции (вместе с фигурными скобками) представляют блок кода. Они составляют логически неделимую группу: ни одна из этих инструкций не может выполниться без другой. С использованием блоков кода многие алгоритмы реализуются более четко и эффективно. Они также позволяют лучше понять истинную природу алгоритмов.

Блок — *это набор логически связанных инструкций*.

В следующей программе используется блок кода. Введите эту программу и выполните ее, и тогда вы поймете, как работает блок кода.

```
// Программа демонстрирует использование блока кода.
#include <iostream>
using namespace std;

int main( )
{
  int a, b;
  cout << "Введите первое число: "; cin >> a;
  cout << "Введите второе число: "; cin >> b;
  if(a < b) {
    cout << "Первое число меньше второго.\n";
    cout << "Их разность равна: " << b-a;
  }

  return 0;
}
```

Эта программа предлагает пользователю ввести два числа с клавиатуры. Если первое число меньше второго, будут выполнены обе *cout*-инструкцин. В противном случае обе они будут опущены. Ни при каких условиях не выполнится только одна из них.

Точки с запятой и расположение инструкции

В C++ точка с запятой означает конец инструкции. Другими словами, каждая отдельная инструкция должна завершаться точкой с запятой.

Как вы знаете, блок — это набор логически связанных инструкций, которые заключены между открывающей и закрывающей фигурными скобками. Блок не завершается точкой с запятой. Поскольку блок состоит из инструкций, каждая из которых завершается точкой с запятой, то в дополнительной точке с запятой нет никакого смысла. Признаком же конца блока служит закрывающая фигурная скобка (зачем еще один признак?).

Язык C++ не воспринимает конец строки в качестве признака конца инструкции. По-этому для компилятора не имеет значения, в каком месте строки располагается инструкция. Например, с точки зрения C++-компилятора, следующий фрагмент кода

```
x = y;
y = y+1;
mul(x, y);
```

аналогичен такой строке:

```
x = y; y = y+1; mul(x, y);
```

Практика отступов

Рассматривая предыдущие примеры, вы, вероятно, заметили, что некоторые инструкции сдвинуты относительно левого края. C++ — язык свободной формы, т.е. его синтаксис не связан позиционными или форматными ограничениями. Это означает, что для C++-компилятора не важно, как будут расположены инструкции по отношению друг к другу. Но у программистов с годами выработался стиль применения отступов, который значительно повышает читабельность программ. В этой книге мы придерживаемся этого стиля и вам советуем поступать так же. Согласно этому стилю после каждой открывающей скобки делается очередной отступ вправо, а после каждой закрывающей скобки на-чало отступа возвращается к прежнему уровню. Существуют также некоторые определенные инструкции, для которых предусматриваются дополнительные отступы (о них речь впереди).

Ключевые слова C++

В стандарте C++ определено 63 ключевых слова. Они показаны в табл. 2.1. Эти ключевые слова (в сочетании с синтаксисом операторов и разделителей) образуют определение языка C++. В ранних версиях C++ определено ключевое слово *overload*, но теперь оно устарело.

Таблица 2.1. Ключевые слова C++

asm	auto	bool	break
case	catch	char	class
const	const_class	continue	default
delete	do	double	dinamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Следует иметь в виду, что в C++ различается строчное и прописное написание букв. Ключевые слова не являются исключением, т.е. все они должны быть написаны строчными буквами. Например, слово *RETURN* не будет распознано в качестве ключевого слова *return*.

Идентификаторы в C++

В C++ идентификатор представляет собой имя, которое присваивается функции переменной или иному элементу, определенному пользователем. Идентификаторы могут состоять из одного или нескольких символов (значимыми должны быть первые символа). Имена переменных должны начинаться с буквы или символа подчеркивания Последующим символом может быть буква, цифра и символ подчеркивания. Символы подчеркивания можно использовать для улучшения читабельности имени переменной например *first_name*. В C++ прописные и строчные буквы воспринимаются как личные символы, т.е. *myvar* и *MyVar* — это разные имена. Вот несколько примеров допустимых идентификаторов.

```
first          x          Addr1          MaxLoad
name23         top        my_var         sample23
```

В C++ нельзя использовать в качестве идентификаторов ключевые слова. Нельзя же использовать в качестве идентификаторов имена стандартных функций (например abs). Помните, что идентификатор не должен начинаться с цифры. Так, *12x*— недопустимый идентификатор. Конечно, вы вольны называть переменные и другие программные элементы по своему усмотрению, но обычно идентификатор отражает н чение или смысловую характеристику элемента, которому он принадлежит.

Стандартная библиотека C++

В примерах программ, представленных в этой главе, использовалась функция *abs()*. По существу функция *abs()* не является частью языка C++, но ее "знает" каждый C++-компилятор. Эта функция, как и множество других, входит в состав *стандартной библиотеки*. В примерах этой книги мы подробно рассмотрим использование многих библиотечных функций C++.

Стандартная библиотека C++ содержит множество встроенных функций, которые программисты могут использовать в своих программах.

В C++ определен довольно большой набор функций, которые содержатся в стандартной библиотеке. Эти функции предназначены для выполнения часто встречающихся задач, включая операции ввода-вывода, математические вычисления и обработку строк. При использовании программистом библиотечной функции компилятор автоматически связывает объектный код этой функции с объектным кодом программы.

Поскольку стандартная библиотека C++ довольно велика, в ней можно найти много полезных функций, которыми действительно часто пользуются программисты. Библиотечные функции можно применять подобно строительным блокам, из которых возводится здание. Чтобы не "изобретать велосипед", ознакомьтесь с документацией на библиотеку используемого вами компилятора. Если вы сами напишете функцию, которая будет "переходить" с вами от программы в программу, ее также можно поместить в библиотеку.

Помимо библиотеки функций, каждый C++-компилятор также содержит *библиотеку классов*, которая является объектно-ориентированной библиотекой. Наконец, в C++ определена стандартная библиотека шаблонов (Standard Template Library— STL). Она предоставляет процедуры "многократного использования", которые можно настраивать в соответствии с конкретными требованиями. Но прежде чем применять библиотеку классов или STL, нам необходимо познакомиться с классами, объектами и понять, в чем состоит суть шаблона.

Глава 3: Основные типы данных

Как вы узнали из главы 2, все переменные в C++ должны быть объявлены до их использования. Это необходимо для компилятора, которому нужно иметь информацию о типе данных, содержащихся в переменных. Только в этом случае компилятор сможет надлежащим образом скомпилировать инструкции, в которых используются переменные. В C++ определено семь основных типов данных: *символьный*, *символьный двубайтовый*, *целочисленный*, *с плавающей точкой*, *с плавающей точкой двойной точности*, *логический* (или булев) и *"не имеющий значения"*. Для объявления переменных этих типов используются ключевые слова *char*, *wchar_t*, *int*, *float*, *double*, *bool* и *void* соответственно. Типичные размеры значений в битах и диапазоны представления для каждого из этих семи типов приведены в табл. 3.1. Помните, что размеры и диапазоны, используемые вашим компилятором, могут отличаться от приведенных здесь. Самое большое различие существует между 16- и 32-разрядными средами: для представления целочисленного значения в 16-разрядной среде используется, как правило, 16 бит, а в 32-разрядной — 32.

Переменные типа *char* используются для хранения 8-разрядных *ASCII*-символов (например

Переменные типа *char* используются для хранения 8-разрядных ASCII-символов (например букв А, Б или В) либо любых других 8-разрядных значений. Чтобы задать символ, необходимо заключить его в одинарные кавычки. Тип *wchar_t* предназначен для хранения символов, входящих в состав больших символьных наборов. Вероятно, вам известно, что в некоторых естественных языках (например китайском) определено очень большое количество символов, для которых 8-разрядное представление (обеспечиваемое типом *char*) весьма недостаточно. Для решения проблем такого рода в язык C++ и был добавлен тип *wchar_t*, который вам пригодится, если вы планируете выходить со своими программами на международный рынок.

Переменные типа *int* позволяют хранить целочисленные значения (не содержащие дробных компонентов). Переменные этого типа часто используются для управления циклами и в условных инструкциях. К переменным типа *float* и *double* обращаются либо для обработки чисел с дробной частью, либо при необходимости выполнения операций над очень большими или очень малыми числами. Типы *float* и *double* различаются значением наибольшего (и наименьшего) числа, которые можно хранить с помощью переменных этих типов. Как показано в табл. 3.1, тип *double* в C++ позволяет хранить число, приблизительно в десять раз превышающее значение типа *float*.

Таблица 3.1. Основные типы данных в C++. Типичные размеры значений и диапазоны представления

Тип	Размер в битах	Диапазон
`char`	8	-127-127 или 0-255
`wchar_t`	16	0-65 535
`int` (16-разрядная среда)	16	-32 768-32 767
`int` (32-разрядная среда)	32	-2 147 483 648-2 147 483 647
`float`	32	3,4E-38-3,4E+38
`double`	64	1,7E-308-1,7E+308
`bool`	-	`true` или `false`
`void`	-	Без значения

Тип *bool* предназначен для хранения булевых (т.е. ИСТИНА/ЛОЖЬ) значений. В C++ определены две булевы константы: *true* и *false*, являющиеся единственными значениями, которые могут иметь переменные типа *bool*.

Как вы уже видели, тип *void* используется для объявления функции, которая не возвращает значения. Другие возможности использования типа *void* рассматриваются ниже в этой книге.

Объявление переменных

Общий формат инструкции объявления переменных выглядит так:

```
тип список_переменных;
```

Здесь элемент *mun* означает допустимый в C++ тип данных, а элемент *список_переменных* может состоять из одного или нескольких имен (идентификаторов), разделенных запятыми. Вот несколько примеров объявлений переменных.

```
int i, j, k;
char ch, chr;
float f, balance;
double d;
```

В C++ имя переменной никак не связано с ее типом.

Согласно стандарту C++ первые 1024 символа любого имени (в том числе и имени переменной) являются значимыми. Это означает, что если два имени различаются хотя бы одним символом из первых 1024, компилятор будет рассматривать их как различные имена.

Переменные могут быть объявлены внутри функций, в определении параметров функций и вне всех функций. В зависимости от места объявления они называются *локальными переменными*, *формальными параметрами* и *глобальными переменными* соответственно. О важности этих трех типов переменных мы поговорим ниже в этой книге, а пока кратко рассмотрим каждый тип в отдельности.

Локальные переменные

Переменные, которые объявляются внутри функции, называются *локальными*. Их могут использовать только инструкции, относящиеся к телу функции. Локальные переменные неизвестны внешним функциям. Рассмотрим пример.

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

void func();
int main()
{
  int x; // Локальная переменная для функции main().
  x = 10;
  func();
  cout << "\n";
  cout << x; // Выводится число 10.

  return 0;
}
void func()
{
  int x; // Локальная переменная для функции func().
  x = -199;
  cout << x; // Выводится число -199.
}
```

Локальная переменная известна только функции, в которой она определена.

В этой программе целочисленная переменная с именем *x* объявлена дважды: сначала в функции *main()*, а затем в функции *func()*. Но переменная *x* из функции *main()* не имеет никакого отношения к переменной *x* из функции *func()*. Другими словами, изменения, которым подвергается переменная *x* из функции *func()*, никак не отражаются на переменной *x* из функции *main()*. Поэтому приведенная выше программа выведет на экран числа *-199* и *10*.

В C++ локальные переменные создаются при вызове функции и разрушаются при выходе из нее. То же самое можно сказать и о памяти, выделяемой для локальных переменных: при вызове функции в нее записываются соответствующие значения, а при выходе из функции память освобождается. Это означает, что локальные переменные не поддерживают своих значений между вызовами функций. (Другими словами, значение локальной переменной теряется при каждом возврате из функции.)

В некоторых литературных источниках, посвященных C++, локальная переменная называется *динамической* или *автоматической переменной*. Но в этой книге мы будем придерживаться более распространенного термина *локальная переменная*.

Формальные параметры

Формальный параметр — *это локальная переменная, которая получает значение*

Формальный параметр — *это локальная переменная, которая получает значение аргумента, переданного функции.*

Как отмечалось в главе 2, если функция имеет аргументы, то они должны быть объявлены. Их объявление осуществляется с помощью *формальных параметров*. Как показано в следующем фрагменте, формальные параметры объявляются после имени функции, внутри круглых скобок.

```
int func1 (int first, int last, char ch)
{
  .
  .
  .
}
```

Здесь функция *func1()* имеет три параметра с именами *first*, *last* и *ch*. С помощью такого объявления мы сообщаем компилятору тип каждой из переменных, которые будут принимать значения, передаваемые функции. Несмотря на то что формальные параметры выполняют специальную задачу получения значений аргументов, передаваемых функции, их можно также использовать в теле функции как обычные локальные переменные. Например, мы можем присвоить им любые значения или использовать в ка-ких-нибудь (допустимых для C++) выражениях. Но, подобно любым другим локальным переменным, их значения теряются по завершении функции.

Глобальные переменные

Глобальные переменные известны всей программе.

Чтобы придать переменной "всепрограммную" известность, ее необходимо сделать глобальной. В отличие от локальных, *глобальные переменные* хранят свои значения на протяжении всего времени жизни (времени существования) программы. Чтобы создать глобальную переменную, ее необходимо объявить вне всех функций. Доступ к глобальной переменной можно получить из любой функции.

В следующей программе переменная *count* объявляется вне всех функций. Ее объявление предшествует функции main(). Но ее с таким же успехом можно разместить в другом месте, главное, чтобы она не принадлежала какой-нибудь функции. Помните: поскольку переменную необходимо объявить до ее использования, *глобальные переменные* лучше всего объявлять в начале программы.

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

void func1();
void func2();
int count; // Это глобальная переменная.
int main()
{
  int i; // Это локальная переменная.
  for(i=0; i<10; i++){
    count = i * 2;
    func1();
  }
  return 0;
}

void func1()
```

```
void func1()
{
  cout << "count: " << count; // Обращение к глобальной переменной.
  cout << '\n'; // Вывод символа новой строки.
  func2();
}

void func2()
{
  int count; // Это локальная переменная.
  for(count=0; count<3; count++) cout <<'.';
}
```

Несмотря на то что переменная *count* не объявляется ни в функции *main()*, ни в функции *func1()*, обе они могут ее использовать. Но в функции *func2()* объявляется локальная переменная count. Здесь при обращении к переменной *count* выполняется доступ к локальной, а не к глобальной переменной. Важно помнить, что, если глобальная и локальная переменные имеют одинаковые имена, все ссылки на "спорное" имя переменной внутри функции, в которой определена локальная переменная, относятся к локальной, а не к глобальной переменной.

Модификаторы типов

В C++ перед такими типами данных, как *char*, *int* и *double*, разрешается использовать *модификаторы*. Модификатор служит для изменения значения базового типа, чтобы он более точно соответствовал конкретной ситуации. Перечислим возможные модификаторы типов.

`signed`

`unsigned`

`long`

`short`

Модификаторы *signed*, *unsigned*, *long* и *short* можно применять к целочисленным базовым типам. Кроме того, модификаторы *signed* и *unsigned* можно использовать с типом *char*, а модификатор *long*— с типом *double*. Все допустимые комбинации базовых типов и модификаторов для 16- и 32-разрядных сред приведены в табл. 3.2 и 3.3. В этих таблицах также указаны типичные размеры значений в битах и диапазоны представления для каждого типа. Безусловно, реальные диапазоны, поддерживаемые вашим компилятором, следует уточнить в соответствующей документации.

Изучая эти таблицы, обратите внимание на количество битов, выделяемых для хранения коротких, длинных и обычных целочисленных значений. Заметьте: в большинстве 16-разрядных сред размер (в битах) обычного целочисленного значения совпадает с размером короткого целого. Также отметьте, что в большинстве 32-разрядных сред размер (в битах) обычного целочисленного значения совпадает с размером длинного целого. "Собака зарыта" в C++-определении базовых типов. Согласно стандарту C++ размер длинного целого должен быть не меньше размера обычного целочисленного значения, а размер обычного целочисленного значения должен быть не меньше размера короткого целого. Размер обычного целочисленного значения должен зависеть от среды выполнения. Это значит, что в 16-разрядных средах для хранения значений типа *int* используется 16 бит, а в 32-разрядных — 32. При этом наименьший допустимый размер для целочисленных значений в любой среде должен составлять 16 бит. Поскольку стандарт C++ определяет только относительные требования к размеру целочисленных типов, нет гарантии, что один тип будет больше (по количеству битов), чем другой. Тем не менее размеры, указанные в

обеих таблицах, справедливы для многих компиляторов.

Таблица 3.2. Все допустимые комбинации базовых типов и модификаторов для 16-разрядной среды

Тип	Размер в битах	Диапазон
`char`	8	-128-127
`unsigned char`	8	0-255
`signed char`	8	-128-127
`int`	16	-32 768-32 767
`unsigned int`	16	0-65 535
`signed int`	16	Аналогичен типу `int`
`short int`	16	Аналогичен типу `int`
`unsigned short int`	16	Аналогичен типу `unsigned int`
`signed short int`	16	Аналогичен типу `short int`
`long int`	32	-2 147 483 648-2 147 483 647
`signed long int`	32	Аналогичен типу `long int`
`unsigned long int`	32	0-4 294 967 295
`float`	32	3,4E-38-3,4E+38
`double`	64	1,7E-308-1,7E+308
`long double`	80	3,4E-4932-1,1E+4932

Несмотря на разрешение, использование модификатора *signed* для целочисленных типов избыточно, поскольку объявление по умолчанию предполагает значение со знаком. Строго говоря, только конкретная реализация определяет, каким будет *char*-объявление: со знаком или без него. Но для большинства компиляторов объявление типа *char* подразумевает значение со знаком. Следовательно, в таких средах использование модификатора *signed* для *char*-объявления также избыточно. В этой книге предполагается, что *char*-значения имеют знак.

Таблица 3.3. Все допустимые комбинации базовых типов и модификаторов для 32-разрядной среды

Тип	Размер в битах	Диапазон
`char`	8	-128-127
`unsigned char`	8	0-255
`signed char`	8	-128-127
`int`	32	-2 147 483 648-2 147 483 647
`unsigned int`	32	0-4 294 967 295
`signed int`	32	Аналогичен типу `int`
`short int`	16	-32 768-32 767
`unsigned short int`	16	0-65 535
`signed short int`	16	-32 768-32 767

Окончание табл. 3.3

Тип	Размер в битах	Диапазон
`long int`	32	Аналогичен типу `int`
`signed long int`	32	Аналогичен типу `signed int`
`unsigned long int`	32	Аналогичен типу `unsigned int`

float	32	3,4E-38-3,4E+38
double	64	1,7E-308-1,7E+308
long double	80	3,4E-4932-1,1E+4932

Различие между целочисленными значениями со знаком и без него заключается в интерпретации старшего разряда. Если задано целочисленное значение со знаком, C++-компилятор сгенерирует код с учетом того, что старший разряд значения используется в качестве *флага знака*. Если флаг знака равен *0*, число считается положительным, а если он равен *1*, — отрицательным. Отрицательные числа почти всегда представляются в *дополнительном коде*. Для получения дополнительного кода все разряды числа берутся в обратном коде, а затем полученный результат увеличивается на единицу.

Целочисленные значения со знаком используются во многих алгоритмах, но максимальное число, которое можно представить со знаком, составляет только половину от максимального числа, которое можно представить без знака. Рассмотрим, например, максимально возможное 16-разрядное целое число (*32 767*):

0 1111111 11111111

Если бы старший разряд этого значения со знаком был установлен равным *1*, то оно бы интерпретировалось как *-1* (в дополнительном коде). Но если объявить его как *unsigned int*-значение, то после установки его старшего разряда в *1* мы получили бы число *65 535*.

Чтобы понять различие в C++-интерпретации целочисленных значений со знаком и без него, выполним следующую короткую программу.

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

/* Эта программа демонстрирует различие между signed- и unsigned-
значениями целочисленного типа.
*/
int main( )
{
  short int i; // короткое int-значение со знаком
  short unsigned int j; // короткое int-значение без знака
  j = 60000;
  i = j;
  cout << i << " " << j;

  return 0;
}
```

При выполнении программа выведет два числа:

```
-5536 60000
```

Дело в том, что битовая комбинация, которая представляет число *60000* как короткое целочисленное значение без знака, интерпретируется в качестве короткого *int*-значения со знаком как число *-5536*.

В C++ предусмотрен сокращенный способ объявления *unsigned*-, *short*- и *long*-значений целочисленного типа. Это значит, что при объявлении *int*-значений достаточно использовать слова *unsigned*, *short* и *long*, не указывая тип *int*, т.е. тип *int* подразумевается. Например, следующие две инструкции объявляют целочисленные переменные без знака.

```
unsigned x;
```

```
unsigned int y;
```

Переменные типа *char* можно использовать не только для хранения *ASCII*-символов, но и для хранения числовых значений. Переменные типа *char* могут содержать "небольшие" целые числа в диапазоне *-128--127* и поэтому их можно использовать вместо *int*-переменных, если вас устраивает такой диапазон представления чисел. Например, в следующей программе *char*-переменная используется для управления циклом, который выводит на экран алфавит английского языка.

```
// Эта программа выводит алфавит в обратном порядке.
#include <iostream>
using namespace std;

int main()
{
  char letter;
  for(letter='Z'; letter >= 'A'; letter--) cout << letter;

  return 0;
}
```

Если цикл *for* вам покажется несколько странным, то учтите, что символ *'A'* представляется в компьютере как число, а значения от *'Z'* до *'A'* являются последовательными и расположены в убывающем порядке.

Литералы

Литералы, называемые также *константами*, — это фиксированные значения, которые не могут быть изменены программой. Мы уже использовали литералы во всех предыдущих примерах программ. А теперь настало время изучить их более подробно.

Константы могут иметь любой базовый тип данных. Способ представления каждой константы зависит от ее типа. Символьные константы заключаются в одинарные кавычки. Например, *'a'* и *'%'* являются символьными литералами. Если необходимо присвоить символ переменной типа *char*, используйте инструкцию, подобную следующей:

```
ch = 'Z';
```

Чтобы использовать двубайтовый символьный литерал (т.е. константу типа *wchar_t*), предварите нужный символ буквой *L*. Например, так.

```
wchar_t wc;
wc = L'A';
```

Здесь переменной *wc* присваивается двубайтовая символьная константа, эквивалентная букве *A*.

Целочисленные константы задаются как числа без дробной части. Например, *10* и *-100* — целочисленные литералы. Вещественные литералы должны содержать десятичную точку, за которой следует дробная часть числа, например *11.123*. Для вещественных констант можно также использовать экспоненциальное представление чисел.

Существует два основных вещественных типа: *float* и *double*. Кроме того, существует несколько модификаций базовых типов, которые образуются с помощью модификаторов типов. Интересно, а как же компилятор определяет тип литерала? Например, число *123.23* имеет тип *float* или *double*? Ответ на этот вопрос состоит из двух частей. Во-первых, C++-компилятор автоматически делает определенные предположения насчет литералов. Во-вторых, при желании программист может явно указать тип литерала.

По умолчанию компилятор связывает целочисленный литерал с совместимым и одновременно наименьшим по занимаемой памяти тип данных, начиная с типа *int*. Следовательно, для 16-разрядных сред число *10* будет связано с типом *int*, а *103 000*— с

типом *long int*.

Единственным исключением из правила "наименьшего типа" являются вещественные (с плавающей точкой) константы, которым по умолчанию присваивается тип *double*. Во многих случаях такие стандарты работы компилятора вполне приемлемы. Однако у программиста есть возможность точно определить нужный тип.

Чтобы задать точный тип числовой константы, используйте соответствующий суффикс. Для вещественных типов действуют следующие суффиксы: если вещественное число завершить буквой *F*, оно будет обрабатываться с использованием типа *float*, а если буквой *L*, подразумевается тип *long double*. Для целочисленных типов суффикс *U* означает использование модификатора типа *unsigned*, а суффикс *L— long*. (Для задания модификатора *unsigned long* необходимо указать оба суффикса *U* и *L*.) Ниже приведены некоторые примеры.

Тип данных	Примеры констант
int	1, 123, 21000, -234
long int	35000L, -34L
unsigned int	10000U, 987U, 40000U
unsigned long	12323UL, 900000UL
float	123.23F, 4.34e-3F
double	23.23, 123123.33, -0.9876324
long double	1001.2L

Шестнадцатеричные и восьмеричные литералы

Иногда удобно вместо десятичной системы счисления использовать восьмеричную или шестнадцатеричную. В *восьмеричной* системе основанием служит число 8, а для выражения всех чисел используются цифры от 0 до 7. В восьмеричной системе число 10 имеет то же значение, что число 8 в десятичной. Система счисления по основанию 16 называется *шестнадцатеричной* и использует цифры от 0 до 9 плюс буквы от A до F, означающие шестнадцатеричные "цифры" 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 равно числу 16 в десятичной системе. Поскольку эти две системы счисления (шестнадцатеричная и восьмеричная) используются в программах довольно часто, в языке C++ разрешено при желании задавать целочисленные литералы не в десятичной, а в шестнадцатеричной или восьмеричной системе. Шестнадцатеричный литерал должен начинаться с префикса *0x* (нуль и буква x) или *0X*, а восьмеричный — с нуля. Приведем два примера.

```
int hex = 0xFF; // 255 в десятичной системе
int oct = 011; // 9 в десятичной системе
```

Строковые литералы

Язык C++ поддерживает еще один встроенный тип литерала, именуемый *строковым*. *Строка*— это набор символов, заключенных в двойные кавычки, например *"это тест"*. Вы уже видели примеры строк в некоторых cout-инструкциях, с помощью которых мы выводили текст на экран. При этом обратите внимание вот на что. Хотя C++ позволяет определять строковые литералы, он не имеет встроенного строкового типа данных. Строки в C++, как будет показано ниже в этой книге, поддерживаются в виде символьных массивов. (Кроме того, стандарт C++ поддерживает строковый тип с помощью библиотечного класса *string*, который также описан ниже в этой книге.)

Осторожно! *Не следует путать строки с символами. Символьный литерал заключается в одинарные кавычки, например 'a'. Однако "a" — это уже строка, содержащая только одну букву.*

Управляющие символьные последовательности

С выводом большинства печатаемых символов прекрасно справляются символьные константы, заключенные в одинарные кавычки, но есть такие "экземпляры" (например

константы, заключенные в одинарные кавычки, но есть такие "экземпляры" (например, символ возврата каретки), которые невозможно ввести в исходный текст программы с клавиатуры. Некоторые символы (например, одинарные и двойные кавычки) в C++ имеют специальное назначение, поэтому иногда их нельзя ввести напрямую. По этой причине в языке C++ разрешено использовать ряд специальных символьных последовательностей (включающих символ "обратная косая черта"), которые также называются *управляющими последовательностями*. Их список приведен в табл. 3.4.

Использование управляющих последовательностей демонстрируется на примере следующей программы. При ее выполнении будут выведены символы перехода на новую строку, обратной косой черты и возврата на одну позицию.

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

int main( )
{
  cout<<"\n\\\b";
  return 0;
}
```

Таблица 3.4. Управляющие символьные последовательности

Код	Значение
\b	Возврат на одну позицию
\f	Подача страницы (для перехода к началу следующей страницы)
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\"	Двойная кавычка
\'	Одинарная кавычка (апостроф)
\\	Обратная косая черта
\v	Вертикальная табуляция
\a	Звуковой сигнал (звонок)
\?	Вопросительный знак
N	Восьмеричная константа (где *N* — это сама восьмеричная константа)
\x*N*	Шестнадцатеричная константа (где *N* — это сама шестнадцатеричная константа)

Инициализация переменных

При объявлении переменной ей можно присвоить некоторое значение, т.е. инициализировать ее, записав после ее имени знак равенства и начальное значение. Общий формат инициализации имеет следующий вид:

```
тип имя_переменной = значение;
```

Вот несколько примеров.

```
char ch = 'a';
int first = 0;
float balance = 123.23F;
```

Несмотря на то что переменные часто инициализируются константами, C++ позволяет инициализировать переменные динамически, т.е. с помощью любого выражения, действительного на момент инициализации. Как будет показано ниже, инициализация играет важную роль при работе с объектами.

Глобальные переменные инициализируются только в начале программы. Локальные переменные инициализируются при каждом входе в функцию, в которой они объявлены. Все глобальные переменные инициализируются нулевыми значениями, если не указаны никакие иные инициализаторы. Неинициализированные локальные переменные будут иметь неизвестные значения до первой инструкции присваивания, в которой они используются.

Рассмотрим простой пример инициализации переменных. В следующей программе используется функция *total()*, которая предназначена для вычисления суммы всех последовательных чисел, начиная с единицы и заканчивая числом, переданным ей в качестве аргумента. Например, сумма ряда чисел, ограниченного числом *3*, равна *1 + 2 + 3 = 6*. В процессе вычисления итоговой суммы функция *total()* отображает промежуточные результаты. Обратите внимание на использование переменной *sum* в функции *total()*.

```
// Пример использования инициализации переменных.
#include <iostream>
using namespace std;

void total(int x);
int main( )
{
  cout << "Вычисление суммы чисел от 1 до 5.\n";
  total(5);
  cout << "\n Вычисление суммы чисел от 1 до 6.\n";
  total(6);

  return 0;
}

void total(int x)
{
  int sum=0; // Инициализируем переменную sum.
  int i, count;
  for(i=1; i<=x; i++) {
    sum = sum + i;
    for(count=0; count<10; count++) cout << '.';
    cout << "Промежуточная сумма равна " << sum << '\n';
  }
}
```

Результаты выполнения этой программы таковы.

```
Вычисление суммы чисел от 1 до 5.
.........Промежуточная сумма равна 1
.........Промежуточная сумма равна 3
.........Промежуточная сумма равна 6
.........Промежуточная сумма равна 10
.........Промежуточная сумма равна 15
Вычисление суммы чисел от 1 до 6.
```

```
.........Промежуточная сумма равна 1
.........Промежуточная сумма равна 3
.........Промежуточная сумма равна 6
.........Промежуточная сумма равна 10
.........Промежуточная сумма равна 15
.........Промежуточная сумма равна 21
```

Как видно по результатам, при каждом вызове функции *total()* переменная *sum* инициализируется нулем.

Операторы

В C++ определен широкий набор встроенных операторов, которые дают в руки программисту мощные рычаги управления при создании и вычислении разнообразнейших выражений. *Оператор* (operator) — это символ, который указывает компилятору на выполнение конкретных математических действий или логических манипуляций. В C++ имеется четыре общих класса операторов: *арифметические*, *поразрядные*, *логические* и *операторы отношений*. Помимо них определены другие операторы специального назначения. В этой главе рассматриваются арифметические, логические и операторы отношений.

Арифметические операторы

В табл. 3.5 перечислены арифметические операторы, разрешенные для применения в C++. Действие операторов *+*, *-*, * и */* совпадает с действием аналогичных операторов в любом другом языке программирования (да и в алгебре, если уж на то пошло). Их можно применять к данным любого встроенного числового типа. После применения оператора деления (*/*) к целому числу остаток будет отброшен. Например, результат целочисленного деления *10/3* будет равен *3*.

Таблица 3.5. Арифметические операторы

Оператор	Действие
+	Сложение
–	Вычитание, а также унарный минус
*	Умножение
/	Деление
%	Деление по модулю
--	Декремент
++	Инкремент

Остаток от деления можно получить с помощью *оператора деления по модулю* (*%*). Этот оператор работает практически так же, как в других языках программирования: возвращает остаток от деления нацело. Например, *10%3* равно *1*. Это означает, что в C++ оператор *"%"* нельзя применять к типам с плавающей точкой (float или double). Деление по модулю применимо только к целочисленным типам. Использование этого оператора демонстрируется в следующей программе.

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

int main( )
{
  int x, y;
  x = 10;
  y = 3;
```

```
  cout << x/y; // Будет отображено число 3.
  cout << "\n";
   cout << x%y; /* Будет отображено число 1, т.е. остаток от деления нацело. */
  cout << "\n";
  x = 1;
  y = 2;
  cout << x/y << " " << x%y; // Будут выведены числа 0 и 1.
  return 0;
}
```

В последней строке результатов выполнения этой программы действительно будут выведены числа *0* и *1*, поскольку при целочисленном делении *1/2* получим *0* с остатком *1*, т.е. выражение *1%2* дает значение *1*.

Унарный минус, по сути, представляет собой умножение значения своего единственного операнда на *-1*. Другими словами, любое числовое значение, которому предшествует знак меняет свой знак на противоположный.

Инкремент и декремент

В C++ есть два оператора, которых нет в некоторых других языках программирования. Это операторы инкремента (*++*) и декремента (--). Они упоминались в главе 2, когда речь шла о цикле for. Оператор инкремента выполняет сложение операнда с числом 1, а оператор декремента вычитает 1 из своего операнда. Это значит, что инструкция

```
x = x + 1;
```

аналогична такой инструкции:

```
++x;
```

А инструкция

```
x = x - 1;
```

аналогична такой инструкции:

```
--x;
```

Операторы инкремента и декремента могут стоять как перед своим операндом (*префиксная форма*), так и после него (*постфиксная форма*). Например, инструкцию

```
x = x + 1;
```

можно переписать в виде префиксной формы

```
++x; // Префиксная форма оператора инкремента.
```

или в виде постфиксной формы:

```
x++; // Постфиксная форма оператора инкремента.
```

В предыдущем примере не имело значения, в какой форме был применен оператор инкремента: префиксной или постфиксной. Но если оператор инкремента или декремента используется как часть большего выражения, то форма его применения очень важна. Если такой оператор применен в префиксной форме, то C++ сначала выполнит эту операцию, чтобы операнд получил новое значение, которое затем будет использовано остальной частью выражения. Если же оператор применен в постфиксной форме, то C++ использует в выражении его старое значение, а затем выполнит операцию, в результате которой операнд обретет новое значение. Рассмотрим следующий фрагмент кода:

```
x = 10;
```

```
y = ++x;
```

В этом случае переменная *y* будет установлена равной *11*. Но если в этом коде префиксную форму записи заменить постфиксной, переменная *y* будет установлена равной *10*:

```
x = 10;
y = x++;
```

В обоих случаях переменная *x* получит значение *11*. Разница состоит лишь в том, в какой момент она станет равной *11* (до присвоения ее значения переменной y или после). Для программиста очень важно иметь возможность управлять временем выполнения операции инкремента или декремента.

Большинство C++-компиляторов для операций инкремента и декремента создают более эффективный код по сравнению с кодом, сгенерированным при использовании обычного оператора сложения и вычитания единицы. Поэтому профессионалы предпочитают использовать (где это возможно) операторы инкремента и декремента.

Арифметические операторы подчиняются следующему порядку выполнения действий.

Приоритет	Операторы
Наивысший	++ --
	- (унарный минус)
	* / %
Низший	+ -

Операторы одного уровня старшинства вычисляются компилятором слева направо. Безусловно, для изменения порядка вычислений можно использовать круглые скобки, которые обрабатываются в C++ так же, как практически во всех других языках программирования. Операции или набор операций, заключенных в круглые скобки, приобретают более высокий приоритет по сравнению с другими операциями выражения.

История происхождения имени C++

Теперь, когда вам стало понятно значение оператора *"++"*, можно сделать предположения насчет происхождения имени C++. Как вы знаете, C++ построен на фундаменте языка C, к которому добавлено множество усовершенствований, большинство из которых предназначены для поддержки объектно-ориентированного программирования. Таким образом, C++ представляет собой *инкрементное* усовершенствование языка C, а результат добавления символов *"++"* (оператора инкремента) к имени C оказался вполне подходящим именем для нового языка.

Бьерн Страуструп сначала назвал свой язык "C с классами" (C with Classes), но, по предложению Рика Маскитти (Rick Mascitti), он позже изменил это название на C++. И хотя успех нового языка еще только предполагался, принятие нового названия (C++) практически гарантировало ему видное место в истории, поскольку это имя было узнаваемым для каждого C-программиста.

Операторы отношений и логические операторы

Операторы отношений и логические (булевы) операторы, которые часто идут "рука об руку", используются для получения результатов в виде значений *ИСТИНА*/*ЛОЖЬ*. Операторы отношений оценивают по "двубалльной системе" отношения между двумя значениями, а логические определяют различные способы сочетания истинных и ложных значений. Поскольку операторы отношений генерируют ИСТИНА/ЛОЖЬ-результаты, то они часто выполняются с логическими операторами. Поэтому мы и рассматриваем их в одном разделе.

Операторы отношений и логические (булевы) операторы перечислены в табл. 3.6. Обратите

внимание на то, что в языке C++ в качестве оператора отношения "не равно" используется символ *"!="*, а для оператора "равно" — двойной символ равенства (==). Согласно стандарту C++ результат выполнения операторов отношений и логических операторов имеет тип *bool*, т.е. при выполнении операций отношений и логических операций получаются значения *true* или *false*. При использовании более старых компиляторов результаты выполнения этих операций имели тип int (нуль или ненулевое целое, например 1). Это различие в интерпретации значений имеет в основном теоретическую основу, поскольку C++ автоматически преобразует значение *true* в *1*, а значение *false* — в *0*, и наоборот.

Операнды, участвующие в операциях "выяснения" отношений, могут иметь практически любой тип, главное, чтобы их можно было сравнивать. Что касается логических операторов, то их операнды должны иметь тип *bool*, и результат логической операции всегда будет иметь тип *bool*. Поскольку в C++ *любое ненулевое число* оценивается как истинное (true), а нуль эквивалентен ложному значению (false), то логические операторы можно использовать в любом выражении, которое дает нулевой или ненулевой результат.

Таблица 3.6. Операторы отношений и логические операторы

Операторы отношений	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
Логические операторы	**Значение**
&&	И
||	ИЛИ
!	НЕ

Помните, что в C++ любое ненулевое число оценивается как true, а нуль— как false.

Логические операторы используются для поддержки базовых логических операций *И*, *ИЛИ* и *НЕ* в соответствии со следующей таблицей истинности. Здесь *1* используется как значение ИСТИНА, а *0* — как значение ЛОЖЬ.

p	q	p И q	p ИЛИ q	НЕ p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Несмотря на то что C++ не содержит встроенный логический оператор *"исключающее ИЛИ"* (XOR), его нетрудно "создать" на основе встроенных. Посмотрите, как следующая функция использует операторы *И*, *ИЛИ* и *НЕ* для выполнения операции *"исключающее ИЛИ"*.

```
bool xor(bool a, bool b)
{
  return (a || b) && !(a && b);
}
```

Эта функция используется в следующей программе. Она отображает результаты применения операторов *И*, *ИЛИ* и *"исключающее ИЛИ"* к вводимым вами же значениям.

(Помните, что здесь единица будет обработана как значение true, а нуль — как false.)

```
// Эта программа демонстрирует использование функции xor().
#include <iostream>
using namespace std;

bool xor(bool a, bool b);
int main( )
{
  bool p, q;
  cout << "Введите P (0 или 1): ";
    cin >> p;
  cout << "Введите Q (0 или 1): ";
    cin >> q;
  cout << "P И Q: " << (p && q) << ' \n';
  cout << "P ИЛИ Q: " << (p || q) << ' \n';
  cout << "P XOR Q: " << xor(p, q) << '\n';
  return 0;
}

bool xor(bool a, bool b)
{
  return (a || b) && !(a && b);
}
```

Вот как выглядит возможный результат выполнения этой программы.

Введите P (0 или 1): 1

Введите Q (0 или 1): 1

P И Q: 1

P ИЛИ Q: 1

P XOR Q: 0

В этой программе обратите внимание вот на что. Хотя параметры функции *xor()* указаны с типом *bool*, пользователем вводятся целочисленные значения (0 или 1). В этом ничего нет странного, поскольку C++ автоматически преобразует число *1* в *true*, а *0* в *false*. И наоборот, при выводе на экран *bool*-значения, возвращаемого функцией *xor()*, оно автоматически преобразуется в число *0* или *1* (в зависимости от того, какое значение "вернулось": false или true). Интересно отметить, что, если типы параметров функции *xor()* и тип возвращаемого ею значения заменить типом int, эта функция будет работать абсолютно так же. Причина проста: все дело в автоматических преобразованиях, выполняемых C++-компилятором между целочисленными и булевыми значениями.

Как операторы отношений, так и логические операторы имеют более низкий приоритет по сравнению с арифметическими операторами. Это означает, что такое выражение, как *10 > 1+12* будет вычислено так, как если бы оно было записано в таком виде: *10 >(1 + 12)*

Результат этого выражения, конечно же, равен значению *ЛОЖЬ*. Кроме того, взгляните еще раз на инструкции вывода результатов работы предыдущей программы на экран.

```
cout << "P И Q: " << (p && q) << '\n';
cout << "P ИЛИ Q: " << (p | | q) << '\n';
```

Без круглых скобок, в которые заключены выражения *p && q* и *p || q*, здесь обойтись нельзя,

поскольку операторы && и || имеют более низкий приоритет, чем оператор вывода данных.

С помощью логических операторов можно объединить в одном выражении любое количество операций отношений. Например, в этом выражении объединено сразу три операции отношений.

```
var>15 || !(10<count) && 3<=item
```

Приоритет операторов отношений и логических операторов показан в следующей таблице.

Приоритет	Операторы
Наивысший	!
	> >= < <=
	== !=
	&&
Низший	||

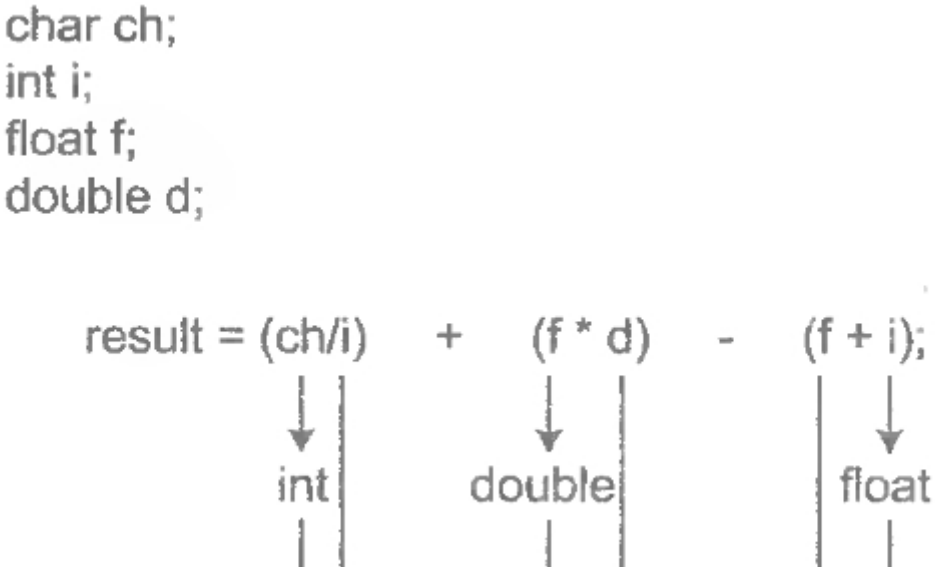
Выражения

Операторы, литералы и переменные — это все составляющие *выражений*. Вероятно, вы уже знакомы с выражениями по предыдущему опыту программирования или из школьного курса алгебры. В следующих разделах мы рассмотрим аспекты выражений, которые касаются их использования в языке C++.

Преобразование типов в выражениях

Если в выражении смешаны различные типы литералов и переменных, компилятор преобразует их к одному типу. Во-первых, все *char*- и *short int*-значения автоматически преобразуются (с расширением "типоразмера") к типу *int*. Этот процесс называется *целочисленным расширением* (integral promotion). Во-вторых, все операнды преобразуются (также с расширением "типоразмера") к типу самого большого операнда. Этот процесс называется *расширением типа* (type promotion), причем он выполняется по операционно. Например, если один операнд имеет тип *int*, а другой — *long int*, то тип *int* расширяется в тип *long int*. Или, если хотя бы один из операндов имеет тип *double*, любой другой операнд приводится к типу *double*. Это означает, что такие преобразования, как из типа *char* в тип *double*, вполне допустимы. После преобразования оба операнда будут иметь один и тот же тип, а результат операции — тип, совпадающий с типом операндов.

Рассмотрим, например, преобразование типов, схематически представленное на рис. 3.1. Сначала символ *ch* подвергается процессу "расширения" типа и преобразуется в значение типа *int*. Затем результат операции *ch/i* приводится к типу *double*, поскольку результат произведения *f*d* имеет тип *double*. Результат всего выражения получит тип *double*, поскольку к моменту его вычисления оба операнда будут иметь тип *double*.



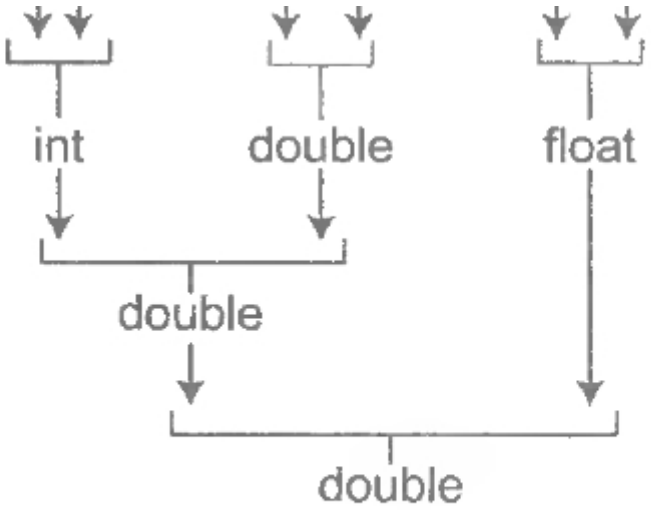


Рис. 3.1. Пример преобразования типов в C++

Преобразования, связанные с типом bool

Как упоминалось выше, значения типа bool автоматически преобразуются в целые числа 0 или 1 при использовании в выражении целочисленного типа. При преобразовании целочисленного результата в тип bool нуль преобразуется в false, а ненулевое значение — в true. И хотя тип bool относительно недавно был добавлен в язык C++, выполнение автоматических преобразований, связанных с типом bool, означает, что его введение в C++ не имеет негативных последствий для кода, написанного для более ранних версий C++. Более того, автоматические преобразования позволяют C++ поддерживать исходное определение значений ЛОЖЬ и ИСТИНА в виде нуля и ненулевого значения. Таким образом, тип bool очень удобен для программиста.

Приведение типов

В C++ предусмотрена возможность установить для выражения заданный тип. Для этого используется *операция приведения типов* (cast). C++ определяет пять видов таких операций. В этом разделе мы рассмотрим только один из них, а остальные четыре описаны ниже в этой книге (после темы создания объектов). Итак, общий формат операции приведения типов таков:

```
(тип) выражение
```

Здесь элемент *тип* означает тип, к которому необходимо привести *выражение*. Например, если вы хотите, чтобы выражение *x/2* имело тип *float*, необходимо написать следующее:

```
(float) x / 2
```

Приведение типов рассматривается как унарный оператор, и поэтому он имеет такой же приоритет, как и другие унарные операторы.

Иногда операция приведения типов оказывается очень полезной. Например, в следующей программе для управления циклом используется некоторая целочисленная переменная, входящая в состав выражения, результат вычисления которого необходимо получить с дробной частью.

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

int main() /* Выводим i и значение i/2 с дробной частью.*/
{
  int i;
  for(i=1; i<=100; ++i )
    cout << i << "/ 2 равно: " << (float) i / 2 << '\n';
```

```
  return 0;
}
```

Без оператора приведения типа *(float)* выполнилось бы только целочисленное деление. Приведение типов в данном случае гарантирует, что на экране будет отображена и дробная часть результата.

Использование пробелов и круглых скобок

Любое выражение в C++ для повышения читабельности может включать пробелы (или символы табуляции). Например, следующие два выражения совершенно одинаковы, но второе прочитать гораздо легче.

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Круглые скобки (так же, как в алгебре) повышают приоритет операций, содержащихся внутри них. Использование избыточных или дополнительных круглых скобок не приведет к ошибке или замедлению вычисления выражения. Другими словами, от них не будет никакого вреда, но зато сколько пользы! Ведь они помогут прояснить (для вас самих в первую очередь, не говоря уже о тех, кому придется разбираться в этом без вас) точный порядок вычислений. Скажите, например, какое из следующих двух выражений легче понять?

```
x = y/3-34*temp+127;
X = (y/3) - (34*temp) + 127;
```

Глава 4: Инструкции управления

В этой главе вы узнаете, как управлять ходом выполнения C++-программы. Существует три категории управляющих инструкций: *инструкции выбора* (if, switch), *итерационные инструкции* (состоящие из for-, while- и do-while-циклов) и *инструкции перехода* (break, continue, return и goto).

За исключением return, все остальные перечисленные выше инструкции описаны в этой главе.

Инструкция if

Инструкция if *позволяет сделать выбор между двумя выполняемыми ветвями программы*.

Инструкция *if* была представлена в главе 2, но здесь мы рассмотрим ее более детально. Полный формат ее записи таков.

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

Здесь под элементом *инструкция* понимается одна *инструкция* языка C++. Часть *else* необязательна. Вместо элемента *инструкция* может быть использован блок инструкций. В этом случае формат записи *if*-инструкции принимает такой вид.

```
if(выражение)
{
  последовательность инструкций
}
else
{
  последовательность инструкций
}
```

Если элемент *выражение*, который представляет собой условное выражение, при

Если элемент *выражение*, который представляет собой условное выражение, при вычислении даст значение *ИСТИНА*, будет выполнена *if*-инструкция; в противном случае *else*-инструкция (если таковая существует). Обе инструкции никогда не выполняются. Условное выражение, управляющее выполнением *if*-инструкции, может иметь любой тип, действительный для C++-выражений, но главное; чтобы результат его вычисления можно было интерпретировать как значение *ИСТИНА* или *ЛОЖЬ*.

Использование *if*-инструкции рассмотрим на примере программы, которая представляет собой версию игры *"Угадай магическое число"*. Программа генерирует случайное число и предлагает вам его угадать. Если вы угадываете число, программа выводит на экран сообщение одобрения ** *Правильно* **. В этой программе представлена еще одна библиотечная функция *rand()*, которая возвращает случайным образом выбранное целое число. Для использования этой функции необходимо включить в программу заголовок *<cstdlib>*.

```
// Программа "Угадай магическое число".
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int magic; // магическое число
  int guess; // вариант пользователя
  magic = rand(); // Получаем случайное число.
  cout << "Введите свой вариант магического числа: ";
    cin >> guess;
  if(guess == magic) cout << "** Правильно **";

  return 0;
}
```

В этой программе для проверки того, совпадает ли с *"магическим числом"* вариант, предложенный пользователем, используется оператор отношения *"=="*. При совпадении чисел на экран выводится сообщение ** *Правильно* **.

Попробуем усовершенствовать нашу программу и в ее новую версию включим *else*-ветвь для вывода сообщения о том, что предположение пользователя оказалось неверным.

```
// Программа "Угадай магическое число":
// 1-е усовершенствование.
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int magic; // магическое число
  int guess; // вариант пользователя
  magic = rand(); // Получаем случайное число.
  cout << "Введите свой вариант магического числа: ";
    cin >> guess;
  if(guess == magic) cout << "** Правильно **";
```

```
  else cout << "...Очень жаль, но вы ошиблись.";

  return 0;
}
```

Условное выражение

Иногда новичков в C++ сбивает с толку тот факт, что для управления if-инструкцией можно использовать любое действительное C++-выражение. Другими словами, тип выражения необязательно ограничивать операторами отношений и логическими операторами или операндами типа bool. Главное, чтобы результат вычисления условного выражения можно было интерпретировать как значение ИСТИНА или ЛОЖЬ. Как вы помните из предыдущей главы, нуль автоматически преобразуется в false, а все ненулевые значения— в true. Это означает, что любое выражение, которое дает в результате нулевое или ненулевое значение, можно использовать для управления if-инструкцией. Например, следующая программа считывает с клавиатуры два целых числа и отображает частное от деления первого на второе. Чтобы не допустить деления на нуль, в программе используется if-инструкция.

```
// Деление первого числа на второе.
#include <iostream>
using namespace std;

int main( )
{
  int a, b;
  cout << "Введите два числа: ";
  cin >> a >> b;
  if(b) cout << a/b << '\n';
  else cout << "На нуль делить нельзя.\n";

  return 0;
}
```

Обратите внимание на то, что значение переменной *b* (делимое) сравнивается с нулем с помощью инструкции *if(b)*, а не инструкции *if(b!=0)*. Дело в том, что, если значение *b* равно нулю, условное выражение, управляющее инструкцией *if*, оценивается как *ЛОЖЬ*, что приводит к выполнению *else*-ветви. В противном случае (если *b* содержит ненулевое значение) условие оценивается как *ИСТИНА*, и деление благополучно выполняется. Нет никакой необходимости использовать следующую if-инструкцию, которая к тому же не свидетельствует о хорошем стиле программирования на C++.

```
if(b != 0) cout << a/b << '\n';
```

Эта форма if-инструкции считается устаревшей и потенциально неэффективной.

Вложенные if-инструкции

Вложенные if-инструкции образуются в том случае, если в качестве элемента *инструкция* (см. полный формат записи) используется другая if-инструкция. Вложенные if-инструкции очень популярны в программировании. Главное здесь — помнить, что else-инструкция всегда относится к ближайшей if-инструкции, которая находится внутри того же программного блока, но еще не связана ни с какой другой else-инструкцией. Вот пример.

```
if(i) {
  if(j) statement1;
  if(k) statement2; // Эта if-инструкция
  else statement3; // связана с этой else-инструкцией
```

```
  else statement3; // связана с этой else-инструкцией.
}
else statement4; // Эта else-инструкция связана с if(i).
```

Как утверждается в комментариях, последняя *else*-инструкция не связана с инструкцией *if(j)*, поскольку они не находятся в одном блоке (несмотря на то, что эта if-инструкция — ближайшая, которая не имеет при себе *"else-пары"*). Внутренняя *else*-инструкция связана с инструкцией *if(k)*, поскольку она — ближайшая и находится внутри того же блока.

Вложенная if-инструкция— *это инструкция, которая используется в качестве элемента инструкция любой другой if- или else-инструкции*.

Язык C++ позволяет 256 уровней вложения, но на практике редко приходится вкладывать if-инструкции на "такую глубину". продемонстрируем использование вложенных инструкций с помощью очередного усовершенствования программы *"Угадай магическое число"* (здесь игрок получает реакцию программы на неправильный ответ).

```
// Программа "Угадай магическое число":
// 2-е усовершенствование.
#include <iostream>
#include <cstdlib>
using namespace std;

int main( )
{
  int magic; // магическое число
  int guess; // вариант пользователя
  magic = rand( ); // Получаем случайное число.
  cout << "Введите свой вариант магического числа: ";
    cin >> guess;
  if(guess == magic) {
    cout << " ** Правильно **\n";
    cout << magic << " и есть то самое магическое число.\n";
  }
  else {
    cout << "...Очень жаль, но вы ошиблись.";
    if(guess > magic) cout <<"Ваш вариант превышает магическое число.\n";
    else cout << " Ваш вариант меньше магического числа.\n";
  }
  return 0;
}
```

Конструкция if-else-if

Очень распространенной в программировании конструкцией, в основе которой лежит вложенная if-инструкция, является "лестница" *if-else-if*. Ее можно представить в следующем виде.

```
if(условие)
  инструкция;
else if(условие)
```

```
else if(условие)
  инструкция;
else if(условие)
  инструкция;
.
.
.
else
  инструкция;
```

Здесь под элементом *условие* понимается условное выражение. Условные выражения вычисляются сверху вниз. Как только в какой-нибудь ветви обнаружится истинный результат, будет выполнена инструкция, связанная с этой ветвью, а вся остальная "лестница" опускается. Если окажется, что ни одно из условий не является истинным, будет выполнена последняя else-инструкция (можно считать, что она выполняет роль условия, которое действует по умолчанию). Если последняя else-инструкция не задана, а все остальные оказались ложными, то вообще никакое действие не будет выполнено.

"Лестница" if-else-if— *это последовательность вложенных if-else-инструкций*.

Работа *if-else-if* - "лестницы" демонстрируется в следующей программе.

```
// Демонстрация использования "лестницы" if-else-if.
#include <iostream>
using namespace std;

int main( )
{
  int x;
  for(x=0; x<6; x++) {
    if(x==1) cout << "x равен единице.\n";
    else if(x==2) cout << "x равен двум.\n";
    else if(x==3) cout<< "x равен трем.\n";
    else if(x==4) cout << "x равен четырем.\n";
    else cout << "x не попадает в диапазон от 1 до 4.\n";
  }
  return 0;
}
```

Результаты выполнения этой программы таковы.

```
x не попадает в диапазон от 1 до 4.
x равен единице,
x равен двум,
x равен трем,
x равен четырем.
x не попадает в диапазон от 1 до 4.
```

Как видите, последняя else-инструкция выполняется только в том случае, если все предыдущие if-условия дали ложный результат.

Цикл for

Цикл for — *самый универсальный цикл C++.*

В главе 2 мы уже использовали простую форму цикла *for*. В этой главе мы рассмотрим этот цикл более детально, и вы узнаете, насколько мощным и гибким средством программирования он является. Начнем с традиционных форм его использования.

Итак, общий формат записи цикла *for* для многократного выполнения одной инструкции имеет следующий вид.

```
for(инициализация; выражение; инкремент) инструкция;
```

Если цикл *for* предназначен для многократного выполнения не одной инструкции, а программного блока, то его общий формат выглядит так.

```
for (инициализация; выражение; инкремент)
{
  последовательность инструкций
}
```

Элемент *инициализация* обычно представляет собой инструкцию присваивания, которая устанавливает *управляющую переменную цикла* равной начальному значению. Эта переменная действует в качестве счетчика, который управляет работой цикла. Элемент *выражение* представляет собой условное выражение, в котором тестируется значение управляющей переменной цикла. Результат этого тестирования определяет, выполнится цикл for еще раз или нет. Элемент *инкремент*— это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации. Обратите внимание на то, что все эти элементы цикла for должны отделяться точкой с запятой. Цикл for будет выполняться до тех пор, пока вычисление элемента *выражение* дает истинный результат. Как только это условное выражение станет ложным, цикл завершится, а выполнение программы продолжится с инструкции, следующей за циклом for.

В следующей программе цикл for используется для вывода значений квадратного корня, извлеченных из чисел от *1* до *99*. Обратите внимание на то, что в этом примере управляющая переменная цикла называется *num*.

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

int main( )
{
  int num;
  double sq_root;
  for(num=1; num<100; num++) {
    sq_root = sqrt((double) num);
    cout << num << " " << sq_root << '\n';
  }
  return 0;
}
```

Вот как выглядят первые строки результатов, выводимых этой программой.

```
1 1
2 1.41421
3 1.73205
4 2
```

```
4 2
5 2.23607
6 2.44949
7 2.64575
8 2.82843
9 3
10 3.16228
11 3.31662
```

В этой программе использована еще одна стандартная функция C++: *sqrt()*. Эта функция возвращает значение квадратного корня из своего аргумента. Аргумент должен иметь тип *double*, и именно поэтому при вызове функции *sqrt()* параметр *num* приводится к типу *double*. Сама функция также возвращает значение типа *double*. Обратите внимание на то, что в программу включен заголовок *<cmath>*, поскольку этот заголовочный файл обеспечивает поддержку функции *sqrt()*.

Важно! *Помимо функции sqrt(), C++ поддерживает широкий набор других математических функций, например sin(), cos(), tan(), log(), ceil() и floor(). Помните, что все математические функции требуют включения в программу заголовка <cmath>.*

Управляющая переменная цикла for может изменяться как с положительным, так и с отрицательным приращением, причем величина этого приращения также может быть любой. Например, следующая программа выводит числа в диапазоне от *100* до *-100* с декрементом, равным *5*.

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

int main()
{
  int i;
  for(i=100; i>=-100; i=i-5) cout << i << ' ';
  return 0;
}
```

Важно понимать, что условное выражение всегда тестируется в начале выполнения цикла for. Это значит, что если первая же проверка условия даст значение ЛОЖЬ, код тела цикла не выполнится ни разу. Вот пример:

```
for(count=10; count<5; count++)
cout << count; // Эта инструкция не выполнится.
```

Этот цикл никогда не выполнится, поскольку уже при входе в него значение его управляющей переменной *count* больше пяти. Это делает условное выражение (*count < 5*) ложным с самого начала. Поэтому даже одна итерация этого цикла не будет выполнена.

Вариации на тему цикла for

Цикл for — одна из наиболее гибких инструкций в C++, поскольку она позволяет получить широкий диапазон вариантов ее использования. Например, для управления циклом for можно использовать несколько переменных. Рассмотрим следующий фрагмент кода.

```
for(x=0, y=10; x<=10; ++x, --y)
  cout << x << ' ' << y << '\n';
```

Здесь запятыми отделяются две инструкции инициализации и два инкрементных выражения. Это делается для того, чтобы компилятор "понимал", что существует две инструкции инициализации и две инструкции инкремента (декремента). В C++ запятая

инструкции инициализации и две инструкции инкремента (декремента). В C++ запятая представляет собой оператор, который, по сути, означает *"сделай это и то"*. Другие применения оператора "запятая" мы рассмотрим ниже в этой книге, но чаще всего он используется в цикле for. При входе в данный цикл инициализируются обе переменные — *x* и *y*. После выполнения каждой итерации цикла переменная *x* инкрементируется, а переменная *y* декрементируется. Использование нескольких управляющих переменных в цикле иногда позволяет упростить алгоритмы. В разделах инициализации и инкремента цикла for можно использовать любое количество инструкций, но обычно их число не превышает двух.

Условным выражением, которое управляет циклом for, может быть любое допустимое C++-выражение. При этом оно необязательно должно включать управляющую переменную цикла. В следующем примере цикл будет выполняться до тех пор, пока пользователь не нажмет клавишу на клавиатуре. В этой программе представлена еще одна (очень важная) библиотечная функция: *kbhit()*. Она возвращает значение ЛОЖЬ, если ни одна клавиша не была нажата на клавиатуре, и значение ИСТИНА в противном случае. Функция не ожидает нажатия клавиши, позволяя тем самым циклу выполняться до тех пор, пока оно не произойдет. Функция *kbhit()* не определяется стандартом C++, но включена в расширение языка C++, которое поддерживается большинством компиляторов. Для ее использования в программу необходимо включить заголовок *<conio.h>*. (Этот заголовок необходимо указывать с расширением *.h*, поскольку он не определен стандартом C++.)

```
#include <iostream>
#include <conio.h>
using namespace std;

int main( )
{
  int i;
  // Вывод чисел на экран до нажатия любой клавиши.
  for(i=0; !kbhit( ); i++)
    cout << i << ' ';

  return 0;
}
```

На каждой итерации цикла вызывается функция *kbhit()*. Если после запуска программы нажать какую-нибудь клавишу, эта функция возвратит значение ИСТИНА, в результате чего выражение *!kbhit()* даст значение ЛОЖЬ, и цикл остановится. Но если не нажимать клавишу, функция возвратит значение ЛОЖЬ, а выражение *!kbhit()* даст значение ИСТИНА, что позволит циклу продолжать "крутиться".

Важно! *Функция kbhit() не входит в состав стандартной библиотеки C++. Дело в том, что стандартная библиотека определяет только минимальный набор функций, который должны иметь все C++-компиляторы. Функция kbhit() не включена в этот минимальный набор, поскольку не все среды могут поддерживать взаимодействие с клавиатурой. Однако функцию kbhit() поддерживают практически все серийно выпускаемые C++- компиляторы. Производители компиляторов могут обеспечить поддержку большего числа функций, чем это необходимо для соблюдения минимальных требований по части стандартной библиотеки C++. Дополнительные же функции позволяют шире использовать возможности среды программирования. Если для вас не проблематичен вопрос переносимости кода в другую среду выполнения, вы можете свободно использовать все функции, поддерживаемые вашим компилятором.*

Отсутствие элементов в определении цикла

В C++ разрешается опустить любой элемент заголовка цикла (*инициализация*, *условное выражение*, *инкремент*) или даже все сразу. Например, мы хотим написать цикл, который

выражение, инкремент) или даже все сразу. Например, мы хотим написать цикл, который должен выполняться до тех пор, пока с клавиатуры не будет введено число *123*. Вот как выглядит такая программа.

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

int main( )
{
  int x;
  for(x=0; x!=123; ) {
    cout << "Введите число: ";
    cin >> x;
  }
  return 0;
}
```

Здесь в заголовке цикла for отсутствует выражение инкремента. Это означает, что при каждом повторении цикла выполняется только одно действие: значение переменной *x* сравнивается с числом *123*. Но если ввести с клавиатуры число *123*, условное выражение, проверяемое в цикле, станет ложным, и цикл завершится. Поскольку выражение инкремента в заголовке цикла for отсутствует, управляющая переменная цикла не модифицируется.

Приведем еще один вариант цикла for, в заголовке которого, как показано в следующем фрагменте кода, отсутствует раздел инициализации.

```
cout << "Введите номер позиции: ";
  cin >> x;
for( ; x<limit; x++) cout << ' ';
```

Здесь пустует раздел инициализации, а управляющая переменная *x* инициализируется значением, вводимым пользователем с клавиатуры до входа в цикл.

К размещению выражения инициализации за пределами цикла, как правило, прибегают только в том случае, когда начальное значение генерируется сложным процессом, который неудобно поместить в определение цикла. Кроме того, раздел инициализации оставляют пустым и в случае, когда управление циклом осуществляется с помощью параметра некоторой функции, а в качестве начального значения управляющей переменной цикла используется значение, которое получает параметр при вызове функции.

Бесконечный цикл

Бесконечный цикл — *это цикл, который никогда не заканчивается*.

Оставив пустым условное выражение цикла for, можно создать бесконечный цикл (цикл, который никогда не заканчивается). Способ создания такого цикла показан на примере следующей конструкции цикла for.

```
for(;;)
{
  //...
}
```

Этот цикл будет работать без конца. Несмотря на существование некоторых задач программирования (например, командных процессоров операционных систем), которые требуют наличия бесконечного цикла, большинство "бесконечных циклов" — это просто циклы со специальными требованиями к завершению. Ближе к концу этой главы будет показано, как завершить цикл такого типа. (Подсказка: с помощью инструкции *break*.)

Циклы временной задержки

В программах часто используются так называемые циклы временной задержки. Их задача — просто "убить время". Для создания таких циклов достаточно оставить пустым тело цикла, т.е. опустить те инструкции, которые повторяет цикл на каждой итерации. Вот пример:

```
for(x=0; x<1000; x++);
```

Этот цикл лишь инкрементирует значение переменной *x* и не делает ничего более. Точка с запятой (в конце строки) необходима по причине того, что цикл for ожидает получить инструкцию, которая может быть пустой (как в данном случае).

Прежде чем двигаться дальше, не помешало бы поэкспериментировать с собственными вариациями на тему цикла for. Это вам поможет убедиться в его гибкости и могуществе.

Инструкция switch

Инструкция switch— *это инструкция многонаправленного ветвления*, *которая позволяет выбрать одну из множества альтернатив*.

Прежде чем переходить к изучению других циклических C++-конструкций, познакомимся с еще одной инструкцией выбора — *switch*. Инструкция *switch* обеспечивает многонаправленное ветвление. Она позволяет делать выбор одной из множества альтернатив. Хотя многонаправленное тестирование можно реализовать с помощью последовательности вложенных *if*-инструкций, во многих ситуациях инструкция *switch* оказывается более эффективным решением. Она работает следующим образом. Значение выражения последовательно сравнивается с константами из заданного списка. При обнаружении совпадения для одного из условий сравнения выполняется последовательность инструкций, связанная с этим условием. Общий формат записи инструкции *switch* таков.

```
switch(выражение) {
  case константа1:
    последовательность инструкций
    break;
  case константа2:
    последовательность инструкци
    break;
  case константа3:
    последовательность инструкций
    break;
  .
  .
  .
  default:
    последовательность инструкций
}
```

Элемент *выражение* инструкции switch должен при вычислении давать целочисленное или символьное значение. (Выражения, имеющие, например, тип с плавающей точкой, не разрешены.) Очень часто в качестве управляющего *switch*-выражения используется одна переменная.

Инструкция break завершает выполнение кода, *определенного инструкцией switch*.

Последовательность инструкций *default*-ветви выполняется в том случае, если ни одна из заданных *case*-констант не совпадет с результатом вычисления *switch*-выражения. Ветвь

default необязательна. Если она отсутствует, то при несовпадении результата выражения ни с одной из *case*-констант никакое действие выполнено не будет. Если такое совпадение все-таки обнаружится, будут выполняться инструкции, соответствующие данной *case*-ветви, до тех пор, пока не встретится инструкция *break* или не будет достигнут конец *switch*-инструкции (либо в *default*-, либо в последней *case*-ветви).

Инструкции default-ветви *выполняются в том случае, если ни одна из case констант не совпадет с результатом вычисления switch-выражения.*

Итак, для применения *switch*-инструкции необходимо знать следующее.

■ Инструкция *switch* отличается от инструкции *if* тем, что *switch*-выражение можно тестировать только с использованием условия равенства (т.е. на совпадение *switch*-выражения с заданными *case*-константами), в то время как условное *if*-выражение может быть любого типа.

■ Никакие две *case*-константы в одной *switch*-инструкции не могут иметь идентичных значений.

■ Инструкция *switch* обычно более эффективна, чем вложенные if-инструкции.

■ Последовательность инструкций, связанная с каждой *case*-ветвью, не является блоком. Однако полная *switch*-инструкция определяет блок. Значимость этого факта станет очевидной после того, как вы больше узнаете о C++.

Согласно стандарту C++ switch-конструкция может иметь не более 16 384 *case*-инструкций. Но на практике (исходя из соображений эффективности) обычно ограничиваются гораздо меньшим их количеством.

Использование *switch*-инструкции демонстрируется в следующей программе. Она создает простую "справочную" систему, которая описывает назначение *for*-, *if*- и *switch*-инструкций. После отображения списка предлагаемых тем, по которым возможно предоставление справки, программа переходит в режим ожидания до тех пор, пока пользователь не сделает свой выбор. Введенное пользователем значение используется в инструкции *switch* для отображения информация по указанной теме. (Вы могли бы в качестве упражнения дополнить информацию по имеющимся темам, а также ввести в эту "справочную" систему новые темы.)

```
// Демонстрация switch-инструкции на примере простой "справочной" системы.
#include <iostream>
using namespace std;

int main( )
{
  int choice;
  cout << "Справка по темам:\n\n";
  cout << "1. for\n";
  cout << "2. if\n";
  cout << "3. switch\n\n";

  cout << "Введите номер темы (1-3): ";
    cin >> choice;
  cout << "\n";
  switch(choice) {
    case 1:
```

```
      cout << "for - это самый универсальный цикл в C++.\n";
      break;
    case 2:
      cout << "if - это инструкция условного ветвления.\n";
      break;
    case 3:
      cout <<"switch - это инструкция многонаправленного ветвления.\n";
      break;
    default:
      cout << "Вы должны ввести число от 1 до 3.\n";
    }
  return 0;
}
```

Вот один из вариантов выполнения этой программы.

```
Справка по темам:
1. for
2. if
3. switch
Введите номер темы (1-3) : 2
```

if - это инструкция условного ветвления.

Формально инструкция *break* необязательна, хотя в большинстве случаев использования switch-конструкций она присутствует. Инструкция *break*, стоящая в последовательности инструкций любой case-ветви, приводит к выходу из всей switch-конструкции и передает управление инструкции, расположенной сразу после нее. Но если инструкция break в case-ветви отсутствует, будут выполнены все инструкции, связанные с данной case-ветвью, а также все последующие инструкции, расположенные под ней, до тех пор, пока все-таки не встретится инструкция break, относящаяся к одной из последующих case-ветвей, или не будет достигнут конец switch-конструкции.

Рассмотрим внимательно следующую программу. Попробуйте предугадать, что будет отображено на экране при ее выполнении.

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

int main( )
{
  int i;
  for(i=0; i<5; i++) {
    switch(i) {
      case 0: cout << "меньше 1\n";
      case 1: cout << "меньше 2\n";
      case 2: cout << "меньше 3\n";
      case 3: cout << "меньше 4\n";
      case 4: cout << "меньше 5\n";
    }
```

```
  cout << ' \n';
  }
  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
меньше 1
меньше 2
меньше 3
меньше 4
меньше 5

меньше 2
меньше 3
меньше 4
меньше 5

меньше 3
меньше 4
меньше 5

меньше 4
меньше 5

меньше 5
```

Как видно по результатам, если инструкция *break* в одной case-ветви отсутствует, выполняются инструкции, относящиеся к следующей case-ветви.

Как показано в следующем примере, в switch-конструкцию можно включать "пустые" case-ветви.

```
switch(i) {
  case 1:
  case 2:
  case 3: do_something();
    break;
  case 4: do_something_else();
    break;
}
```

Если переменная *i* в этом фрагменте кода получает значение *1*, *2* или *3*, вызывается функция *do_something()*. Если же значение переменной *i* равно *4*, делается обращение к функции *do_something_else()*. Использование "пачки" нескольких пустых case-ветвей характерно для случаев, когда они используют один и тот же код.

Вложенные инструкции switch

Инструкция switch может быть использована как часть case -последовательности внешней инструкции switch. В этом случае она называется *вложенной* инструкцией switch. Необходимо отметить, что case-константы внутренних и внешних инструкций switch могут иметь одинаковые значения, при этом никаких конфликтов не возникнет. Например,

следующии фрагмент кода вполне допустим.

```
switch(ch1) {
  case 'A': cout <<"Эта константа A - часть внешней инструкции switch";
    switch(ch2) {
       case 'A': cout <<"Эта константа A - часть внутренней инструкции switch";
        break;
      case 'B': // ...
    }
    break;
  case 'B': // ...
```

Цикл while

Инструкция while — *еще один способ организации циклов в C++.*

Общая форма цикла *while* имеет такой вид:

```
while(выражение) инструкция;
```

Здесь под элементом *инструкция* понимается либо одиночная инструкция, либо блок инструкций. Работой цикла управляет элемент *выражение*, который представляет собой любое допустимое C++-выражение. Элемент *инструкция* выполняется до тех пор, пока условное выражение возвращает значение ИСТИНА. Как только это *выражение* становится ложным, управление передается инструкции, которая следует за этим циклом.

Использование цикла *while* демонстрируется на примере следующей небольшой программы. Практически все компиляторы поддерживают расширенный набор символов, который не ограничивается символами *ASCII*. В расширенный набор часто включаются специальные символы и некоторые буквы из алфавитов иностранных языков. *ASCII*-символы используют значения, не превышающие число *127*, а расширенный набор символов— значения из диапазона *128-255*. При выполнении этой программы выводятся все символы, значения которых лежат в диапазоне *32-255* (32 — это код пробела). Выполнив эту программу, вы должны увидеть ряд очень интересных символов.

```
/* Эта программа выводит все печатаемые символы, включая расширенный набор символов, если таковой существует.
*/
#include <iostream>
using namespace std;

int main(){
  unsigned char ch;
  ch = 32;
  while(ch) {
    cout << ch;
    ch++;
  }
  return 0;
}
```

Рассмотрим *while*-выражение из предыдущей программы. Возможно, вас удивило, что оно состоит всего лишь из одной переменной *ch*. Но "ларчик" здесь открывается просто. Поскольку переменная *ch* имеет здесь тип *unsigned char*, она может содержать значения от *0* до *255*. Если ее значение равно *255*, то после инкрементирования оно "сбрасывается" в нуль.

Следовательно, факт равенства значения переменной *ch* нулю служит удобным способом завершить *while*-цикл.

Подобно циклу for, условное выражение проверяется при входе в цикл while, а это значит, что тело цикла (при ложном результате вычисления условного выражения) может не выполниться ни разу. Это свойство цикла устраняет необходимость отдельного тестирования до начала цикла. Следующая программа выводит строку, состоящую из точек. Количество отображаемых точек равно значению, которое вводит пользователь. Программа не позволяет "рисовать" строки, если их длина превышает 80 символов. Проверка на допустимость числа выводимых точек выполняется внутри условного выражения цикла, а не снаружи.

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

int main( )
{
  int len;
  cout << "Введите длину строки (от 1 до 79): ";
    cin >> len;
  while(len>0 && len<80) {
    cout << '.';
    len--;
  }
  return 0;
}
```

Тело while-цикла может вообще не содержать ни одной инструкции. Вот пример:

```
while(rand( ) != 100);
```

Этот цикл выполняется до тех пор, пока случайное число, генерируемое функцией *rand()*, не окажется равным числу *100*.

Цикл do-while

Цикл do-while — *это единственный цикл, который всегда делает итерацию хотя бы один раз*.

В отличие от циклов *for* и *while*, в которых условие проверяется при входе, цикл *do-while* проверяет условие при выходе из цикла. Это значит, что цикл *do-while* всегда выполняется хотя бы один раз. Его общий формат имеет такой вид.

```
do{
  инструкции;
}while(выражение);
```

Несмотря на то что фигурные скобки необязательны, если элемент *инструкции* состоит только из одной инструкции, они часто используются для улучшения читабельности конструкции *do-while*, не допуская тем самым путаницы с циклом *while*. Цикл *do-while* выполняется до тех пор, пока остается истинным элемент *выражение*, который представляет собой условное выражение.

В следующей программе цикл *do-while* выполняется до тех пор, пока пользователь не введет число *100*.

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

```
using namespace std;

int main()
{
  int num;
  do {
    cout << "Введите число (100 - для выхода): ";
    cin >> num;
  }while(num != 100);
  return 0;
}
```

Используя цикл do-while, мы можем еще более усовершенствовать программу "Угадай магическое число". На этот раз программа "не выпустит" вас из цикла угадывания, пока вы не угадаете это число.

```
// Программа "Угадай магическое число":
// 3-е усовершенствование.
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
  int magic; // магическое число
  int guess; // вариант пользователя
  magic = rand(); // Получаем случайное число.
  do {
    cout << "Введите свой вариант магического числа: ";
      cin >> guess;
    if(guess == magic) {
      cout << "** Правильно ** ";
      cout << magic <<" и есть то самое магическое число.\n";
    }
    else {
      cout << "...Очень жаль, но вы ошиблись.";
      if(guess > magic)
        cout <<" Ваш вариант превышает магическое число.\n";
      else cout <<" Ваш вариант меньше магического числа.\n";
    }
  }while(guess != magic);
  return 0;
}
```

Использование инструкции continue

Инструкция continue позволяет немедленно перейти к выполнению следующей итерации цикла.

В C++ существует средство "досрочного" выхода из текущей итерации цикла. Этим средством является инструкция *continue*. Она принудительно выполняет переход к следующей итерации, опуская выполнение оставшегося кода в текущей. Например, в следующей программе инструкция *continue* используется для "ускоренного" поиска чётных чисел в диапазоне от 0 до 100.

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

int main()
{
  int x;
  for(x=0; x<=100; x++) {
    if(x%2) continue;
    cout << x << ' ';
  }
  return 0;
}
```

Здесь выводятся только четные числа, поскольку при обнаружении нечётного числа происходит преждевременный переход к следующей итерации, и *cout*-инструкция опускается.

В циклах while и do-while инструкция *continue* передает управление непосредственно инструкции, проверяющей условное выражение, после чего циклический процесс продолжает "идти своим чередом". А в цикле for после выполнения инструкции *continue* сначала вычисляется инкрементное выражение, а затем— условное. И только после этого циклический процесс будет продолжен.

Использование инструкции break для выхода из цикла

Инструкция break позволяет немедленно выйти из цикла.

С помощью инструкции *break* можно организовать немедленный выход из цикла, опустив выполнение кода, оставшегося в его теле, и проверку условного выражения. При обнаружении внутри цикла инструкции *break* цикл завершается, а управление передается инструкции, следующей, после цикла. Рассмотрим простой пример.

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

int main()
{
  int t;
  // Цикл работает для значений t от 0 до 9, а не до 100!
  for(t=0; t<100; t++) {
    if(t==10) break;
    cout << t <<' ';
  }
  return 0;
}
```

Эта программа выведет на экран числа от *0* до 9, а не до *100*, поскольку инструкция *break* при

Эта программа выведет на экран числа от 0 до 9, а не до 100, поскольку инструкция break при значении *t*, равном *10*, обеспечивает немедленный выход из цикла.

Инструкция *break* обычно используется циклах, в которых при создании особых условий необходимо обеспечить немедленное их завершение. Следующий фрагмент содержит пример ситуации, когда по нажатию клавиши выполнение цикла останавливается.

```
for(i=0; i<1000; i++) {
  // Какие-то действия.
  if(kbhit()) break;
}
```

Инструкция *break* приводит к выходу из самого внутреннего цикла. Рассмотрим пример.

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

int main()
{
  int t, count;
  for(t=0; t<100; t++) {
    count = 1;
    for(;;) {
      cout << count << ' ';
      count++;
      if(count==10) break;
    }
    cout << '\n';
  }
  return 0;
}
```

Эта программа *100* раз выводит на экран числа от *0* до *9*. При каждом выполнении инструкции *break* управление передается назад во внешний цикл *for*.

На заметку. *Инструкция break, которая завершает выполнение инструкции switch, влияет только на инструкцию switch, а не на содержащий ее цикл.*

На примере предыдущей программы вы убедились, что в C++ с помощью инструкция for можно создать бесконечный цикл. (Бесконечные циклы можно также создавать, используя инструкции while или do-while, но цикл for — это традиционное решение.) Для выхода из бесконечного цикла необходимо использовать инструкцию break. (Безусловно, инструкцию break можно использовать и для завершения небесконечного цикла.)

Вложенные циклы

Как было продемонстрировано на примере предыдущей программы, один цикл можно вложить в другой. В C++ разрешено использовать до 256 уровней вложения. Вложенные циклы используются для решения задач самого разного профиля. Например, в следующей программе вложенный цикл for позволяет найти простые числа в диапазоне от *2* до *1000*.

```
/* Эта программа выводит простые числа, найденные в диапазоне от 2 до
1000.
*/
#include <iostream>
using namespace std;
```

```
int main()
{
  int i, j;
  for(i=2; i<1000; i++) {
    for(j=2; j<=(i/j); j++)
       if(!(i%j)) break; // Если число имеет множитель, значит, оно не простое.
    if(j > (i/j)) cout << i << " - простое число\n";
  }
  return 0;
}
```

Эта программа определяет, является ли простым число, которое содержится в переменной *i*, путем последовательного его деления на значения, расположенные между числом *2* и результатом вычисления выражения *i/j*. (Остановить перебор множителей можно на значении выражения *i/j*, поскольку число, которое превышает *i/j*, уже не может быть множителем значения *i*.) Если остаток от деления *i/j* равен нулю, значит, число *i* не является простым. Но если внутренний цикл завершится полностью (без досрочного окончания по инструкции *break*), это означает, что текущее значение переменной *i* действительно является простым числом.

Инструкция goto

Инструкция goto — *это C++-инструкция безусловного перехода*.

Долгие годы эта инструкция находилась в немилости у программистов, поскольку способствовала, с их точки зрения, созданию *"спагетти-кода"*. Однако инструкция *goto* по-прежнему используется, и иногда даже очень эффективно. В этой книге не делается попытка "реабилитации" законных прав этой инструкции в качестве одной из форм управления программой. Более того, необходимо отметить, что в любой ситуации (в области программирования) можно обойтись без инструкции *goto*, поскольку она не является элементом, обеспечивающим полноту описания языка программирования. Вместе с тем, в определенных ситуациях ее использование может быть очень полезным В этой книге было решено ограничить использование инструкции *goto* рамками этого раздела, так как, по мнению большинства программистов, она вносит в программу лишь беспорядок и делает ее практически нечитабельной. Но, поскольку использование инструкции *goto* в некоторых случаях может сделать намерение программиста яснее, ей стоит уделить некоторое внимание.

Инструкция *goto* требует наличия в программе *метки*. *Метка* — это действительный в C++ идентификатор, за которым поставлено двоеточие. При выполнении инструкции *goto* управление программой передается инструкции, указанной с помощью *метки*. *Метка* должна находиться в одной функции с инструкцией *goto*, которая ссылается на эту метку.

Метка — *это идентификатор, за которым стоит двоеточие*.

Например, с помощью инструкции *goto* и *метки* можно организовать следующий цикл на 100 итераций.

```
x = 1;
  loop1:
    x++;
    if(x < 100) goto loop1;
```

Иногда инструкцию *goto* стоит использовать для выхода из глубоко вложенных инструкций цикла. Рассмотрим следующий фрагмент кода.

```
for(...) {
  for(...) {
    while(...) {
      if(...) goto stop;
    }
  }
}
stop:
  cout << "Ошибка в программе.\n";
```

Чтобы заменить инструкцию *goto*, пришлось бы выполнить ряд дополнительных проверок. В данном случае инструкция *goto* существенно упрощает программный код. Простым применением инструкции *break* здесь не обошлось, поскольку она обеспечила бы выход лишь из самого внутреннего цикла.

Важно! *Инструкцию goto следует применять с оглядкой (как сильнодействующее лекарство). Если без нее ваш код будет менее читабельным или для вас важна скорость выполнения программы, то в таких случаях использование инструкции goto показано.*

Итак, подведем итоги...

Следующий пример представляет собой последнюю версию программы "Угадай магическое число". В ней использованы многие средства C++-программирования, представленные в этой главе, и, прежде чем переходить к следующей, убедитесь в том, что хорошо понимаете все рассмотренные здесь элементы языка C++. Этот вариант программы позволяет сгенерировать новое число, сыграть в игру и выйти из программы.

```
// Программа "Угадай магическое число": последняя версия.
#include <iostream>
#include <cstdlib>
using namespace std;

void play(int m);
int main( )
{
  int option;
  int magic;
  magic = rand( );
  do {
    cout << "1. Получить новое магическое число\n";
    cout << "2. Сыграть\n";
    cout << "3. Выйти из программы\n";
    do {
      cout << "Введите свой вариант: ";
        cin >> option;
    }while(option<1 || option>3);
    switch(option){
      case 1:
```

```
      magic = rand();
      break;
    case 2:
      play(magic);
      break;
    case 3:
      cout << "До свидания !\n";
      break;
   }
  }while(option != 3);
  return 0;
}

// Сыграем в игру.
void play(int m)
{
  int t, x;
  for(t=0; t<100; t++) {
   cout << "Угадайте магическое число: ";
     cin >> x;
   if(x==m) {
     cout << "** Правильно **\n";
     return;
   }
   else
     if(x<m) cout << "Маловато.\n";
     else cout << "Многовато.\n";
  }
  cout << "Вы использовали все шансы угадать число. " << "Попытайтесь снова.\n";
}
```

Глава 5: Массивы и строки

В этой главе мы рассматриваем массивы. *Массив* (array) — это коллекция переменных одинакового типа, обращение к которым происходит с применением общего для всех имени. В C++ массивы могут быть одно- или многомерными, хотя в основном используются одномерные массивы. Массивы представляют собой удобное средство группирования связанных переменных.

Чаще всего используются символьные массивы, в которых хранятся строки. Как упоминалось выше, в C++ не определен встроенный тип данных для хранения строк. Поэтому строки реализуются как массивы символов. Такой подход к реализации строк дает C++-программисту больше "рычагов" управления по сравнению с теми языками, в которых используется отдельный строковый тип данных.

Одномерные массивы

Одномерный массив *— это список связанных переменных. Для объявления одномерного массива используется следующая форма записи:*

массива используется следующая форма записи.

```
тип имя_массива [размер];
```

Здесь с помощью элемента записи *тип* объявляется базовый тип массива. *Базовый тип* определяет тип данных каждого элемента, составляющего массив. Количество элементов, которые будут храниться в массиве, определяется элементом *размер*. Например, при выполнении приведенной ниже инструкции объявляется *int*-массив (состоящий из *10* элементов) с именем *sample*.

```
int sample[10];
```

Индекс идентифицирует конкретный элемент массива.

Доступ к отдельному элементу массива осуществляется с помощью индекса. *Индекс* описывает позицию элемента внутри массива. В C++ первый элемент массива имеет нулевой индекс. Поскольку массив *sample* содержит *10* элементов, его индексы изменяются от *0* до *9*. Чтобы получить доступ к элементу массива по индексу, достаточно указать нужный номер элемента в квадратных скобках. Так, первым элементом массива *sample* является *sample[0]*, а последним — *sample[9]*. Например, следующая программа помещает в массив *sample* числа от *0* до *9*.

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

int main( )
{
  int sample[10]; // Эта инструкция резервирует область памяти для 10 элементов типа int.
  int t;

  // Помещаем в массив значения.
  for(t=0; t<10; ++t) sample[t]=t;
    // Отображаем массив.
    for(t=0; t<10; ++t)
      cout << sample[t] << ' ';

  return 0;
}
```

В C++ все массивы занимают смежные ячейки памяти. (Другими словами, элементы массива в памяти расположены последовательно друг за другом.) Ячейка с наименьшим адресом относится к первому элементу массива, а с наибольшим — к последнему. Например, после выполнения этого фрагмента кода

```
int i [7];
int j;
for(j=0; j<7; j++)i[j]=j;
```

массив *i* будет выглядеть следующим образом.



Для одномерных массивов общий размер массива в байтах вычисляется так:

всего байтов = *размер типа в байтах* ***x*** *количество элементов*.

Массивы часто используются в программировании, поскольку позволяют легко обрабатывать большое количество связанных переменных. Например, в следующей программе создается массив из десяти элементов, каждому элементу присваивается случайное число, а затем на экране отображаются минимальное и максимальное значения.

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

int main( )
{
  int i, min_value, max_value;
  int list [10];
  for(i=0; i<10; i++) list[i] = rand( );

  // Находим минимальное значение.
  min_value = list[0];
  for(i=1; i<10; i++)
    if(min_value > list[i]) min_value = list[i];
  cout << "Минимальное значение: " << min_value << ' \n';

  // Находим максимальное значение.
  max_value = list[0];
  for(i=1; i<10; i++)
    if(max_value < list[i]) max_value = list[i];
  cout << "Максимальное значение: " << max_value << '\n';

  return 0;
}
```

В C++ нельзя присвоить один массив другому. В следующем фрагменте кода, например, присваивание *a = b;* недопустимо.

```
int a[10], b[10];
// ...
a = b; // Ошибка!!!
```

Чтобы поместить содержимое одного массива в другой, необходимо отдельно выполнить присваивание каждого значения.

На границах массивов погранзаставы нет

В C++ не выполняется никакой проверки "нарушения границ" массивов, т.е. ничего не может помешать программисту обратиться к массиву за его пределами. Если это происходит при выполнении инструкции присваивания, могут быть изменены значения в ячейках памяти, выделенных некоторым другим переменным или даже вашей программе. Другими словами, обращение к массиву (размером *N* элементов) за границей *N*-го элемента может привести к разрушению программы при отсутствии каких-либо замечаний со стороны компилятора и без выдачи сообщений об ошибках во время работы программы. Это означает, что вся ответственность за соблюдение границ массивов лежит только на программистах, которые должны гарантировать корректную работу с массивами. Другими словами, программист обязан использовать массивы достаточно большого размера, чтобы в них можно было без осложнений помещать данные, но лучше всего в программе предусмотреть проверку пересечения границ массивов.

предусмотреть проверку пересечения границ массивов.

Например, C++-компилятор "молча" скомпилирует и позволит запустить следующую программу на выполнение, несмотря на то, что в ней происходит выход за границы массива *crash*.

Осторожно! *Не выполняйте следующий пример программы. Это может разрушить вашу систему.*

```
// Некорректная программа. Не выполняйте ее!
int main( )
{
  int crash[10], i;
  for(i=0; i<100; i++) crash[i]=i;
  return 1;
}
```

В данном случае цикл for выполнит 100 итераций, несмотря на то, что массив *crash* предназначен для хранения лишь десяти элементов. При выполнении этой программы возможна перезапись важной информации, что может привести к аварийной остановке программы.

Вас, возможно, удивляет такая "непредусмотрительность" C++, которая выражается в отсутствии встроенных средств динамической проверки на "неприкосновенность" границ массивов. Напомню, однако, что язык C++ предназначен для профессиональных программистов, и его задача — предоставить им возможность создавать максимально эффективный код. Любая проверка корректности доступа средствами C++ существенно замедляет выполнение программы. Поэтому подобные действия оставлены на рассмотрение программистам. Как будет показано ниже в этой книге, при необходимости программист может сам определить тип массива и заложить в него проверку нерушимости границ.

Сортировка массива

Одной из самых распространенных операций, выполняемых над массивами, является сортировка. Существует множество различных алгоритмов сортировки. Широко применяется, например, сортировка перемешиванием и сортировка методом Шелла. Известен также алгоритм *Quicksort* (быстрая сортировка с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины). Однако самым простым считается алгоритм сортировки пузырьковым методом. Несмотря на то что пузырьковая сортировка не отличается высокой эффективностью (и в самом деле, его производительность неприемлема для сортировки больших массивов), его вполне успешно можно применять для сортировки массивов малого размера.

Алгоритм сортировки пузырьковым методом получил свое название от способа, используемого для упорядочивания элементов массива. Здесь выполняются повторяющиеся операции сравнения и при необходимости меняются местами смежные элементы. При этом элементы с меньшими значениями постепенно перемещаются к одному концу массива, а элементы с большими значениями — к другому. Этот процесс напоминает поведение пузырьков воздуха в резервуаре с водой. Пузырьковая сортировка выполняется путем нескольких проходов по массиву, во время которых при необходимости осуществляется перестановка элементов, оказавшихся "не на своем месте". Количество проходов, гарантирующих получение отсортированного массива, равно количеству элементов в массиве, уменьшенному на единицу.

В следующей программе реализована сортировка массива (целочисленного типа), содержащего случайные числа. Эта программа заслуживает внимательного разбора.

```
// Использование метода пузырьковой сортировки
```

```
// для упорядочения массива.
#include <iostream>
#include <cstdlib>
using namespace std;

int main( )
{
  int nums[10];
  int a, b, t;
  int size;
  size = 10; // Количество элементов, подлежащих сортировке.

  // Помещаем в массив случайные числа.
  for(t=0; t<size; t++) nums[t] = rand( );

  // Отображаем исходный массив.
  cout << "Исходный массив: ";
  for(t=0; t<size; t++) cout << nums[t] << ' ';
  cout << '\n';

  // Реализация метода пузырьковой сортировки.
  for(a=1; a<size; a++)
    for(b=size-1; b>=a; b--) {
      if(nums[b-1] > nums[b]) {     // Элементы неупорядочены.
        // Меняем элементы местами.
        t = nums[b-1];
        nums[b-1] = nums[b];
        nums[b] = t;
      }
    }
  }

  // Конец пузырьковой сортировки.
  // Отображаем отсортированный массив.
  cout << "Отсортированный массив: ";
  for(t=0; t<size; t++)
    cout << nums[t] << ' ';

  return 0;
}
```

Хотя алгоритм пузырьковой сортировки пригоден для небольших массивов, для массивов большого размера он становится неэффективным. Более универсальным считается алгоритм *Quicksort*. В стандартную библиотеку C++ включена функция *qsort()*, которая реализует одну из версий этого алгоритма. Но, прежде чем использовать ее, вам необходимо изучить больше средств C++. (Подробно функция *qsort()* рассмотрена в главе 20.)

20.)

Строки

Чаще всего одномерные массивы используются для создания символьных строк. В C++ *строка* определяется как символьный массив, который завершается нулевым символом (*'\0'*). При определении длины символьного массива необходимо учитывать признак ее завершения и задавать его длину на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве.

Строка — *это символьный массив, который завершается нулевым символом*.

Например, объявляя массив *str*, предназначенный для хранения *10*-символьной строки, следует использовать следующую инструкцию.

```
char str [11];
```

Заданный здесь размер (*11*) позволяет зарезервировать место для нулевого символа в конце строки.

Как упоминалось выше в этой книге, C++ позволяет определять строковые литералы. Вспомним, что *строковый литерал* — это список символов, заключенный в двойные кавычки. Вот несколько примеров.

```
"Привет"
"Мне нравится C++"
"#$%@@#$"
""
```

Строка, приведенная последней (""), называется нулевой. Она состоит только из одного нулевого символа (признака завершения строки). Нулевые строки используются для представления пустых строк.

Вам не нужно вручную добавлять в конец строковых констант нулевые символы. C++-компилятор делает это автоматически. Следовательно, строка "ПРИВЕТ" в памяти размещается так, как показано на этом рисунке:



Считывание строк с клавиатуры

Проще всего считать строку с клавиатуры, создав массив, который примет эту строку с помощью инструкции *cin*. Считывание строки, введенной пользователем с клавиатуры, отображено в следующей программе.

```
// Использование cin-инструкции для считывания строки с клавиатуры.
#include <iostream>
using namespace std;

int main( )
{
  char str[80];
  cout << "Введите строку: ";
    cin >> str; // Считываем строку с клавиатуры.
  cout << "Вот ваша строка: ";
  cout << str;

  return 0;
}
```

Несмотря на то что эта программа формально корректна, она не лишена недостатков. Рассмотрим следующий результат ее выполнения.

```
Введите строку: Это проверка
Вот ваша строка: Это
```

Как видите, при выводе строки, введенной с клавиатуры, программа отображает только слово *"Это"*, а не всю строку. Дело в том, что оператор *">>"* прекращает считывание строки, как только встречает символ пробела, табуляции или новой строки (будем называть эти символы *пробельными*).

Для решения этой проблемы можно использовать еще одну библиотечную функцию *gets()*. Общий формат ее вызова таков.

```
gets(имя_массива);
```

Если в программе необходимо считать строку с клавиатуры, вызовите функцию *gets()*, а в качестве аргумента передайте имя массива, не указывая индекса. После выполнения этой функции заданный массив будет содержать текст, введенный с клавиатуры. Функция *gets()* считывает вводимые пользователем символы до тех пор, пока он не нажмет клавишу *<Enter>*. Для вызова функции *gets()* в программу необходимо включить заголовок *<cstdio>*.

В следующей версии предыдущей программы демонстрируется использование функции *gets()*, которая позволяет ввести в массив строку символов, содержащую пробелы.

```
// Использование функции gets() для считывания строки с клавиатуры.
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
  char str[80];
  cout << "Введите строку: ";
  gets(str); // Считываем строку с клавиатуры.
  cout << "Вот ваша строка: ";
  cout << str;

  return 0;
}
```

На этот раз после запуска новой версии программы на выполнение и ввода с клавиатуры текста *"Это простой тест"* строка считывается полностью, а затем так же полностью и отображается.

```
Введите строку: Это простой тест
Вот ваша строка: Это простой тест
```

В этой программе следует обратить внимание на следующую инструкцию.

```
cout << str;
```

Здесь (вместо привычного литерала) используется имя строкового массива. И хотя причина такого использования инструкции *cout* вам станет ясной после прочтения еще нескольких глав этой книги, пока кратко заметим, что имя символьного массива, который содержит строку, можно использовать везде, где допустимо применение строкового литерала.

При этом имейте в виду, что ни оператор *">>"*, ни функция *gets()* не выполняют граничной проверки (на отсутствие нарушения границ массива). Поэтому, если пользователь введет

строку, длина которои превышает размер массива, возможны неприятности, о которых упоминалось выше. Из сказанного следует, что оба описанных здесь варианта считывания строк с клавиатуры потенциально опасны. Однако после подробного рассмотрения C++-возможностей ввода-вывода в главе 18 мы узнаем способы, позволяющие обойти эту проблему.

Некоторые библиотечные функции обработки строк

Язык C++ поддерживает множество функций обработки строк. Самыми распространенными из них являются следующие.

```
strcpy( )
strcat( )
strlen( )
strcmp( )
```

Для вызова всех этих функций в программу необходимо включить заголовок *<cstring>*. Теперь познакомимся с каждой функцией в отдельности.

Функция strcpy()

Общий формат вызова функции *strcpy()* таков:

```
strcpy (to, from);
```

Функция *strcpy()* копирует содержимое строки *from* в строку *to*. Помните, что массив, используемый для хранения строки *to*, должен быть достаточно большим, чтобы в него можно было поместить строку из массива *from*. В противном случае массив *to* переполнится, т.е. произойдет выход за его границы, что может привести к разрушению программы.

Использование функции *strcpy()* демонстрируется в следующей программе, которая копирует строку *"Привет"* в строку *str*.

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

int main( )
{
  char str[80];
  strcpy(str, "Привет");
  cout << str;

  return 0;
}
```

Функция strcat()

Обращение к функции *strcat()* имеет следующий формат.

```
strcat(s1, s2);
```

Функция *strcat()* присоединяет строку *s2* к концу строки *s1*, при этом строка *s2* не изменяется. Обе строки должны завершаться нулевым символом. Результат вызова этой функции, т.е. результирующая строка *s1* также будет завершаться нулевым символом. Использование функции *strcat()* демонстрируется в следующей программе, которая должна вывести на экран строку *"Привет всем!"*.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

int main()
{
  char s1[20], s2[10];
  strcpy(s1, "Привет");
  strcpy(s2, " всем!");
  strcat (s1, s2);
  cout << s1;

  return 0;
}
```

Функция strcmp()

Обращение к функции *strcmp()* имеет следующий формат:

```
strcmp(s1, s2);
```

Функция *strcmp()* сравнивает строку *s2* со строкой *s1* и возвращает значение *0*, если они равны. Если строка *s1* лексикографически (т.е. в соответствии с алфавитным порядком) больше строки *s2*, возвращается положительное число. Если строка *s1* лексикографически меньше строки *s2*, возвращается отрицательное число.

Использование функции *strcmp()* демонстрируется в следующей программе, которая служит для проверки правильности пароля, введенного пользователем (для ввода пароля с клавиатуры и его верификации служит функция *password()*).

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

bool password();
int main()
{
  if(password()) cout << "Вход разрешен.\n";
  else cout << "В доступе отказано.\n";

  return 0;
}

// Функция возвращает значение true, если пароль принят, и значение false в противном случае.
bool password()
{
  char s[80];
  cout << "Введите пароль: ";
  gets(s);
  if(strcmp(s, "пароль")) { // Строки различны.
    cout << "Пароль недействителен.\n";
    return false;
```

```
  }
  // Сравниваемые строки совпадают.
  return true;
}
```

При использовании функции *strcmp()* важно помнить, что она возвращает число *0* (т.е. значение false), если сравниваемые строки равны. Следовательно, если вам необходимо выполнить определенные действия при условии совпадения строк, вы должны использовать оператор *НЕ* (*!*). Например, при выполнении следующей программы запрос входных данных продолжается до тех пор, пока пользователь не введет слово *"Выход"*.

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

int main( )
{
  char s [80];
  for(;;) {
    cout << "Введите строку: ";
    gets (s);
    if(!strcmp("Выход", s)) break;
  }

  return 0;
}
```

Функция strlen()

Общий формат вызова функции *strlen()* таков:

```
strlen(s);
```

Здесь *s* — строка. Функция *strlen()* возвращает длину строки, указанной аргументом *s*.

При выполнении следующей программы будет показана длина строки, введенной с клавиатуры.

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

int main( )
{
  char str[80];
  cout << "Введите строку: "; gets(str);
  cout << "Длина строки равна: " << strlen(str);

  return 0;
}
```

Если пользователь введет строку *"Привет всем!"*, программа выведет на экране число *12*. При подсчете символов, составляющих заданную строку, признак завершения строки

При подсчете символов, составляющих заданную строку, признак завершения строки (нулевой символ) не учитывается.

А при выполнении этой программы строка, введенная с клавиатуры, будет отображена на экране в обратном порядке. Например, при вводе слова *"привет"* программа отобразит слово *"тевирп"*. Помните, что строки представляют собой символьные массивы, которые позволяют ссылаться на каждый элемент (символ) в отдельности.

```
// Отображение строки в обратном порядке.
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int main( )
{
  char str[80];
  int i;
  cout << "Введите строку: ";
  gets(str);
  for(i=strlen(str)-1; i>=0; i--)
    cout << str[i];
  return 0;
}
```

В следующем примере продемонстрируем использование всех этих четырех строковых функций.

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

int main( )
{
  char s1[80], s2 [80];
  cout << "Введите две строки: ";
  gets (s1); gets(s2);
  cout << "Их длины равны: " << strlen (s1);
  cout << ' '<< strlen(s2) << '\n';

  if(!strcmp(s1, s2)) cout << "Строки равны \n";
  else cout << "Строки не равны \n";
  strcat(s1, s2);
  cout << s1 << '\n';

  strcpy(s1, s2);
  cout << s1 << " и " << s2 << ' ';
  cout << "длины равны\n";
```

```
  cout << "теперь равны\n";
  return 0;
}
```

Если запустить эту программу на выполнение и по приглашению ввести строки *"привет"* и *"всем"*, то она отобразит на экране следующие результаты:

```
Их длины равны: 6 4
Строки не равны
привет всем
всем и всем теперь равны
```

Последнее напоминание: не забывайте, что функция *strcmp()* возвращает значение *false*, если строки равны. Поэтому, если вы проверяете равенство строк, необходимо использовать оператор *"!"* (*НЕ*), чтобы реверсировать условие (т.е. изменить его на обратное), как было показано в предыдущей программе.

Использование признака завершения строки

Факт завершения нулевыми символами всех C++-строк можно использовать для упрощения различных операций над ними. Следующий пример позволяет убедиться в том, насколько простой код требуется для замены всех символов строки их прописными эквивалентами.

```
// Преобразование символов строки  в их прописные эквиваленты.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

int main()
{
  char str[80];
  int i;
  strcpy(str, "test");
  for(i=0; str[i]; i++) str[i] = toupper(str[i]);
    cout << str;
  return 0;
}
```

Эта программа при выполнении выведет на экран слово *TEST*. Здесь используется библиотечная функция *toupper()*, которая возвращает прописной эквивалент своего символьного аргумента. Для вызова функции *toupper()* необходимо включить в программу заголовок *<cctype>*.

Обратите внимание на то, что в качестве условия завершения цикла for используется массив *str*, индексируемый управляющей переменной *i* (*str[i]*). Такой способ управления циклом вполне приемлем, поскольку за истинное значение в C++ принимается любое ненулевое значение. Вспомните, что все печатные символы представляются значениями, не равными нулю, и только символ, завершающий строку, равен нулю. Следовательно, этот цикл работает до тех пор, пока индекс не укажет на нулевой признак конца строки, т.е. пока значение *str[i]* не станет нулевым. Поскольку нулевой символ отмечает конец строки, цикл останавливается в точности там, где нужно. При дальнейшей работе с этой книгой вы увидите множество примеров, в которых нулевой признак конца строки используется подобным образом.

Важно! *Помимо функции toupper(), стандартная библиотека C++ содержит много других функций обработки символов. Например, функцию toupper() дополняет функция tolower(),*

функция-обработчик сигналов. Например, функция toupper() библиотеки функций ctype.h, которая возвращает строчный эквивалент своего символьного аргумента. Часто используются такие функции, как isalpha(), isdigit(), isspace() и ispunct(), которые принимают символьный аргумент и определяют, принадлежит ли он к соответствующей категории. Например, функция isalpha() возвращает значение ИСТИНА, если ее аргументом является буква (элемент алфавита).

Двумерные массивы

В C++ можно использовать многомерные массивы. Простейший многомерный массив — двумерный. Двумерный массив, по сути, представляет собой список одномерных массивов. Чтобы объявить двумерный массив целочисленных значений размером *10x20* с именем *twod*, достаточно записать следующее:

```
int twod[10][20];
```

Обратите особое внимание на это объявление. В отличие от многих других языков программирования, в которых при объявлении массива значения размерностей отделяются запятыми, в C++ каждая размерность заключается в собственную пару квадратных скобок.

Чтобы получить доступ к элементу массива *twod* с координатами *3,5* , необходимо использовать запись *twod[3][5]*. В следующем примере в двумерный массив помещаются последовательные числа от *1* до *12*.

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

int main( )
{
  int t,i, num[3] [4];
  for(t=0; t<3; ++t) {
    for(i=0; i<4; ++i) {
      num[t][i] = (t*4)+i+l;
      cout << num[t][i] << ' ';
    }
    cout << '\n';
  }

  return 0;
}
```

В этом примере элемент *num[0][0]* получит значение *1*, элемент *num[0][1]* — значение *2*, элемент *num[0][2]* — значение *3* и т.д. Значение элемента *num[2][3]* будет равно числу *12*. Схематически этот массив можно представить, как показано на рис. 5.1.

В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй — столбец. Из этого следует, что, если к доступ элементам массива предоставить в порядке, в котором они реально хранятся в памяти, то правый индекс будет изменяться быстрее, чем левый.



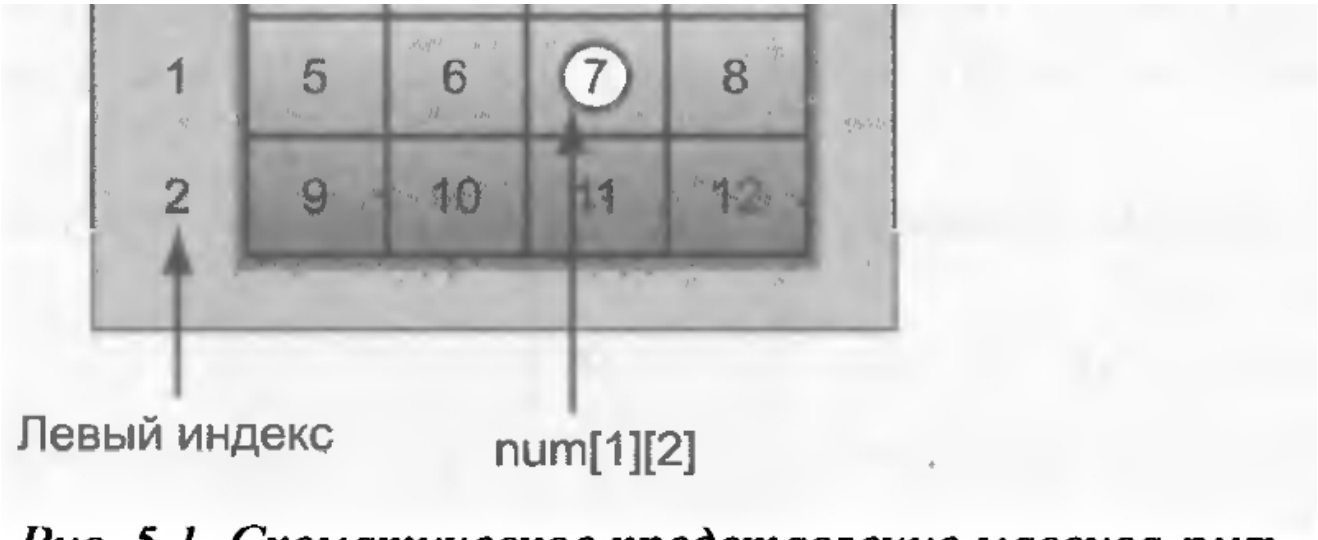


Рис. 5.1. Схематическое представление массива num

Необходимо помнить, что место хранения для всех элементов массива определяется во время компиляции. Кроме того, память, выделенная для хранения массива, используется в течение всего времени существования массива. Для определения количества байтов памяти, занимаемой двумерным массивом, используйте следующую формулу.

число байтов = число строк ***x*** *число столбцов* ***x*** *размер типа в байтах*

Следовательно, двумерный целочисленный массив размерностью *10x5* занимает в памяти *10x5x2*, т.е. *100* байт (если целочисленный тип имеет размер 2 байт).

Многомерные массивы

В C++, помимо двумерных, можно определять массивы трех и более измерений. Вот как объявляется многомерный массив.

```
тип имя[размер1] [размер2]... [размерN];
```

Например, с помощью следующего объявления создается трехмерный целочисленный массив размером *4x10x3*.

```
int multidim[4][10][3];
```

Как упоминалось выше, память, выделенная для хранения всех элементов массива, используется в течение всего времени существования массива. Массивы с числом измерений, превышающим три, используются нечасто, хотя бы потому, что для их хранения требуется большой объем памяти. Например, хранение элементов четырехмерного символьного массива размером *10x6x9x4* займет *2 160 байт*. А если каждую размерность увеличить в *10* раз, то занимаемая массивом память возрастет до *21 600 000 байт*. Как видите, большие многомерные массивы способны "съесть" большой объем памяти, а программа, которая их использует, может очень быстро столкнуться с проблемой нехватки памяти.

Инициализация массивов

В C++ предусмотрена возможность инициализации массивов. Формат инициализации массивов подобен формату инициализации других переменных.

```
тип имя_массива [размер] = {список_значений};
```

Здесь элемент *список_значений* представляет собой список значений инициализации элементов массива, разделенных запятыми. Тип каждого значения инициализации должен быть совместим с базовым типом массива (элементом *тип*). Первое значение инициализации будет сохранено в первой позиции массива, второе значение — во второй и т.д. Обратите внимание на то, что точка с запятой ставится после закрывающей фигурной скобки (*}*).

Например, в следующем примере *10*-элементный целочисленный массив

инициализируется числами от *1* до *10*.

```
int i [ 10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

После выполнения этой инструкции элемент *i[0]* получит значение *1*, а элемент *i[9]* — значение *10*.

Для символьных массивов, предназначенных для хранения строк, предусмотрен сокращенный вариант инициализации, который имеет такую форму,

```
char имя_массива[размер] = "строка";
```

Например, следующий фрагмент кода инициализирует массив *str* фразой *"привет"*,

```
  char str[7] = "привет";
```

Это равнозначно поэлементной инициализации.

```
  char str [7] = {'п', 'р', 'и', 'в', 'е', 'т', '\0'};
```

Поскольку в C++ строки должны завершаться нулевым символом, убедитесь, что при объявлении массива его размер указан с учетом признака конца. Именно поэтому в предыдущем примере массив *str* объявлен как *7*-элементный, несмотря на то, что в слове "привет" только шесть букв. При использовании строкового литерала компилятор добавляет нулевой признак конца строки автоматически.

Многомерные массивы инициализируются по аналогии с одномерными. Например, в следующем фрагменте программы массив *sqrs* инициализируется числами от *1* до *10* и квадратами этих чисел.

```
int sqrs[10][2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100
};
```

Теперь рассмотрим, как элементы массива *sqrs* располагаются в памяти (рис. 5.2).

При инициализации многомерного массива список инициализаторов каждой размерности (подгруппу инициализаторов) можно заключить в фигурные скобки. Вот, например, как выглядит еще один вариант записи предыдущего объявления.

```
int sqrs[10][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
  {8, 64},
```

```
  {9, 81},
  {10, 100}
};
```



Рис. 5.2. Схематическое представление инициализированного массива `sqrs`

При использовании подгрупп инициализаторов недостающие члены подгруппы будут инициализированы нулевыми значениями автоматически.

В следующей программе массив *sqrs* используется для поиска квадрата числа, введенного пользователем. Программа сначала выполняет поиск заданного числа в массиве, а затем выводит соответствующее ему значение квадрата.

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

int sqrs[10][2] = {
  {1, 1},
  {2, 4},
  {3, 9},
  {4, 16},
  {5, 25},
  {6, 36},
  {7, 49},
```

```
  {8, 64},
  {9, 81},
  {10, 100}
};
int main()
{
  int i, j;
  cout << "Введите число от 1 до 10: ";
    cin >> i;
  // Поиск значения i.
  for(j=0; j<10; j++)
    if(sqrs[j][0]==i) break;
  cout << "Квадрат числа " << i << " равен ";
  cout << sqrs[j][1];
  return 0;
}
```

Глобальные массивы инициализируются в начале выполнения программы, а локальные — при каждом вызове функции, в которой они содержатся. Рассмотрим пример.

```
#include <iostream>
#include <cstring>
using namespace std;
void f1();
int main()
{
  f1();
  f1();
  return 0;
}
void f1()
{
  char s[80]="Это просто тест\n";
  cout << s;
  strcpy(s, "ИЗМЕНЕНО\n"); // Изменяем значение строки s.
  cout << s;
}
```

При выполнении этой программы получаем такие результаты.

```
Это просто тест
ИЗМЕНЕНО
Это просто тест
```

```
Это просто тест
ИЗМЕНЕНО
```

В этой программе массив *s* инициализируется при каждом вызове функции *f1()*. Тот факт, что при ее выполнении массив *s* изменяется, никак не влияет на его повторную инициализацию при последующих вызовах функции *f1()*. Поэтому при каждом входе в нее на экране отображается следующий текст.

```
Это просто тест
```

"Безразмерная" инициализация массивов

Предположим, что мы используем следующий вариант инициализации массивов для построения таблицы сообщений об ошибках.

```
char e1[14] = "Деление на 0\n";
char e2[23] = "Конец файла\n";
char e3[21] = "В доступе отказано\n";
```

Нетрудно предположить, что вручную неудобно подсчитывать символы в каждом сообщении, чтобы определить корректный размер массива. К счастью, в C++ предусмотрена возможность автоматического определения длины массивов путем использования их "безразмерного" формата. Если в инструкции инициализации массива не указан его размер, C++ автоматически создаст массив, размер которого будет достаточным для хранения всех значений инициализаторов. При таком подходе предыдущий вариант инициализации массивов для построения таблицы сообщений об ошибках можно переписать так.

```
char e1[] = "Деление на 0\n";
char e2[] = "Конец файла\n";
char e3[] = "В доступе отказано\n";
```

Помимо удобства в первоначальном определении массивов, метод "безразмерной" инициализации позволяет изменить любое сообщение без пересчета его длины. Тем самым устраняется возможность внесения ошибок, вызванных случайным просчетом.

"Безразмерная" инициализация массивов не ограничивается одномерными массивами. При инициализации многомерных массивов вам необходимо указать все данные, за исключением крайней слева размерности, чтобы C++-компилятор мог должным образом индексировать массив. Используя "безразмерную" инициализацию массивов, можно создавать таблицы различной длины, позволяя компилятору автоматически выделять область памяти, достаточную для их хранения.

В следующем примере массив *sqrs* объявляется как "безразмерный".

```
int sqrs[][2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16,
  5, 25,
  6, 36,
  7, 49,
  8, 64,
  9, 81,
  10, 100
};
```

Преимущество такой формы объявления перед "габаритной" (с точным указанием всех

размерностей) состоит в том, что программист может удлинять или укорачивать таблицу значений инициализации, не изменяя размерности массива.

Массивы строк

Существует специальная форма двумерного символьного массива, которая представляет собой массив строк. В использовании массивов строк нет ничего необычного. Например, в программировании баз данных для выяснения корректности вводимых пользователем команд входные данные сравниваются с содержимым массива строк, в котором записаны допустимые в данном приложении команды. Для создания массива строк используется двумерный символьный массив, в котором размер левого индекса определяет количество строк, а размер правого — максимальную длину каждой строки. Например, при выполнении следующей инструкции объявляется массив, предназначенный для хранения *30* строк длиной *80* символов,

```
char str_array[30][80];
```

Массив строк — *это специальная форма двумерного массива символов*.

Получить доступ к отдельной строке довольно просто: достаточно указать только левый индекс. Например, следующая инструкция вызывает функцию *gets()* для записи третьей строки массива *str_array*.

```
gets(str_array[2]);
```

Чтобы лучше понять, как следует обращаться с массивами строк, рассмотрим следующую короткую программу, которая принимает строки текста, вводимые с клавиатуры, и отображает их на экране после ввода пустой строки.

```
// Вводим строки текста и отображаем их на экране.
#include <iostream>
#include <cstdio>
using namespace std;

int main( )
{
  int t, i;
  char text[100][80];
  for(t=0; t<100; t++) {
    cout << t << ": ";
    gets(text[t]);
    if(!text[t] [0]) break; // Выход из цикла по пустой строке.
  }
  // Отображение строк на экране.
  for(i=0; i<t; i++)
    cout << text[i] << ' \n';

  return 0;
}
```

Обратите внимание на то, как в программе выполняется проверка на ввод пустой строки. Функция *gets()* возвращает строку нулевой длины, если единственной нажатой клавишей оказалась клавиша *<Enter>*. Это означает, что первым байтом в строке будет нулевой символ. Нулевое значение всегда интерпретируется как ложное, но взятое с отрицанием (*!*) дает значение ИСТИНА, которое позволяет выполнить немедленный выход из цикла с помощью инструкции *break*.

помощью инструкции break.

Пример использования массивов строк

Массивы строк обычно используются для обработки таблиц данных. Рассмотрим, например, упрощенную базу данных служащих, в которой хранится имя, номер телефона, количество часов, отработанных служащим за отчетный период, и размер почасового оклада для каждого служащего. Чтобы создать такую программу для коллектива, состоящего из десяти служащих, определим четыре массива (из них первые два будут массивами строк).

```
char name[10][80]; // Массив имен служащих.
char phone[10][20]; // Массив телефонных номеров служащих.
float hours[10]; // Массив часов, отработанных за неделю.
float wage[10]; // Массив окладов.
```

Чтобы ввести информацию о каждом служащем, воспользуемся следующей функцией *enter()*.

```
// Функция ввода информации в базу данных.
void enter()
{
  int i;
  char temp[80];
  for(i=0; i<10; i++) {
    cout << "Введите фамилию: ";
      cin >> name[i];
    cout << "Введите номер телефона: ";
      cin >> phone[i];
    cout << "Введите количество отработанных часов: ";
      cin >> hours[i];
    cout << "Введите оклад: ";
      cin >> wage[i];
  }
}
```

На основании введенных данных можно составить отчет, вычислив заработную плату, которая причитается каждому служащему. Для этого воспользуемся следующей функцией *report()*.

```
// Отображение отчета.
void report()
{
  int i;
  for(i=0; i<10; i++) {
    cout << name[i] << ' ' << phone[i] << '\n';
    cout << "Заработная плата за неделю: "<< wage[i] * hours[i];
    cout << '\n';
  }
}
```

Полностью программа базы данных служащих приведена ниже. Обратите особое внимание на то, как реализуется доступ к каждому массиву. Эта версия программы ведения базы

данных служащих еще далека от совершенства, поскольку введенная в нее информация теряется сразу же по выходу из программы. Ниже в этой книге мы научимся сохранять информацию в дисковом файле.

```
// Простая программа ведения базы данных служащих.
#include <iostream>
using namespace std;

char name[10][80]; // Массив имен служащих.
char phone[10] [20]; // Массив телефонных номеров служащих.
float hours[10]; // Массив часов, отработанных за неделю.
float wage[10]; // Массив окладов.

int menu( );
void enter( ), report( );
int main( )
{
  int choice;
  do {
    choice = menu( ); // Получаем команду, выбранную пользователем.
    switch(choice) {
      case 0: break;
      case 1: enter( );
        break;
      case 2: report( );
        break;
      default: cout << "Попробуйте еще раз.\n\n";
    }
  }while(choice != 0);

  return 0;
}

// Функция возвращает команду, выбранную пользователем.
int menu( )
{
  int choice;
  cout << "0. Выход из программы\n";
  cout << "1. Ввод информации\n";
  cout << "2. Генерирование отчета\n";
  cout << "\n Выберите команду: ";
    cin >> choice;

  return choice;
}
```

```
// Функция ввода информации в базу данных.
void enter()
{
  int i;
  char temp[80];
  for(i=0; i<10; i++) {
    cout << "Введите фамилию: ";
      cin >> name[i];
    cout << "Введите номер телефона: ";
      cin >> phone[i];
    cout << "Введите количество отработанных часов: ";
      cin >> hours[i];
    cout << "Введите оклад: ";
      cin >> wage[i];
  }
}

// Отображение отчета.
void report()
{
  int i;
  for(i=0; i<10; i++) {
    cout << name[i] << ' ' << phone[i] << '\n';
    cout << "Заработная плата за неделю: "<< wage[i] * hours[i];
    cout << '\n';
  }
}
```

Глава 6: Указатели

Указатели, без сомнения, — один из самых важных и сложных аспектов C++. В значительной степени мощь многих средств C++ определяется использованием указателей. Например, благодаря им обеспечивается поддержка связных списков и динамического выделения памяти, и именно они позволяют функциям изменять содержимое своих аргументов. Однако об этом мы поговорим в последующих главах, а пока (т.е. в этой главе) мы рассмотрим основы применения указателей и покажем, как можно избежать некоторых потенциальных проблем, связанных с их использованием.

При рассмотрении темы указателей нам придется использовать такие понятия, как размер базовых C++-типов данных. В этой главе мы предположим, что символы занимают в памяти один байт, целочисленные значения — четыре, значения с плавающей точкой типа float — четыре, а значения с плавающей точкой типа double — восемь (эти размеры характерны для типичной 32-разрядной среды).

Что представляют собой указатели

Указатели — это переменные, которые хранят адреса памяти. Чаще всего эти адреса обозначают местоположение в памяти других переменных. Например, если *x* содержит адрес переменной *y*, то о переменной, *x* говорят, что она *"указывает"* на *y*.

Указатель — *это переменная, которая содержит адрес другой переменной.*

Указатель — *это переменная, которая содержит адрес другой переменной*.

Переменные-указатели (или *переменные типа указатель*) должны быть соответственно объявлены. Формат объявления переменной-указателя таков:

```
тип *имя_переменной;
```

Здесь элемент *тип* означает базовый тип указателя, причем он должен быть допустимым C++-типом. Элемент *имя_переменной* представляет собой имя переменной-указателя. Рассмотрим пример. Чтобы объявить переменную *p* указателем на *int*-значение, используйте следующую инструкцию.

```
int *p;
```

Для объявления указателя на *float*-значение используйте такую инструкцию.

```
float *p;
```

В общем случае использование символа "звездочка" (*) перед именем переменной в инструкции объявления превращает эту переменную в указатель.

Базовый тип указателя определяет тип данных, на которые он будет ссылаться.

Тип данных, на которые будет ссылаться указатель, определяется его базовым типом. Рассмотрим еще один пример.

```
int *ip; // указатель на целочисленное значение
double *dp; // указатель на значение типа double
```

Как отмечено в комментариях, переменная *ip* — это указатель на int-значение, поскольку его базовым типом является тип int, а переменная *dp* — указатель на double-значение, поскольку его базовым типом является тип double, Следовательно, в предыдущих примерах переменную *ip* можно использовать для указания на int-значения, а переменную *dp* на double-значения. Однако помните: не существует реального средства, которое могло бы помешать указателю ссылаться на *"бог-знает-что"*. Вот потому-то указатели потенциально опасны.

Операторы, используемые с указателями

С указателями используются два оператора: *"*"* и *"&"* Оператор *"&"* — унарный. Он возвращает адрес памяти, по которому расположен его операнд. (Вспомните: унарному оператору требуется только один операнд.) Например, при выполнении следующего фрагмента кода

```
balptr = &balance;
```

в переменную *balptr* помещается адрес переменной *balance*. Этот адрес соответствует области во внутренней памяти компьютера, которая принадлежит переменной *balance*. Выполнение этой инструкции никак не повлияло на значение переменной *balance*. Назначение оператора можно *"перевести"* на русский язык как *"адрес переменной*", перед которой он стоит. Следовательно, приведенную выше инструкцию присваивания можно выразить так: *"переменная balptr получает адрес переменной balance"*. Чтобы лучше понять суть этого присваивания, предположим, что переменная *balance* расположена в области памяти с адресом *100*. Следовательно, после выполнения этой инструкции переменная *balptr* получит значение *100*.

Второй оператор работы с указателями (*) служит дополнением к первому (&). Это также унарный оператор, но он обращается к значению переменной, расположенной по адресу, заданному его операндом. Другими словами, он ссылается на значение переменной, адресуемой заданным указателем. Если (продолжая работу с предыдущей инструкцией присваивания) переменная *balptr* содержит адрес переменной *balance*, то при выполнении инструкции

```
value = *balptr;
```

переменной *value* будет присвоено значение переменной *balance*, на которую указывает переменная *balptr*. Например, если переменная *balance* содержит значение *3200*, после выполнения последней инструкции переменная *value* будет содержать значение *3200*, поскольку это как раз то значение, которое хранится по адресу *100*. Назначение оператора *"*"* можно выразить словосочетаинем *"по адресу"*. В данном случае предыдущую инструкцию можно прочитать так: *"переменная value получает значение (расположенное) по адресу balptr"*. Действие приведенных выше двух инструкций схематично показано на рис. 6.1.

Последовательность операций, отображенных на рис. 6.1, реализуется в следующей программе.

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

int main( )
{
  int balance;
  int *balptr;
  int value;

  balance = 3200;
  balptr = &balance;
  value = *balptr;
  cout << "Баланс равен:" << value <<'\n';

  return 0;
}
```

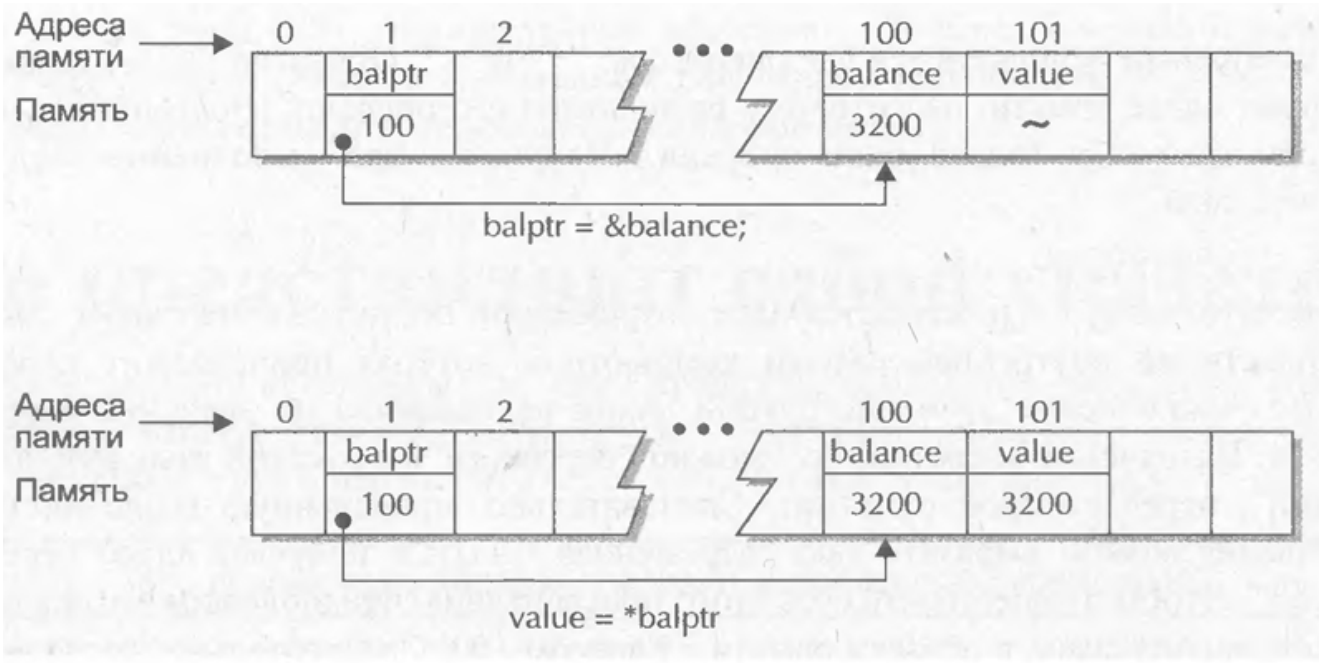


Рис. 6.1. Действие операторов "*" и "&"

При выполнении этой программы получаем такие результаты:

```
Баланс равен: 3200
```

К сожалению, знак умножения (*) и оператор со значением *"по адресу"* обозначаются одинаковыми символами *"звездочка"*, что иногда сбивает с толку новичков в языке C++. Эти операции никак не связаны одна с другой. Имейте в виду, что операторы *"*"* и *"&"* имеют более высокий приоритет, чем любой из арифметических операторов, за исключением унарного минуса, приоритет которого такой же, как у операторов, применяемых для работы с указателями.

Операции, выполняемые с помощью указателей, часто называют операциями непрямого доступа, поскольку мы косвенно получаем доступ к переменной посредством некоторой другой переменной.

Операция непрямого доступа — *это процесс использования указателя для доступа к некоторому объекту*.

О важности базового типа указателя

На примере предыдущей программы была показана возможность присвоения переменной

На примере предыдущей программы была показана возможность присвоения переменной *value* значения переменной *balance* посредством операции непрямого доступа, т.е. с использованием указателя. Возможно, при этом у вас промелькнул вопрос: *"Как C++-компилятор узнает, сколько необходимо скопировать байтов в переменную value из области памяти, адресуемой указателем balptr?"*. Сформулируем тот же вопрос в более общем виде: как C++-компилятор передает надлежащее количество байтов при выполнении операции присваивания с использованием указателя? Ответ звучит так. Тип данных, адресуемый указателем, определяется базовым типом указателя. В данном случае, поскольку *balptr* представляет собой указатель на целочисленный тип, C++-компилятор скопирует в переменную *value* из области памяти, адресуемой указателем *balptr*, четыре байт информации (что справедливо для 32-разрядной среды), но если бы мы имели дело с *double*-указателем, то в аналогичной ситуации скопировалось бы восемь байт.

Переменные-указатели должны всегда указывать на соответствующий тип данных. Например, при объявлении указателя типа *int* компилятор "предполагает", что все значения, на которые ссылается этот указатель, имеют тип *int*. C++-компилятор попросту не позволит выполнить операцию присваивания с участием указателей (с обеих сторон от оператора присваивания), если типы этих указателей несовместимы (по сути не одинаковы).

Например, следующий фрагмент кода некорректен.

```
int *p;
double f;
// ...
p = &f; // ОШИБКА!
```

Некорректность этого фрагмента состоит в недопустимости присваивания *double*-указателя *int*-указателю. Выражение *&f* генерирует указатель на *double*-значение, а *p* — указатель на целочисленный тип *int*. Эти два типа несовместимы, поэтому компилятор отметит эту инструкцию как ошибочную и не скомпилирует программу.

Несмотря на то что, как было заявлено выше, при присваивании два указателя должны быть совместимы по типу, это серьезное ограничение можно преодолеть (правда, на свой страх и риск) с помощью операции приведения типов. Например, следующий фрагмент кода теперь формально корректен.

```
int *p;
double f;
// ...
p = (int *) &f; // Теперь формально все ОК!
```

Операция приведения к типу (*int **) вызовет преобразование *double*- к *int*-указателю. Все же использование операции приведения в таких целях несколько сомнительно, поскольку именно базовый тип указателя определяет, как компилятор будет обращаться с данными, на которые он ссылается. В данном случае, несмотря на то, что *p* (после выполнения последней инструкции) в действительности указывает на значение с плавающей точкой, компилятор по-прежнему "считает", что он указывает на целочисленное значение (поскольку *p* по определению — int-указатель).

Чтобы лучше понять, почему использование операции приведения типов при присваивании одного указателя другому не всегда приемлемо, рассмотрим следующую программу.

```
// Эта программа не будет выполняться правильно.
#include <iostream>
using namespace std;

int main()
{
```

```
  double x, y;
  int *p;
  x = 123.23;
   p = (int *) &x; // Используем операцию приведения типов для присваивания double-указателя int-указателю.
  y = *p; // Что происходит при выполнении этой инструкции?
  cout << y; // Что выведет эта инструкция?

  return 0;
}
```

Как видите, в этой программе переменной *p* (точнее, указателю на целочисленное значение) присваивается адрес переменной *x* (которая имеет тип double). Следовательно, когда переменной *y* присваивается значение, адресуемое указателем *p*, переменная *y* получает только четыре байт данных (а не все восемь, требуемые для double-значения), поскольку *p*— указатель на целочисленный тип int. Таким образом, при выполнении cout-инструкции на экран будет выведено не число *123.23*, а, как говорят программисты, *"мусор"*. (Выполните программу и убедитесь в этом сами.)

Присваивание значений с помощью указателей

При присваивании значения области памяти, адресуемой указателем, его (указатель) можно использовать с левой стороны от оператора присваивания. Например, при выполнении следующей инструкции (если *p* — указатель на целочисленный тип)

```
*p = 101;
```

число *101* присваивается области памяти, адресуемой указателем *p*. Таким образом, эту инструкцию можно прочитать так: *"по адресу p помещаем значение 101"*. Чтобы инкрементировать или декрементировать значение, расположенное в области памяти, адресуемой указателем, можно использовать инструкцию, подобную следующей.

```
(*p)++;
```

Круглые скобки здесь обязательны, поскольку оператор *"*"* имеет более низкий приоритет, чем оператор *"++"*.

Присваивание значений с использованием указателей демонстрируется в следующей программе.

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

int main( )
{
  int *p, num;
  p = &num;
  *p = 100;
  cout << num << ' ';
  (*p)++;
  cout << num << ' ';
  (*p)--;
  cout << num << '\n';

  return 0;
}
```

Вот такие результаты генерирует эта программа

Вот такие результаты генерирует эта программа.

```
100 101 100
```

Использование указателей в выражениях

Указатели можно использовать в большинстве допустимых в C++ выражений. Но при этом следует применять некоторые специальные правила и не забывать, что некоторые части таких выражений необходимо заключать в круглые скобки, чтобы гарантированно получить желаемый результат.

Арифметические операции над указателями

С указателями можно использовать только четыре арифметических оператора: *++*, --, *+* и -. Чтобы лучше понять, что происходит при выполнении арифметических действий с указателями, начнем с примера. Пусть *p1* — указатель на int-переменную с текущим значением *2 ООО* (т.е. p1 содержит адрес 2 ООО). После выполнения (в 32-разрядной среде) выражения

```
p1++;
```

содержимое переменной-указателя *p1* станет равным *2 004*, а не *2 001*! Дело в том, что при каждом инкрементировании указатель *p1* будет указывать на следующее int-значение. Для операции декрементирования справедливо обратное утверждение, т.е. при каждом декрементировании значение *p1* будет уменьшаться на *4*. Например, после выполнения инструкции

```
p1--;
```

указатель *p1* будет иметь значение *1 996*, если до этого оно было равно *2 000*. Итак, каждый раз, когда указатель инкрементируется, он будет указывать на область памяти, содержащую следующий элемент базового типа этого указателя. А при каждом декрементировании он будет указывать на область памяти, содержащую предыдущий элемент базового типа этого указателя.

Для указателей на символьные значения результат операций инкрементирования и декрементирования будет таким же, как при "нормальной" арифметике, поскольку символы занимают только один байт. Но при использовании любого другого типа указателя при инкрементировании или декрементировании значение переменной-указателя будет увеличиваться или уменьшаться на величину, равную размеру его базового типа.

Арифметические операции над указателями не ограничиваются использованием операторов инкремента и декремента. Со значениями указателей можно выполнять операции сложения и вычитания, используя в качестве второго операнда целочисленные значения. Выражение

```
p1 = p1 + 9;
```

заставляет *p1* ссылаться на девятый элемент базового типа указателя *p1* относительно элемента, на который *p1* ссылался до выполнения этой инструкций.

Несмотря на то что складывать указатели нельзя, один указатель можно вычесть из другого (если они оба имеют один и тот же базовый тип). Разность покажет количество элементов базового типа, которые разделяют эти два указателя.

Помимо сложения указателя с целочисленным значением и вычитания его из указателя, а также вычисления разности двух указателей, над указателями никакие другие арифметические операции не выполняются. Например, с указателями нельзя складывать float- или double-значения.

Чтобы понять, как формируется результат выполнения арифметических операций над указателями, выполним следующую короткую программу. Она выводит реальные физические адреса, которые содержат указатель на *int*-значение (*i*) и указатель на *float*-значение (*f*). Обратите внимание на каждое изменение адреса (зависящее от базового типа указателя), которое происходит при повторении цикла. (Для большинства 32-разрядных компиляторов значение *i* будет увеличиваться на 4, а значение *f* — на 8.) Отметьте также, что при использовании указателя в cout-инструкции его адрес автоматически отображается в

при использовании указателя в cout-инструкции его адрес автоматически отображается в формате адресации, применяемом для текущего процессора и среды выполнения.

```
// Демонстрация арифметических операций над указателями.
#include <iostream>
using namespace std;

int main()
{
  int *i, j[10];
  double *f, g[10];
  int x;
  i = j;
  f = g;
  for(x=0; x<10; x++)
    cout << i+x << ' ' << f+x << '\n';

  return 0;
}
```

Вот как выглядит возможный вариант выполнения этой программы (ваши результаты могут отличаться от приведенных, но интервалы между значениями должны быть такими же).

```
0012FE5C 0012FE84
0012FE60 0012FE8C
0012FE64 0012FE94
0012FE68 0012FE9C
0012FE6C 0012FEA4
0012FE70 0012FEAC
0012FE74 0012FEB4
0012FE78 0012FEBC
0012FE7C 0012FEC4
0012FE80 0012FECC
```

Узелок на память. *Все арифметические операции над указателями выполняются относительно базового типа указателя*.

Сравнение указателей

Указатели можно сравнивать, используя операторы отношения ==, < и >. Однако для того, чтобы результат сравнения указателей поддавался интерпретации, сравниваемые указатели должны быть каким-то образом связаны. Например, если *p1* и *p2* — указатели, которые ссылаются на две отдельные и никак не связанные переменные, то любое сравнение *p1* и *p2* в общем случае не имеет смысла. Но если *p1* и *p2* указывают на переменные, между которыми существует некоторая связь (как, например, между элементами одного и того же массива), то результат сравнения указателей *p1* и *p2* может иметь определенный смысл. Ниже в этой главе мы рассмотрим пример программы, в которой используется сравнение указателей.

Указатели и массивы

В C++ указатели и массивы тесно связаны между собой, причем настолько, что зачастую понятия *"указатель"* и *"массив"* взаимозаменяемы. В этом разделе мы попробуем проследить эту связь. Для начала рассмотрим следующий фрагмент программы:

проследить эту связь. Для начала рассмотрим следующий фрагмент программы.

```
char str[80];
char *p1;
p1 = str;
```

Здесь *str* представляет собой имя массива, содержащего *80* символов, а *p1* — указатель на тип *char*. Особый интерес представляет третья строка, при выполнении которой переменной *p1* присваивается адрес первого элемента массива *str*. (Другими словами, после этого присваивания *p1* будет указывать на элемент *str[0]*.) Дело в том, что в C++ использование имени массива без индекса генерирует указатель на первый элемент этого массива. Таким образом, при выполнении присваивания *p1 = str* адрес *str[0]* присваивается указателю *p1*. Это и есть ключевой момент, который необходимо четко понимать: неиндексированное имя массива, использованное в выражении, означает указатель на начало этого массива.

Имя массива без индекса образует указатель на начало этого массива.

Поскольку после рассмотренного выше присваивания *p1* будет указывать на начало массива *str*, указатель *p1* можно использовать для доступа к элементам этого массива. Например, если нужно получить доступ к пятому элементу массива *str*, используйте одно из следующих выражений:

```
str[4]
```

или

```
*(p1+4)
```

В обоих случаях будет выполнено обращение к пятому элементу. Помните, что индексирование массива начинается с нуля, поэтому при индексе, равном четырем, обеспечивается доступ к пятому элементу. Точно такой же эффект производит суммирование значения исходного указателя (*p1*) с числом *4*, поскольку *p1* указывает на первый элемент массива *str*.

Необходимость использования круглых скобок, в которые заключено выражение *p1+4*, обусловлена тем, что оператор *"*"* имеет более высокий приоритет, чем оператор *"+"*. Без этих круглых скобок выражение бы свелось к получению значения, адресуемого указателем *p1*, т.е. значения первого элемента массива, которое затем было бы увеличено на *4*.

Важно! *Убедитесь лишний раз в правильности использования круглых скобок в выражении с указателями. В противном случае ошибку будет трудно отыскать, поскольку внешне программа может выглядеть вполне корректной. Если у вас есть сомнения в необходимости их использования, примите решение в их пользу — вреда от этого не будет*.

В действительности в C++ предусмотрено два способа доступа к элементам массивов: с помощью индексирования массивов и арифметики указателей. Дело в том, что арифметические операции над указателями иногда выполняются быстрее, чем индексирование массивов, особенно при доступе к элементам, расположение которых отличается строгой упорядоченностью. Поскольку быстродействие часто является определяющим фактором при выборе тех или иных решений в программировании, то использование указателей для доступа к элементам массива— характерная особенность многих C++-программ. Кроме того, иногда указатели позволяют написать более компактный код по сравнению с использованием индексирования массивов.

Чтобы лучше понять различие между использованием индексирования массивов и арифметических операций над указателями, рассмотрим две версии одной и той же программы. В этой программе из строки текста выделяются слова, разделенные пробелами. Например, из строки *"Привет дружище"* программа должна выделить слова *"Привет"* и *"дружище"*. Программисты обычно называют такие разграниченные символьные последовательности лексемами (token). При выполнении программы входная строка посимвольно копируется в другой массив (с именем token) до тех пор, пока не встретится пробел. После этого выделенная лексема выводится на экран, и процесс продолжается до

тех пор, пока не будет достигнут конец строки. Например, если в качестве входной строки использовать строку *Это лишь простой тест.*, программа отобразит следующее.

```
Это
лишь
простой
тест.
```

Вот как выглядит версия программы разбиения строки на слова с использованием арифметики указателей.

```
// Программа разбиения строки на слова:
// версия с использованием указателей.
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
  char str[80];
  char token[80];
  char *p, *q;

  cout << "Введите предложение: ";
  gets(str);
  p = str;
  // Считываем лексему из строки.
  while(*p) {
    q = token; // Устанавливаем q для указания на начало массива token.
    /* Считываем символы до тех пор, пока не встретится либо пробел, либо
нулевой символ (признак завершения строки). */
    while(*p != ' ' && *p) {
      *q = *p;
      q++; p++;
    }
    if(*p) p++; // Перемещаемся за пробел.
    *q = '\0'; // Завершаем лексему нулевым символом.
    cout << token << '\n';
  }
  return 0;
}
```

А вот как выглядит версия той же программы с использованием индексирования массивов.

```
// Программа разбиения строки на слова:
// версия с использованием индексирования массивов.
#include <iostream>
#include <cstdio>
```

```
using namespace std;

int main()
{
  char str[80];
  char token[80];
  int i, j;
  cout << "Введите предложение: ";
  gets(str);
  // Считываем лексему из строки.
  for(i=0; ; i++) {
    /* Считываем символы до тех пор пока не встретится либо пробел, либо нулевой символ (признак завершения строки). */
    for(j=0; str[i]!=' ' && str[i]; j++, i++)
      token[j] = str[i];
    token[j] = '\0'; // Завершаем лексему нулевым символом.
    cout << token << '\n';
    if(!str[i]) break;
  }
  return 0;
}
```

У этих программ может быть различное быстродействие, что обусловлено особенностями генерирования кода C++-компиляторами. Как правило, при использовании индексирования массивов генерируется более длинный код (с большим количеством машинных команд), чем при выполнении арифметических действий над указателями. Поэтому неудивительно, что в профессионально написанном C++-коде чаще встречаются версии, ориентированные на обработку указателей. Но если вы— начинающий программист, смело используйте индексирование массивов, пока не научитесь свободно обращаться с указателями.

Индексирование указателя

Как было показано выше, можно получить доступ к массиву, используя арифметические действия над указателями. Интересно то, что в C++ указатель, Который ссылается на массив, можно индексировать так, как если бы это было имя массива (это говорит о тесной связи между указателями и массивами). Соответствующий такому подходу синтаксис обеспечивает альтернативу арифметическим операциям над указателями, поскольку он более удобен в некоторых ситуациях. Рассмотрим пример.

```
// Индексирование указателя подобно массиву.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
  char str[20] = "I love you";
  char *p;
  int i;
```

```
  p = str;
  // Индексируем указатель.
  for(i=0; p[i]; i++) p[i] = toupper(p[i]);
  cout << p; // Отображаем строку.
  return 0;
}
```

При выполнении программа отобразит на экране следующее.

```
I LOVE YOU
```

Вот как работает эта программа. Сначала в массив *str* вводится строка *"I love you"*. Затем адрес начала этой строки присваивается указателю *p*. После этого каждый символ строки *str* с помощью функции *toupper()* преобразуется в его прописной эквивалент посредством индексирования указателя *p*. Помните, что выражение *p[i]* по своему действию идентично выражению **(p+i)*.

О взаимозаменяемости указателей и массивов

Выше было показано, что указатели и массивы очень тесно связаны. И действительно, во многих случаях они взаимозаменяемы. Например, с помощью указателя, который содержит адрес начала массива, можно получить доступ к элементам этого массива либо посредством арифметических действий над указателем, либо посредством индексирования массива. Однако в общем случае указатели и массивы не являются взаимозаменяемыми. Рассмотрим, например, такой фрагмент кода.

```
int num[10];
int i;
for(i=0; i<10; i++) {
  *num = i; // Здесь все в порядке.
  num++; // ОШИБКА — переменную num модифицировать нельзя.
}
```

Здесь используется массив целочисленных значений с именем *num*. Как отмечено в комментарии, несмотря на то, что совершенно приемлемо применить к имени *num* оператор "*" (который обычно применяется к указателям), абсолютно недопустимо модифицировать значение *num*. Дело в том, что *num* — это константа, которая указывает на начало массива. И ее, следовательно, инкрементировать никак нельзя. Другими словами, несмотря на то, что имя массива (без индекса) действительно генерирует указатель на начало массива, его значение изменению не подлежит.

Хотя имя массива генерирует константу-указатель, на него, тем не менее, (подобно указателям) можно включать в выражения, если, конечно, оно при этом не модифицируется. Например следующая инструкция, при выполнении которой элементу *num[3]* присваивается значение *100*, вполне допустима.

```
*(num+3) = 100; // Здесь все в порядке, поскольку num не изменяется.
```

Указатели и строковые литералы

Возможно, вас удивит способ обработки C++-компиляторами строковых литералов, подобных следующему.

```
cout << strlen("C++-компилятор");
```

Если C++-компилятор обнаруживает строковый литерал, он сохраняет его в таблице строк программы и генерирует указатель на нужную строку. Поэтому следующая программа совершенно корректна и при выполнении выводит на экран фразу: *Работа с указателями - сплошное удовольствие!*.

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

int main()
{
  char *s;
  s = "Работа с указателями - сплошное удовольствие!\n";
  cout << s;
  return 0;
}
```

При выполнении этой программы символы, образующие строковый литерал, сохраняются в таблице строк, а переменной *s* присваивается указатель на соответствующую строку в этой таблице.

Таблица строк — *это таблица, сгенерированная компилятором для хранения строк, используемых в программе*.

Поскольку указатель на таблицу строк конкретной программы при использовании строкового литерала генерируется автоматически, то можно попытаться использовать этот факт для модификации содержимого данной таблицы. Однако такое решения вряд ли можно назвать удачным. Дело в том, что C++-компиляторы создают оптимизированные таблицы, в которых один строковый литерал может использоваться в двух (или больше) различных местах программы. Поэтому "насильственное" изменение строки может вызвать нежелательные побочные эффекты. Более того, строковые литералы представляют собой константы, и некоторые современные C++-компиляторы попросту не позволят менять их содержимое. А при попытке сделать это будет сгенерирована ошибка времени выполнения.

Все познается в сравнении

Выше отмечалось, что значение одного указателя можно сравнивать с другим. Но, чтобы сравнение указателей имело смысл, сравниваемые указатели должны быть каким-то образом связаны друг с другом. Чаще всего такая связь устанавливается в случае, когда оба указателя указывают на элементы одного и того же массива. Например, даны два указателя (с именами *A* и *B*), которые ссылаются на один и тот же массив. Если *A* меньше *B*, значит, указатель *A* указывает на элемент, индекс которого меньше индекса элемента, адресуемого указателем *B*. Такое сравнение особенно полезно для определения граничных условий.

Сравнение указателей демонстрируется в следующей программе. В этой программе создается две переменных типа указатель. Одна (с именем *start*) первоначально указывает на начало массива, а вторая (с именем *end*) — на его конец. По мере ввода пользователем чисел массив последовательно заполняется от начала к концу. Каждый раз, когда в массив вводится очередное число, указатель *start* инкрементируется. Чтобы определить, заполнился ли массив, в программе просто сравниваются значения указателей *start* и *end*. Когда *start* превысит *end*, массив будет заполнен "до отказа". Программе останется лишь вывести содержимое заполненного массива на экран.

```
// Пример сравнения указателей.
#include <iostream>
using namespace std;

int main()
{
  int num[10];
  int *start, *end;
  start = num;
  end = &num[9];
```

```
  end = &num[9];
  while(start <= end) {
    cout << "Введите число: ";
      cin >> *start;
    start++;
  }
  start << num; /* Восстановление исходного значения указателя */
  while(start <= end) {
    cout << *start << ' ';
    start++;
  }
  return 0;
}
```

Как показано в этой программе, поскольку *start* и *end* оба указывают на общий объект (в данном случае им является массив *num*), их сравнение может иметь смысл. Подобное сравнение часто используется в профессионально написанном C++-коде.

Массивы указателей

Указатели, подобно данным других типов, могут храниться в массивах. Вот, например, как выглядит объявление 10-элементного массива указателей на int-значения.

```
int *ipa[10];
```

Здесь каждый элемент массива *ipa* содержит указатель на целочисленное значение.

Чтобы присвоить адрес int-переменной с именем *var* третьему элементу этого массива указателей, запишите следующее.

```
ipa[2] = &var;
```

Помните, что здесь *ipa* — массив указателей на целочисленные значения. Элементы этого массива могут содержать только значения, которые представляют собой адреса переменных целочисленного типа. Вот поэтому переменная *var* предваряется оператором Чтобы присвоить значение переменной *var* целочисленной переменной *x* с помощью массива *ipa*, используйте такой синтаксис.

```
x = *ipa[2];
```

Поскольку адрес переменной *var* хранится в элементе *ipa[2]*, применение оператора *"*"* к этой индексированной переменной позволит получить значение переменной *var*.

Подобно другим массивам, массивы указателей можно инициализировать. Как правило, инициализированные массивы указателей используются для хранения указателей на строки. Например, чтобы создать функцию, которая выводит счастливые предсказания, можно следующим образом определить массив *fortunes*,

```
char *fortunes[] = {
  "Вскоре деньги потекут к Вам рекой.\n",
  "Вашу жизнь озарит новая любовь.\n",
  "Вы будете жить долго и счастливо.\n",
  "Деньги, вложенные сейчас в дело, принесут доход.\n",
  "Близкий друг будет искать Вашего расположения.\n"
};
```

Не забывайте, что C++ обеспечивает хранение всех строковых литералов в таблице строк,

связанной с конкретной программой, поэтому массив нужен только Для хранения указателей на эти строки. Таким образом, для вывода второго сообщения достаточно использовать инструкцию, подобную следующей.

```
cout << fortunes[1];
```

Ниже программа предсказаний приведена целиком. Для получения случайных чисел используется функция *rand()*, а для получения случайных чисел в диапазоне от *0* до *4* — оператор деления по модулю, поскольку именно такие числа могут служить для доступа к элементам массива по индексу.

```
#include <iostream>
#include <cstdlib>
#include <conio.h>
using namespace std;

char *fortunes[] = {
  "Вскоре деньги потекут к Вам рекой.\n",
  "Вашу жизнь озарит новая любовь.\n",
  "Вы будете жить долго и счастливо.\n",
  "Деньги, вложенные сейчас в дело, принесут доход.\n",
  "Близкий друг будет искать Вашего расположения.\n"
};

int main( )
{
  int chance;
  cout <<"Чтобы узнать свою судьбу, нажмите любую клавишу: ";
  // Рандомизируем генератор случайных чисел.
  while(!kbhit( )) rand( );
  cout << '\n';
  chance = rand( );
  chance = chance % 5;
  cout << fortunes[chance];

  return 0;
}
```

Обратите внимание на цикл *while*, который вызывает функцию *rand()* до тех пор, пока не будет нажата какая-либо клавиша. Поскольку функция *rand()* всегда генерирует одну и ту же последовательность случайных чисел, важно иметь возможность программно использовать эту последовательность с некоторой произвольной позиции. (В противном случае каждый раз после запуска программа будет выдавать одно и то же "предсказание".) Эффект случайности достигается за счет повторяющихся обращений к функции *rand()*. Когда пользователь нажмет клавишу, цикл остановится на некоторой, случайной позиции последовательности генерируемых чисел, и эта позиция определит номер сообщения, которое будет выведено на экран. Напомню, что функция *kbhit()* представляет собой довольно распространенное расширение библиотеки функций C++, обеспечиваемое многими компиляторами, но не входит в стандартный пакет библиотечных функций C++.

В следующем примере используется двумерный массив указателей для создания программы, которая отображает синтаксис-памятку по ключевым словам C++. В программе

инициализируется список указателей на строки. Первая размерность массива предназначена для указания на ключевые слова C++, а вторая — на краткое их описание. Список завершается двумя нулевыми строками, которые используются в качестве признака конца списка. Пользователь вводит ключевое слово, а программа должна вывести на экран его описание. Как видите, этот список содержит всего несколько ключевых слов. Поэтому его продолжение остается за вами.

```
// Простая памятка по ключевым словам C++.
#include <iostream>
#include <cstring>
using namespace std;

char *keyword[][2] = {
  "for", "for(инициализация; условие; инкремент)",
  "if", "if(условие) ... else ... ",
  "switch", "switch(значение) {case-список}",
  "while", "while(условие) ...",
  // Сюда нужно добавить остальные ключевые слова C++.
  "", "" // Список должен завершаться нулевыми строками.
};

int main()
{
  char str[80];
  int i;
  cout << "Введите ключевое слово: ";
    cin >> str;
  // Отображаем синтаксис.
  for(i=0; *keyword[i][0]; i++)
    if(!strcmp(keyword[i][0], str))
      cout << keyword[i][1];
  return 0;
}
```

Вот пример выполнения этой программы.

```
Введите ключевое слово: for
for (инициализация; условие; инкремент)
```

В этой программе обратите внимание на выражение, управляющее циклом *for*. Оно приводит к завершению цикла, когда элемент *keyword[i][0]* содержит указатель на нуль, который интерпретируется как значение ЛОЖЬ. Следовательно, цикл останавливается, когда встречается нулевая строка, которая завершает массив указателей.

Соглашение о нулевых указателях

Объявленный, но не инициализированный указатель будет содержать произвольное значение. При попытке использовать указатель до присвоения ему конкретного значения можно разрушить не только собственную программу, но даже и операционную систему (отвратительнейший, надо сказать, тип ошибки!). Поскольку не существует гарантированного способа избежать использования неинициализированного указателя, C++-программисты приняли процедуру, которая позволяет избегать таких ужасных ошибок. По соглашению, если указатель содержит нулевое значение, считается, что он ни на что не

По соглашению, если указатель содержит нулевое значение, считается, что он ни на что не ссылается. Это значит, что, если всем неиспользуемым указателям присваивать нулевые значения и избегать использования нулевых указателей, можно избежать случайного использования неинициализированного указателя. Вам следует придерживаться этой практики программирования.

При объявлении указатель любого типа можно инициализировать нулевым значением, например, как это делается в следующей инструкции,

```
float *p = 0; // p — теперь нулевой указатель.
```

Для тестирования указателя используется инструкция if (любой из следующих ее вариантов).

```
if(p) // Выполняем что-то, если p — не нулевой указатель.
```

```
if(!p) // Выполняем что-то, если p — нулевой указатель.
```

Соблюдая упомянутое выше соглашение о нулевых указателях, вы можете избежать многих серьезных проблем, возникающих при использование указателей.

Указатели и 16-разрядные среды

Несмотря на то что в настоящее время большинство вычислительных сред 32-разрядные, все же немало пользователей до сих пор работают в 16-разрядных (в основном, это DOS и Windows 3.1) и, естественно, с 16-разрядным кодом. Эти операционные системы были разработаны для процессоров семейства Intel 8086, которые включают такие модификации, как 80286, 80386, 80486 и Pentium (при работе в режиме эмуляции процессора 8086). И хотя при написании нового кода программисты, как правило, ориентируются на использование 32-разрядной среды выполнения, все же некоторые программы по-прежнему создаются и поддерживаются в более компактных 16-разрядных средах. Поскольку некоторые темы актуальны только для 16-разрядных сред, программистам, которые работают в них, будет полезно получить информацию о том, как адаптировать "старый" код к новой среде, т.е. переориентировать 16-разрядный код на 32-разрядный.

При написании 16-разрядного кода для процессоров семейства Intel 8086 программист вправе рассчитывать на шесть способов компиляции программ, которые различаются организацией компьютерной памяти. Программы можно компилировать для миниатюрной, малой, средней, компактной, большой и огромной моделей памяти. Каждая из этих моделей по-своему оптимизирует пространство, резервируемое для данных, кода и стека. Различие в организации компьютерной памяти объясняется использованием процессорами семейства Intel 8086 сегментированной архитектуры при выполнении 16-разрядного кода. В 16-разрядном сегментированном режиме процессоры семейства Intel 8086 делят память на 16K сегментов.

В некоторых случаях модель памяти может повлиять на поведение указателей и ваши возможности по их использованию. Основная проблема возникает при инкрементировании указателя за пределы сегмента. Рассмотрение особенностей каждой из 16-разрядных моделей памяти выходит за рамки этой книги. Главное, чтобы вы знали, что, если вам придется работать в 16-разрядной среде и ориентироваться на процессоры семейства Intel 8086, вы должны изучить документацию, прилагаемую к используемому вами компилятору, и подробно разобраться в моделях памяти и их влиянии на указатели.

И последнее. При написании программ для современной 32-разрядной среды необходимо знать, что в ней используется единственная модель организации памяти, которая называется *одноуровневой*, *несегментированной* или *линейной* (flat model).

Многоуровневая непрямая адресация

Можно создать указатель, который будет ссылаться на другой указатель, а тот — на конечное значение. Эту ситуацию называют *многоуровневой непрямой адресацией* (multiple indirection) или использованием указателя на указатель. Идея *многоуровневой непрямой адресации* схематично проиллюстрирована на рис. 6.2. Как видите, значение обычного указателя (при одноуровневой непрямой адресации) представляет собой адрес переменной, которая содержит некоторое значение. В случае применения указателя на

переменной, которая содержит некоторое значение. В случае применения указателя на указатель первый содержит адрес второго, а тот ссылается на переменную, содержащую определенное значение.



Рис. 6.2. Одноуровневая и многоуровневая непрямая адресация

При использовании непрямой адресации можно организовать любое желаемое количество уровней, но, как правило, ограничиваются лишь двумя, поскольку увеличение числа уровней часто ведет к возникновению концептуальных ошибок.

Переменную, которая является указателем на указатель, нужно объявить соответствующим образом. Для этого достаточно перед ее именем поставить дополнительный символ "звездочка"(*). Например, следующее объявление сообщает компилятору о том, что *balance* — это указатель на указатель на значение типа *int*.

```
int **balance;
```

Необходимо помнить, что переменная *balance* здесь — не указатель на целочисленное значение, а указатель на указатель на int-значение.

Чтобы получить доступ к значению, адресуемому указателем на указатель, необходимо дважды применить оператор "*" как показано в следующем коротком примере.

```
// Использование многоуровневой непрямой адресации.
#include <iostream>
using namespace std;

int main( )
{
  int x, *p, **q;
  x = 10;
  p = &x;
  q = &p;
  cout << **q; // Выводим значение переменной x.

  return 0;
}
```

Здесь переменная *p* объявлена как указатель на int-значеине, а переменная *q* — как указатель на указатель на int-значеине. При выполнении этой программы мы получим значение переменной *x*, т.е. число *10*.

Проблемы, связанные с использованием указателей

Для программиста нет ничего более страшного, чем "взбесившиеся" указатели! Указатели можно сравнить с энергией атома: они одновременно и чрезвычайно полезны и чрезвычайно опасны. Если проблема связана с получением указателем неверного значения, то такую ошибку отыскать труднее всего.

Трудности выявления ошибок, связанных с указателями, объясняются тем, что сам по себе указатель не обнаруживает проблему. Проблема может проявиться только косвенно, возможно, даже в результате выполнения нескольких инструкций после "крамольной" операции с указателем. Например, если один указатель случайно получит адрес "не тех" данных, то при выполнении операции с этим "сомнительным" указателем адресуемые данные могут подвергнуться нежелательному изменению, и, что самое здесь неприятное, это "тайное" изменение, как правило, становится явным гораздо позже. Такое "запаздывание" существенно усложняет поиск ошибки, поскольку посылает вас по "ложному следу". К тому моменту, когда проблема станет очевидной, вполне возможно, что указатель-виновник внешне будет выглядеть "безобидной овечкой", и вам придется затратить еще немало времени, чтобы найти истинную причину проблемы.

Поскольку для многих работать с указателями — значит потенциально обречь себя на поиски ответа на вопрос "Кто виноват?", мы попытаемся рассмотреть возможные "овраги" на пути отважного программиста и показать обходные пути, позволяющие избежать изматывающих "мук творчества".

Неинициализированные указатели

Классический пример ошибки, допускаемой при работе с указателями, — использование *неинициализированного указателя*. Рассмотрим следующий фрагмент кода.

```
// Эта программа некорректна.
int main( )
{
  int x, *p;
  x = 10;
  *p = x; // На что указывает переменная p?
  return 0;
}
```

Здесь указатель *p* содержит неизвестный адрес, поскольку он нигде не определен. У вас нет возможности узнать, где записано значение переменной *x*. При небольших размерах программы (например, как в данном случае) ее странности (которые заключаются в том, что указатель *p* содержит адрес, не принадлежащий ни коду программы, ни области данных) могут никак не проявляться, т.е. программа внешне будет работать нормально. Но по мере ее развития и, соответственно, увеличения ее объема, вероятность того, что *p* станет указывать либо на код программы, либо на область данных, возрастет. В один прекрасный день программа вообще перестанет работать. Способ не допустить создания таких программ очевиден: прежде чем использовать указатель, позаботьтесь о том, чтобы он ссылался на что-нибудь действительное!

Некорректное сравнение указателей

Сравнение указателей, которые не ссылаются на элементы одного и того же массива, в общем случае некорректно и часто приводит к возникновению ошибок. Никогда не стоит полагаться на то, что различные объекты будут размещены в памяти каким-то определенным образом (где-то рядом) или на то, что все компиляторы и операционные

среды будут обрабатывать ваши данные одинаково. Поэтому любое сравнение указателей, которые ссылаются на различные объекты, может привести к неожиданным последствиям. Рассмотрим пример.

```
char s[80];
char y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1 < p2) . . .
```

Здесь используется некорректное сравнение указателей, поскольку C++ не дает никаких гарантий относительно размещения переменных в памяти. Ваш код должен быть написан таким образом, чтобы он работал одинаково устойчиво вне зависимости от того, где расположены данные в памяти.

Было бы ошибкой предполагать, что два объявленных массива будут расположены в памяти "плечом к плечу", и поэтому можно обращаться к ним путем индексирования их с помощью одного и того же указателя. Предположение о том, что инкрементируемый указатель после выхода за границы первого массива станет ссылаться на второй, совершенно ни на чем не основано и потому неверно. Рассмотрим этот пример внимательно.

```
int first[101;
int second[10];
int *p, t;
p = first;
for(t=0; t<20; ++t) {
  *p = t;
  p++;
}
```

Цель этой программы — инициализировать элементы массивов *first* и *second* числами от *0* до *19*. Однако этот код не позволяет надеяться на достижение желаемого результата, несмотря на то, что в некоторых условиях и при использовании определенных компиляторов эта программа будет работать так, как задумано автором. Не стоит полагаться на то, что массивы *first* и *second* будут расположены в памяти компьютера последовательно, причем первым обязательно будет массив *first*. Язык C++ не гарантирует определенного расположения объектов в памяти, и потому эту программу нельзя считать корректной.

Не забывайте об установке указателей

Следующая (некорректная) программа должна принять строку, введенную с клавиатуры, а затем отобразить *ASCII*-код для каждого символа этой строки. (Обратите внимание на то, что для вывода *ASCII*-кодов на экран используется операция приведения типов.) Однако эта программа содержит серьезную ошибку.

```
// Эта программа некорректна.
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int main()
```

```
{
  char s [80];
  char *p1;
  p1 = s;
  do {
    cout << "Введите строку: ";
    gets(p1); // Считываем строку.

    // Выводим ASCII-значения каждого символа.
    while(*p1) cout << (int) *p1++ << ' ';
    cout << ' \n';
  }while(strcmp (s, "конец"));

  return 0;
}
```

Сможете ли вы сами найти здесь ошибку?

В приведенном выше варианте программы указателю *p1* присваивается адрес массива *s* только один раз. Это присваивание выполняется вне цикла. При входе в *do-while*-цикл (т.е. при первой его итерации) *p1* действительно указывает на первый символ массива *s*. Но при втором проходе того же цикла *p1* указатель будет содержать значение, которое останется после выполнения предыдущей итерации цикла, поскольку указатель *p1* не устанавливается заново на начало массива *s*. Рано или поздно граница массива *s* будет нарушена.

Вот как выглядит корректный вариант той же программы.

```
// Эта программа корректна.
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

int main( )
{
  char s[80];
  char *p1;
  do {
    p1 = s; // Устанавливаем p1 при каждой итерации цикла.
    cout << "Введите строку: ";
    gets(p1); // Считываем строку.

    // Выводим ASCII-значения каждого символа.
    while(*p1) cout << (int) *p1++ <<' ';
    cout << '\n';
  }while(strcmp(s, "конец"));

  return 0;
}
```

Итак, в этом варианте программы в начале каждой итерации цикла указатель *p1* устанавливается на начало строки.

Узелок на память. *Чтобы использование указателей было безопасным, нужно в любой момент знать, на что они ссылаются.*

Глава 7:Функции, часть первая: ОСНОВЫ

В этой главе мы приступаем к углубленному рассмотрению функций. Функции — это строительные блоки C++, а потому без полного их понимания невозможно стать успешным C++-программистом. Мы уже коснулись темы функций в главе 2 и использовали их в каждом примере программы. В этой главе мы познакомимся с ними более детально. Данная тема включает рассмотрение правил действия областей видимости функций, рекурсивных функций, некоторых специальных свойств функции main(), инструкции return и прототипов функций.

Правила действия областей видимости функций

Правила действия областей видимости определяют возможность получения доступа к объекту и время его существования.

Правила действия областей видимости любого языка программирования — это правила, которые позволяют управлять доступом к объекту из различных частей программы. Другими словами, правила действия областей видимости определяют, какой код имеет доступ к той или иной переменной. Эти правила также определяют время "жизни" переменной. Как упоминалось выше, существует три вида переменных: локальные переменные, формальные параметры и глобальные переменные. На этот раз мы рассмотрим правила действия областей видимости с точки зрения функций.

Локальные переменные

Как вы уже знаете, переменные, объявленные внутри функции, называются локальными. Но в C++ предусмотрено более "внимательное" отношение к локальным переменным, чем мы могли заметить до сих пор. В C++ переменные могут быть включены в блоки. Это означает, что переменную можно объявить внутри любого блока кода, после чего она будет локальной по отношению к этому блоку. (Помните, что блок начинается с открывающей фигурной скобки и завершается закрывающей.) В действительности переменные, локальные по отношению к функции, образуют просто специальный случай более общей идеи.

Локальную переменную могут использовать лишь инструкции, включенные в блок, в котором эта переменная объявлена. Другими словами, локальная переменная неизвестна за пределами собственного блока кода. Следовательно, инструкции вне блока не могут получить доступ к объекту, определенному внутри блока.

Важно понимать, что локальные переменные существуют только во время выполнения программного блока, в котором они объявлены. Это означает, что локальная переменная создается при входе в "свой" блок и разрушается при выходе из него. А поскольку локальная переменная разрушается при выходе из "своего" блока, ее значение теряется.

Самым распространенным программным блоком является функция. В C++ каждая функция определяет блок кода, который начинается с открывающей фигурной скобки этой функции и завершается ее закрывающей фигурной скобкой. Код функции и ее данные — это ее "частная собственность", и к ней не может получить доступ ни одна инструкция из любой другой функции, за исключением инструкции ее вызова. (Например, невозможно использовать инструкцию *goto* для перехода в середину кода другой функции.) Тело функции надежно скрыто от остальной части программы, и если в функции не используются глобальные переменные, то она не может оказать никакого влияния на другие части программы, равно, как и те на нее. Таким образом, содержимое одной функции совершенно независимо от содержимого другой. Другими словами, код и данные, определенные в одной функции, не могут взаимодействовать с кодом и данными, определенными в другой, поскольку две функции имеют различные области видимости.

Поскольку каждая функция определяет собственную область видимости, переменные, объявленные в одной функции, не оказывают никакого влияния на переменные, объявленные в другой, причем даже в том случае, если эти переменные имеют одинаковые имена. Рассмотрим, например, следующую программу.

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

void f1();
int main()
{
  char str[] = "Это - массив str в функции main().";
  cout << str << '\n';
  f1();
  cout << str << '\n';

  return 0;
}

void f1()
{
  char str[80];
  cout << "Введите какую-нибудь строку: ";
    cin >> str;
  cout << str << '\n';
}
```

Символьный массив *str* объявляется здесь дважды: первый раз в функции *main()* и еще раз — в функции *f1()*. При этом массив *str*, объявленный в функции *main()*, не имеет никакого отношения к одноименному массиву из функции *f1()*. Как разъяснялось выше, каждый массив (в данном случае str) известен только блоку кода, в котором он объявлен. Чтобы убедиться в этом, достаточно выполнить приведенную выше программу. Как видите, несмотря на то, что массив *str* получает строку, вводимую пользователем при выполнении функции *f1()*, содержимое массива *str* в функции *main()* остается неизменным.

Язык C++ содержит ключевое слово *auto*, которое можно использовать для объявления локальных переменных. Но поскольку все неглобальные переменные являются по умолчанию *auto*-переменными, то к этому ключевому слову практически никогда не прибегают. Поэтому вы не найдете в этой книге ни одного примера с его использованием. Но если вы захотите все-таки применить его в своей программе, то знайте, что размещать его нужно непосредственно перед типом переменной, как показано ниже.

```
auto char ch;
```

Обычной практикой является объявление всех переменных, используемых в функции, в начале программного блока этой функции. В этом случае всякий, кому придется разбираться в коде этой функции, легко узнает, какие переменные в ней используются. Тем не менее начало блока функции — это не единственно возможное место для объявления локальных переменных. Локальные переменные можно объявлять в любом месте блока кода. Переменная, объявленная в блоке, локальна по отношению к этому блоку. Это означает, что такая переменная не существует до тех пор, пока не будет выполнен вход в блок, а разрушение такой переменной происходит при выходе из ее блока. При этом никакой код вне этого блока не может получить доступ к этой переменной (даже код, принадлежащий той же функции).

Ни б

чтоы лучше понять вышесказанное, рассмотрим следующую программу.

```
/* Эта программа демонстрирует локальность переменных по отношению к блоку.
*/
#include <iostream>
#include <cstring>
using namespace std;

int main( )
{
  int choice;
  cout << "(1) сложить числа или ";
  cout << "(2) конкатенировать строки?: ";
    cin >> choice;
  if(choice == 1) {
    int a, b; /* Активизируются две int-переменные. */
    cout << "Введите два числа: ";
      cin >> a >> b;
    cout << "Сумма равна " << a+b << '\n';
  }
  else {
    char s1 [80], s2[80]; /* Активизируются две строки. */
    cout << "Введите две строки: ";
      cin >> s1;
      cin >> s2;
    strcat(s1, s2);
    cout << "Конкатенация равна " << s1 << '\n';
  }
  return 0;
}
```

Эта программа в зависимости от выбора пользователя обеспечивает ввод либо двух чисел, либо двух строк. Обратите внимание на объявление переменных *a* и *b* в *if*-блоке и переменных *s1* и *s2* в *else*-блоке. Существование этих переменных начнется с момента входа в соответствующий блок и прекратится сразу после выхода из него. Если пользователь выберет сложение чисел, будут созданы переменные *a* и *b*, а если он захочет конкатенировать строки— переменные *s1* и *s2*. Наконец, ни к одной из этих переменных нельзя обратиться извне их блока, даже из части кода, принадлежащей той же функции. Например, если вы попытаетесь скомпилировать следующую (некорректную) версию программы, то получите сообщение об ошибке.

```
/* Эта программа некорректна. */
#include <iostream>
#include <cstring>
using namespace std;
```

```
int main( )
{
  int choice;
  cout << "(1) сложить числа или ";
  cout << "(2) конкатенировать строки?: ";
    cin >> choice;
  if(choice == 1) {
    int a, b; /* Активизируются две int-переменные. */
    cout << "Введите два числа: ";
      cin >> a >> b;
    cout << "Сумма равна " << a+b << '\n';
  }
  else {
    char s1 [80], s2 [80]; /* Активизируются две строки. */
    cout << "Введите две строки: ";
      cin >> s1;
      cin >> s2;
    strcat (s1, s2);
    cout << "Конкатенация равна " << s1 << '\n';
  }
  a = 10; // *** Ошибка ***
  // Переменная a здесь неизвестна!
  return 0;
}
```

Поскольку в данном случае переменная *a* неизвестна вне своего *if*-блока, компилятор выдаст ошибку при попытке ее использовать.

Если имя переменной, объявленной во внутреннем блоке, совпадает с именем переменной, объявленной во внешнем блоке, то "внутренняя" переменная переопределяет "внешнюю" в пределах области видимости внутреннего блока. Рассмотрим пример.

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

int main( )
{
  int i, j;
  i = 10;
  j = 100;
  if(j > 0) {
    int i; // Эта переменная i отделена от внешней переменной i.
    i = j /2;
    cout << "Внутренняя переменная i: " << i << '\n';
  }
  cout << "Внешняя переменная i: " << i << '\n';
```

```
  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Внутренняя переменная i: 50
Внешняя переменная i: 10
```

Здесь переменная *i*, объявленная внутри *if*-блока, переопределяет, или скрывает, внешнюю переменную *i*. Изменения, которым подверглась внутренняя переменная *i*, не оказывают никакого влияния на внешнюю *i*. Более того, вне *if*-блока внутренняя переменная *i* больше не существует, и поэтому внешняя переменная *i* снова становится видимой.

Поскольку локальные переменные создаются с каждым входом и разрушаются с каждым выходом из программного блока, в котором они объявлены, они не хранят своих значений между активизациями блоков. Это особенно важно помнить в отношении функций. При вызове функции ее локальные переменные создаются, а при выходе из нее — разрушаются. Это означает, что локальные переменные не сохраняют своих значений между вызовами функций. (Существует один способ обойти это ограничение — он будет рассмотрен ниже в этой книге.)

Локальные переменные не хранят своих значений между активизациями.

Локальные переменные хранятся в стеке, если не задан иной способ хранения. Поскольку стек — это динамически изменяемая область памяти, локальные переменные не могут в общем случае сохранять свои значения между вызовами функций.

Как упоминалось выше, несмотря на то, что локальные переменные обычно объявляются в начале своего блока, это не является обязательным. Локальные переменные можно объявить в любом месте блока, главное, чтобы это было сделано до их использования. Например, следующая программа вполне допустима.

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

int main( )
{
  cout << "Введите число: ";
  int a; // Объявляем одну переменную.
  cin >> a;

  cout << "Введите второе число: ";
  int b; // Объявляем еще одну переменную.
  cin >> b;

  cout << "Произведение равно: " << a*b << '\n';

  return 0;
}
```

В этом примере переменные *a* и *b* не объявляются до тех пор, пока они станут нужными. Все же большинство программистов объявляют все локальные переменные в начале блока, в котором они используются, но это, как говорится, вопрос стилистики (или вкуса).

Объявление переменных в итерационных инструкциях и инструкциях выбора

Переменную можно объявить в разделе инициализации цикла *for* или условном выражении инструкций *if*, *switch* или *while*. Переменная, объявленная в одной из этих инструкций, имеет

область видимости, которая ограничена блоком кода, управляемым этой инструкцией. Например, переменная, объявленная в инструкции цикла *for*, будет локальной для этого цикла, как показано в следующем примере.

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

int main( )
{
  // Переменная i локальная для цикла for.
  for(int i=0; i<10; i++) {
    cout << i << " ";
    cout << "в квадрате равно " << i * i << "\n";
  }
  // i = 10; // *** Ошибка *** -- i здесь неизвестна!
  return 0;
}
```

Здесь переменная *i* объявляется в разделе инициализации цикла *for* и используется для управления этим циклом. А за пределами цикла переменная *i* неизвестна.

В общем случае, если управляющая переменная цикла *for* не нужна за пределами этого цикла, то объявление ее внутри *for*-инструкции (как показано в этом примере) хорошо тем, что оно ограничивает ее существование рамками цикла и тем самым предотвращает случайное использование в каком-то другом месте программы. Профессиональные программисты часто объявляют управляющую переменную цикла внутри *for*-инструкции. Но если переменная требуется коду вне цикла, ее нельзя объявлять в инструкции *for*.

Важно! *Утверждение о том, что переменная, объявленная в разделе инициализации цикла for, является локальной по отношению к этому циклу или не является таковой, изменилось со временем (имеется в виду время, в течение которого развивался язык C++). Первоначально такая переменная была доступна после выхода из цикла for. Однако стандарт C++ ограничивает область видимости этой переменной рамками цикла for. Но следует иметь в виду, что различные компиляторы и теперь по-разному "смотрят" на эту ситуацию.*

Если ваш компилятор полностью соблюдает стандарт C++, то вы можете также объявить переменную в условном выражении инструкций *if*, *switch* или *while*. Например, в следующем фрагменте кода

```
if(int x = 20) {
  cout << "Это значение переменной x: ";
  cout << x;
}
```

объявляется переменная *x*, которой присваивается число *20*. Поскольку это выражение оценивается как истинное, инструкция *cout* будет выполнена. Область видимости переменных, объявленных в условном выражении инструкции, ограничивается блоком кода, управляемым этой инструкцией. Следовательно, в данном случае переменная *x* неизвестна за пределами инструкции *if*. По правде говоря, далеко не все программисты считают объявление переменных в условном выражении инструкций признаком хорошего стиля программирования, и поэтому такой прием в этой книге больше не повторится.

Формальные параметры

Как вы знаете, если функция использует аргументы, она должна объявить переменные, которые будут принимать значения этих аргументов. Эти переменные называются *формальными параметрами* функции. Если не считать получения значений аргументов при вызове функции, то поведение формальных параметров ничем не отличается от поведения любых других локальных переменных внутри функции. Область видимости параметра ограничивается рамками его функции.

Программист должен гарантировать, что тип объявляемых им формальных параметров совпадает с типом аргументов, передаваемых функции. И еще. Несмотря на то что эти переменные выполняют специальную задачу получения значений аргументов, их можно использовать подобно любым другим локальным переменным. Например, параметру внутри функции можно присвоить какое-нибудь новое значение.

Глобальные переменные

Глобальные переменные во многих отношениях противоположны локальным. Они известны на протяжении всей программы, их можно использовать в любом месте кода. и они сохраняют свои значения во время выполнения всего кода программы. Следовательно, их область видимости расширяется до объема всей программы. Глобальная переменная создается путем ее объявления вне какой бы то ни было функции. Благодаря их глобальности доступ к этим переменным можно получить из любого выражения, вне зависимости от функции, в которой это выражение находится.

Если глобальная и локальная переменные имеют одинаковые имена, то преимущество находится на стороне локальной переменной. Другими словами, локальная переменная скроет глобальную с таким же именем. Таким образом, несмотря на то, что к глобальной переменной теоретически можно получить доступ из любого кода программы, практически это возможно только в случае, если одноименная локальная переменная не переопределит глобальную.

Использование глобальных переменных демонстрируется в следующей программе. Как видите, переменные *count* и *num_right* объявлены вне всех функций, следовательно, они— глобальные. Из обычных практических соображений лучше объявлять глобальные переменные поближе к началу программы. Но формально они просто должны быть объявлены до их первого использования. Предлагаемая для рассмотрения программа— всего лишь простой тренажер по выполнению арифметического сложения. Сначала пользователю предлагается указать количество упражнений. Для выполнения каждого упражнения вызывается функция *drill()*, которая генерирует два случайных числа в диапазоне от *0* до *99*. Пользователю предлагается сложить эти числа, а затем проверяется ответ. На каждое упражнение дается три попытки. В конце программа отображает количество правильных ответов. Обратите особое внимание на глобальные переменные, используемые в этой программе.

```
// Простая программа-тренажер по выполнению сложения.
```

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

void drill();
int count; // Переменные count и num_right — глобальные.
int num_right;
int main()
{
  cout << "Сколько практических упражнений: ";
    cin >> count;
  num_right = 0;
  do {
    drill(); count--;
  }while(count);
  cout << "Вы дали " << num_right<< " правильных ответов.\n";
  return 0;
}

void drill()
{
  int count; /* Эта переменная count — локальная и никак не связана с
одноименной глобальной.*/
  int a, b, ans;
  // Генерируем два числа между 0 и 99.
  a = rand() % 100;
  b = rand() % 100;
  // Пользователь получает три попытки дать правильный ответ.
  for(count=0; count<3; count++) {
    cout << "Сколько будет " << a << " + " << b << "? ";
      cin >> ans;
    if(ans==a+b) {
      cout << "Правильно\n";
      num_right++;
      return;
    }
  }
  cout << "Вы использовали все свои попытки.\n";
  cout << "Ответ равен " << a+b << '\n';
}
```

При внимательном изучении этой программы вам должно быть ясно, что как функция *main()*, так и функция *drill()* получают доступ к глобальной переменной *num_right*. Но с переменной *count* дело обстоит несколько сложнее. В функции *main()* используется глобальная переменная *count*. Однако в функции *drill()* объявляется локальная переменная *count*

переменная *count*. Однако в функции *dim()* объявляется локальная переменная *count*. Поэтому здесь при использовании имени *count* подразумевается именно локальная, а не глобальная переменная *count*. Помните, что, если в функции глобальная и локальная переменные имеют одинаковые имена, то при обращении к этому имени подразумевается локальная, а не глобальная переменная.

Хранение глобальных переменных осуществляется в некоторой определенной области памяти, специально выделяемой программой для этих целей. Глобальные переменные полезны в том случае, когда в нескольких функциях программы используются одни и те же данные, или когда переменная должна хранить свое значение на протяжении выполнения всей программы. Однако без особой необходимости следует избегать использования глобальных переменных, и на это есть три причины.

■ Они занимают память в течение всего времени выполнения программы, а не только тогда, когда действительно необходимы.

■ Использование глобальной переменной в "роли", с которой легко бы "справилась" локальная переменная, делает такую функцию менее универсальной, поскольку она полагается на необходимость определения данных вне этой функции.

■ Использование большого количества глобальных переменных может привести к появлению ошибок в работе программы, поскольку при этом возможно проявление неизвестных и нежелательных побочных эффектов. Основная проблема, характерная для разработки больших C++-программ, — случайная модификация значения переменной в каком-то другом месте программы. Чем больше глобальных переменных в программе, тем больше вероятность ошибки.

Передача указателей и массивов в качестве аргументов

До сих пор в приводимых здесь примерах функциям передавались значения простых переменных. Но возможны ситуации, когда в качестве аргументов необходимо использовать указатели и массивы. Рассмотрению особенностей передачи аргументов этого типа и посвящены следующие подразделы.

Вызов функций с указателями

В C++ разрешается передавать функции указатели. Для этого достаточно объявить параметр типа указатель. Рассмотрим пример.

```
// Передача функции указателя.
#include <iostream>
using namespace std;

void f (int *j);
int main( )
{
  int i;
  int *p;
  p = &i; // Указатель p теперь содержит адрес переменной i.
  f(p);
  cout << i; // Переменная i теперь содержит число 100.
  return 0;
}

void f (int *j)
{
  *j = 100; // Переменной, адресуемой указателем j, присваивается число
```

```
100.
}
```

Как видите, в этой программе функция *f()* принимает один параметр: указатель на целочисленное значение. В функции *main()* указателю *p* присваивается адрес переменной *i*. Затем из функции *main()* вызывается функция *f()*, а указатель *p* передается ей в качестве аргумента. После того как параметр-указатель *j* получит значение аргумента *p*, он (так же, как и *p*) будет указывать на переменную *i*, определенную в функции *main()*. Таким образом, при выполнении операции присваивания

```
*j = 100;
```

переменная *i* получает значение *100*. Поэтому программа отобразит на экране число *100*. В общем случае приведенная здесь функция *f()* присваивает число *100* переменной, адрес которой был передан этой функции в качестве аргумента.

В предыдущем примере необязательно было использовать переменную *p*. Вместо нее при вызове функции *f()* достаточно использовать переменную *i*, предварив ее оператором *"&"* (при этом, как вы знаете, генерируется адрес переменной *i*). После внесения оговоренного изменения предыдущая программа приобретает такой вид.

```
// Передача указателя функции -- исправленная версия.
#include <iostream>
using namespace std;

void f (int *j);
int main( )
{
  int i;
  f(&i);
  cout << i;
  return 0;
}

void f (int * j)
{
  *j = 100; // Переменной, адресуемой указателем j, присваивается число
100.
}
```

Передавая указатель функции, необходимо понимать следующее. При выполнении некоторой операции в функции, которая использует указатель, эта операция фактически выполняется над переменной, адресуемой этим указателем. Таким образом, такая функция может изменить значение объекта, адресуемого ее параметром.

Вызов функций с массивами

Если массив является аргументом функции, то необходимо понимать, что при вызове такой функции ей передается только адрес первого элемента массива, а не полная его копия. (Помните, что в C++ имя массива без индекса представляет собой указатель на первый элемент этого массива.) Это означает, что объявление параметра должно иметь тип, совместимый с типом аргумента. Вообще существует три способа объявить параметр, который принимает указатель на массив. Во-первых, параметр можно объявить как массив, тип и размер которого совпадает с типом и размером массива, используемого при вызове функции. Этот вариант объявления параметра-массива продемонстрирован в следующем примере.

```
#include <iostream>
```

```
using namespace std;

void display(int num[10]);
int main( )
{
  int t[10], i;
  for(i=0; i<10; ++i) t[i]=i;
  display(t); // Передаем функции массив t.

  return 0;
}

// Функция выводит все элементы массива.
void display(int num[10])
{
  int i;
  for(i=0; i<10; i++) cout << num[i] <<' ';
}
```

Несмотря на то что параметр *num* объявлен здесь как целочисленный массив, состоящий из 10 элементов, C++-компилятор автоматически преобразует его в указатель на целочисленное значение. Необходимость этого преобразования объясняется тем, что никакой параметр в действительности не может принять массив целиком. А так как будет передан один лишь указатель на массив, то функция должна иметь параметр, способный принять этот указатель.

Второй способ объявления параметра-массива состоит в его представлении в виде безразмерного массива, как показано ниже.

```
void display(int num[])
{
  int i;
  for(i=0; i<10; i++) cout << num[i] << ' ';
}
```

Здесь параметр *num* объявляется как целочисленный массив неизвестного размера. Поскольку C++ не обеспечивает проверку нарушения границ массива, то реальный размер массива — нерелевантный фактор для подобного параметра (но, безусловно, не для программы в целом). Целочисленный массив при таком способе объявления также автоматически преобразуется C++-компилятором в указатель на целочисленное значение.

Наконец, рассмотрим третий способ объявления параметра-массива. При передаче массива функции ее параметр можно объявить как указатель. Как раз этот вариант чаще всего используется профессиональными программистами. Вот пример:

```
void display(int *num)
{
  int i;
  for(i=0; i<10; i++) cout << num[i] << ' ';
}
```

Возможность такого объявления параметра (в данном случае *num*) объясняется тем, что любой указатель (подобно массиву) можно индексировать с помощью символов квадратных

скобок (*[]*). Таким образом, все три способа объявления параметра-массива приводятся к одинаковому результату, который можно выразить одним словом: указатель.

Однако отдельный элемент массива, используемый в качестве аргумента, обрабатывается подобно обычной переменной. Например, рассмотренную выше программу можно было бы переписать, не используя передачу целого массива:

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

void display(int num);
int main( )
{
  int t[10],i;
  for(i=0; i<10; ++i) t[i]=i;
  for(i=0; i<10; i++) display(t[i]);

  return 0;
}

// Функция выводит одно число.
void display(int num)
{
  cout << num << ' ';
}
```

Как видите, параметр, используемый функцией *display()*, имеет тип *int*. Здесь не важно, что эта функция вызывается с использованием элемента массива, поскольку ей передается только один его элемент.

Помните, что, если массив используется в качестве аргумента функции, то функции передается адрес этого массива. Это означает, что код функции может потенциально изменить реальное содержимое массива, используемого при вызове функции. Например, в следующей программе функция *cube()* преобразует значение каждого элемента массива в куб этого значения. При вызове функции *cube()* в качестве первого аргумента необходимо передать адрес массива значений, подлежащих преобразованию, а в качестве второго — его размер.

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

void cube(int *n, int num);
int main( )
{
  int i, nums[10];
  for(i=0; i<10; i++) nums[i] = i+1;

  cout << "Исходное содержимое массива: ";
  for(i=0; i<10; i++) cout << nums[i] << ' ';
  cout << '\n';

  cube(nums, 10); // Вычисляем кубы значений.
  cout << "Измененное содержимое: ";
```

```
  for(i=0; i<10; i++) cout << nums[i] << ' ';
  return 0;
}

void cube(int *n, int num)
{
  while(num) {
    *n = *n * *n * *n;
    num--;
    n++;
  }
}
```

Результаты выполнения этой программы таковы.

```
Исходное содержимое массива: 12345678910
Измененное содержимое: 1 8 27 64 125 216 343 512 729 1000
```

Как видите, после обращения к функции *cube()* содержимое массива *nums* изменилось: каждый элемент стал равным кубу исходного значения. Другими словами, элементы массива *nums* были модифицированы инструкциями, составляющими тело функции *cube()*, поскольку ее параметр *n* указывает на массив *nums*.

Передача функциям строк

Как вы уже знаете, строки в C++ — это обычные символьные массивы, которые завершаются нулевым символом. Таким образом, при передаче функции строки реально передается только указатель (типа char*) на начало этой строки. Рассмотрим, например, следующую программу. В ней определяется функция *stringupper()*, которая преобразует строку символов в ее прописной эквивалент.

```
// Передача функции строки.
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

void stringupper(char *str);
int main()
{
  char str[80];
  strcpy(str, "Мне нравится C++");
  stringupper(str);
  cout << str; // Отображаем строку с использованием прописного написания символов.
  return 0;
}

void stringupper(char *str)
{
```

```
  while(*str) {
     *str = toupper(*str); // Получаем прописной эквивалент одного символа.
    str++; // Переходим к следующему символу.
  }
}
```

Результаты выполнения этой программы таковы.

```
МНЕ НРАВИТСЯ С++
```

Обратите внимание на то, что параметр *str* функции *stringupper()* объявляется с использованием типа *char**. Это позволяет получить указатель на символьный массив, который содержит строку. Рассмотрим еще один пример передачи строки функции. Как вы узнали в главе 5, стандартная библиотечная функция *strlen()* возвращает длину строки. В следующей программе показан один из возможных вариантов реализации этой функции.

```
// Одна из версий функции strlen().
#include <iostream>
using namespace std;

int mystrlen(char *str);
int main()
{
  cout << "Длина строки ПРИВЕТ ВСЕМ равна: ";
  cout << mystrlen("ПРИВЕТ ВСЕМ");
  return 0;
}

// Нестандартная реализация функции strlen().
int mystrlen(char *str)
{
  int i;
  for(i=0; str[i]; i++); // Находим конец строки.
  return i;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Длина строки ПРИВЕТ ВСЕМ равна: 11
```

В качестве упражнения вам стоило бы попытаться самостоятельно реализовать другие строковые функции, например *strcpy()* или *strcat()*. Этот тест позволит узнать, насколько хорошо вы освоили такие элементы языка С++, как массивы, строки и указатели.

Аргументы функции main(): argc и argv

Аргумент командной строки представляет собой информацию, задаваемую в командной строке после имени программы.

Иногда возникает необходимость передать информацию программе при ее запуске. Как правило, это реализуется путем передачи аргументов командной строки функции *main()*. Аргумент командной строки представляет собой информацию, указываемую в команде (командной строке), предназначенной для выполнения операционной системой, после имени программы. (В Windows команда "Run" (Выполнить) также использует командную строку.) Например, С++-программы можно компилировать путем выполнения следующей команды

команды,

```
cl prog_name
```

Здесь элемент *prog_name* — имя программы, которую мы хотим скомпилировать. Имя программы передается C++-компилятору в качестве аргумента командной строки.

В C++ для функции *main()* определено два встроенных, но необязательных параметра, *argc* и *argv*, которые получают свои значения от аргументов командной строки. В конкретной операционной среде могут поддерживаться и другие аргументы (такую информацию необходимо уточнить по документации, прилагаемой к вашему компилятору). Рассмотрим параметры *argc* и *argv* более подробно.

На заметку. *Формально для имен параметров командной строки можно выбирать любые идентификаторы, однако имена argc и argv используются по соглашению уже в течение нескольких лет, и поэтому имеет смысл не прибегать к другим идентификаторам, чтобы любой программист, которому придется разбираться в вашей программе, смог быстро идентифицировать их как параметры командной строки.*

Параметр *argc* имеет целочисленный тип и предназначен для хранения количества аргументов командной строки. Его значение всегда не меньше единицы, поскольку имя программы также является одним из учитываемых аргументов. Параметр *argv* представляет собой указатель на массив символьных указателей. Каждый указатель в массиве *argv* ссылается на строку, содержащую аргумент командной строки. Элемент *argv[0]* указывает на имя программы; элемент *argv[1]* — на первый аргумент, элемент *argv[2]* — на второй и т.д. Все аргументы командной строки передаются программе как строки, поэтому числовые аргументы необходимо преобразовать в программе в соответствующий внутренний формат.

Важно правильно объявить параметр *argv*. Обычно это делается так.

```
char *argv[];
```

Доступ к отдельным аргументам командной строки можно получить путем индексации массива *argv*. Как это сделать, показано в следующей программе. При ее выполнении на экран выводится приветствие ("Привет"), а за ним — ваше имя, которое должно быть первым аргументом командной строки.

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Вы забыли ввести свое имя.\n";
    return 1;
  }
  cout << "Привет, " << argv[1] << '\n';
  return 0;
}
```

Предположим, что вас зовут *Том* и что вы назвали эту программу именем name. Тогда, если запустить эту программу, введя команду *name Том*, результат ее работы должен выглядеть так: *Привет, Том*. Например, вы работаете с диском A, и в ответ на приглашение на ввод команды должны ввести упомянутую выше команду и получить следующий результат.

```
A>name Том
Привет, Том
A>
```

В C++ точно не оговорено, как должны быть представлены аргументы командной строки, поскольку среды выполнения (операционные системы) имеют здесь большие различия. Однако чаще всего используется следующее соглашение: каждый аргумент командной строки должен быть отделен пробелом или символом табуляции. Как правило, запятые, точки с запятой и тому подобные знаки не являются допустимыми разделителями аргументов. Например, строка

```
один, два и три
```

состоит из четырех строковых аргументов, в то время как строка

```
один, два, три
```

включает только два, поскольку запятая не является допустимым разделителем.

Если необходимо передать в качестве одного аргумента командной строки набор символов, который содержит пробелы, то его нужно заключить в кавычки. Например, этот набор символов будет воспринят как один аргумент командной строки:

```
"это лишь один аргумент"
```

Следует иметь в виду, что представленные здесь примеры применимы к широкому диапазону сред, но это не означает, что ваша среда входит в их число.

Чтобы получить доступ к отдельному символу в одном из аргументов командной строки, при обращении к массиву *argv* добавьте второй индекс. Например, при выполнении приведенной ниже программы посимвольно отображаются все аргументы, с которыми она была вызвана.

```
/* Эта программа посимвольно выводит все аргументы командной строки, с
которыми она была вызвана.
*/
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  int t, i;
  for(t=0; t<argc; ++t) {
    i = 0;
    while(argv[t][i]) {
      cout << argv[t][i];
      ++i;
    }
    cout << ' ';
  }
  return 0;
}
```

Нетрудно догадаться, что первый индекс массива *argv* позволяет получить доступ к соответствующему аргументу командной строки, а второй — к конкретному символу этого строкового аргумента.

Обычно аргументы *argc* и *argv* используются для ввода в программу начальных параметров, исходных значений, имен файлов или вариантов (режимов) работы программы. В C++ можно ввести столько аргументов командной строки, сколько допускает операционная система. Использование аргументов командной строки придает программе

система. Использование аргументов командной строки придает программе профессиональный вид и позволяет использовать ее в командном файле (исполняемом текстовом файле, содержащем одну или несколько команд).

Передача числовых аргументов командной строки

Как упоминалось выше, при передаче программе числовых данных в качестве аргументов командной строки эти данные принимаются в строковой форме. В программе должно быть предусмотрено их преобразование в соответствующий внутренний формат с помощью одной из стандартных библиотечных функций, поддерживаемых C++. Например, при выполнении следующей программы выводится сумма двух чисел, которые указываются в командной строке после имени программы. Для преобразования аргументов командной строки во внутреннее представление здесь используется стандартная библиотечная функция *atof()*. Она преобразует число из строкового формата в значение типа double.

```
/* Эта программа отображает сумму двух числовых аргументов командной строки.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  double a, b;
  if(argc!=3) {
    cout << "Использование: add число число\n";
    return 1;
  }
  a = atof(argv[1]);
  b = atof(argv[2]);
  cout << a + b;
  return 0;
}
```

Чтобы сложить два числа, используйте командную строку такого вида (предполагая, что эта программа имеет имя *add*).

```
C>add 100.2 231
```

Преобразование числовых строк в числа

Стандартная библиотека C++ включает несколько функций, которые позволяют преобразовать строковое представление числа в его внутренний формат. Для этого используются такие функции, как *atoi()*, *atol()* и *atof()*. Они преобразуют строку в целочисленное значение (типа *int*), длинное целое (типа *long*) и значение с плавающей точкой (типа *double*) соответственно. Использование этих функций (для их вызова необходимо включить в программу заголовочный файл *<cstdlib>*) демонстрируется в следующей программе.

```
// Демонстрация использования функций atoi(), atol() и atof().
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
int main( )
{
  int i;
  long j;
  double k;
  i = atoi ("100");
  j = atol("100000");
  k = atof("-0.123");
  cout << i << ' ' << j << ' ' << k;
  cout << ' \n';
  return 0;
}
```

Результаты выполнения этой программы таковы.

```
100 100000 -0.123
```

Функции преобразования строк полезны не только при передаче числовых данных программе через аргументы командной строки, но и в ряде других ситуаций.

Инструкция return

До сих пор (начиная с главы 2) мы использовали инструкцию *return* без подробных разъяснений. Напомним, что инструкция *return* выполняет две важные операции. Во-первых, она обеспечивает немедленное возвращение управления к инициатору вызова функции. Во-вторых, ее можно использовать для передачи значения, возвращаемого функцией. Именно этим двум операциям и посвящен данный раздел.

Завершение функции

Как вы уже знаете, управление от функции передается инициатору ее вызова в двух ситуациях: либо при обнаружении закрывающейся фигурной скобки, либо при выполнении инструкции *return*. Инструкцию *return* можно использовать с некоторым заданным значением либо без него. Но если в объявлении функции указан тип возвращаемого значения (т.е. не тип *void*), то функция должна возвращать значение этого типа. Только *void*-функции могут использовать инструкцию *return* без какого бы то ни было значения.

Для *void*-функций инструкция *return* главным образом используется как элемент программного управления. Например, в приведенной ниже функции выводится результат возведения числа в положительную целочисленную степень. Если же показатель степени окажется отрицательным, инструкция *return* обеспечит выход из функции, прежде чем будет сделана попытка вычислить такое выражение. В этом случае инструкция *return* действует как управляющий элемент, предотвращающий нежелательное выполнение определенной части функции.

```
void power(int base, int exp)
{
  int i;
   if(exp<0) return; /* Чтобы не допустить возведения числа в отрицательную степень, здесь выполняется возврат в вызывающую функцию и игнорируется остальная часть функции. */
  i = 1;
  for( ; exp; exp--) i = base * i;
  cout << "Результат равен: " << i;
}
```

}

Функция может содержать несколько инструкций *return*. Функция будет завершена при выполнении хотя бы одного из них. Например, следующий фрагмент кода совершенно правомерен.

```
void f()
{
  // ...
  switch(c) {
    case 'a': return;
    case 'b': // ...
    case 'c': return;
  }
  if(count<100) return;
  // ...
}
```

Однако следует иметь в виду, что слишком большое количество инструкций *return* может ухудшить ясность алгоритма и ввести в заблуждение тех, кто будет в нем разбираться. Несколько инструкций *return* стоит использовать только в том случае, если они способствуют ясности функции.

Возврат значений

Каждая функция, кроме типа *void*, возвращает какое-нибудь значение. Это значение явно задается с помощью инструкции *return*. Другими словами, любую не *void*-функцию можно использовать в качестве операнда в выражении. Следовательно, каждое из следующих выражений допустимо в C++.

```
x = power(y);
if(max(x, y)) > 100) cout << "больше";
switch(abs (x)) {
```

Несмотря на то что все не *void*-функции возвращают значения, они необязательно должны быть использованы в программе. Самый распространенный вопрос относительно значений, возвращаемых функциями, звучит так: "Поскольку функция возвращает некоторое значение, то разве я не должен (должна) присвоить это значение какой-нибудь переменной?". Ответ: нет, это необязательно. Если значение, возвращаемое функцией, не участвует в присваивании, оно попросту отбрасывается (теряется).

Рассмотрим следующую программу, в которой используется стандартная библиотечная функция *abs()*.

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

int main()
{
  int i;
  i = abs(-10); // строка 1
  cout << abs(-23); // строка 2
  abs(100); // строка 3
```

```
  return 0;
}
```

Функция *abs()* возвращает абсолютное значение своего целочисленного аргумента. Она использует заголовок *<cstdlib>*. В строке *1* значение, возвращаемое функцией *abs()*, присваивается переменной *i*. В строке *2* значение, возвращаемое функцией *abs()*, ничему не присваивается, но используется инструкцией *cout*. Наконец, в строке *3* значение, возвращаемое функцией *abs()*, теряется, поскольку не присваивается никакой другой переменной и не используется как часть выражения.

Если функция, тип которой отличен от типа *void*, завершается в результате обнаружения закрывающейся фигурной скобки, то значение, которое она возвращает, не определено (т.е. неизвестно). Из-за особенностей формального синтаксиса C++ не void-функция не обязана выполнять инструкцию *return*. Это может произойти в том случае, если конец функции будет достигнут до обнаружения инструкции *return*. Но, поскольку функция объявлена как возвращающая значение, значение будет таки возвращено, даже если это просто "мусор". В общем случае любая создаваемая вами не *void*-функция должна возвращать значение посредством явно выполняемой инструкции *return*.

Выше упоминалось, что *void*-функция может иметь несколько инструкций *return*. То же самое относится и к функциям, которые возвращают значения. Например, представленная в следующей программе функция *find_substr()* использует две инструкции *return*, которые позволяют упростить алгоритм ее работы. Эта функция выполняет поиск заданной подстроки в заданной строке. Она возвращает индекс первого обнаруженного вхождения заданной подстроки или значение *-1*, если заданная подстрока не была найдена. Например, если в строке *"Я люблю C++ "* необходимо отыскать подстроку *"люблю"*, то функция *find_substr()* возвратит число *2* (которое представляет собой индекс символа *"л"* в строке *"Я люблю C++ "*).

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

int find_substr(char *sub, char *str);
int main( )
{
  int index;
  index = find_substr("три", "один два три четыре");
  cout << "Индекс равен " << index; // Индекс равен 9.

  return 0;
}

// Функция возвращает индекс искомой подстроки или -1, если она не была найдена.
int find_substr(char *sub, char *str)
{
  int t;
  char *p, *p2;
  for(t=0; str[t]; t++) {
    p = &str[t]; // установка указателей
    p2 = sub;
    while(*p2 && *p2==*p) { // проверка совпадения
      p++; p2++;
```

```
    }
    /* Если достигнут конец p2-строки (т.е. подстроки), то подстрока была
найдена. */
    if(!*p2) return t; // Возвращаем индекс подстроки.
  }
  return -1; // Подстрока не была обнаружена.
}
```

Результаты выполнения этой программы таковы.

```
Индекс равен 9
```

Поскольку искомая подстрока существует в заданной строке, выполняется первая инструкция *return*. В качестве упражнения измените программу так, чтобы ею выполнялся поиск подстроки, которая не является частью заданной строки. В этом случае функция *find_substr()* должна возвратить значение *-1* (благодаря второй инструкции *return*).

Функцию можно объявить так, чтобы она возвращала значение любого типа данных, действительного для C++ (за исключением массива: функция не может возвратить массив). Способ объявления типа значения, возвращаемого функцией, аналогичен тому, который используется для объявления переменных: имени функции должен предшествовать спецификатор типа. Спецификатор типа сообщает компилятору, значение какого типа данных должна возвратить функция. Указываемый в объявлении функции тип должен быть совместимым с типом данных, используемым в инструкции return. В противном случае компилятор отреагирует сообщением об ошибке.

Функции, которые не возвращают значений (void-функции)

Как вы заметили, функции, которые не возвращают значений, объявляются с указанием типа *void*. Это ключевое слово не допускает их использования в выражениях и защищает от неверного применения. В следующем примере функция *print_vertical()* выводит аргумент командной строки в вертикальном направлении (вниз) по левому краю экрана. Поскольку эта функция не возвращает никакого значения, в ее объявлении использовано ключевое слово void.

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

void print_vertical(char *str);
int main(int argc, char *argv[])
{
  if(argc==2) print_vertical(argv[1]);
  return 0;
}

void print_vertical(char *str)
{
  while(*str)
  cout << *str++ << '\n';
}
```

Поскольку *print_vertical()* объявлена как *void*-функция, ее нельзя использовать в выражении. Например, следующая инструкция неверна и поэтому не скомпилируется.

```
x = print_vertical("Привет!"); // ошибка
```

Важно! *В первых версиях языка С не был предусмотрен тип void. Таким образом, в старых С-программах функции, не возвращающие значений, по умолчанию имели тип int. Если вам придется встретиться с такими функциями при переводе старых С-программ "на рельсы" С++, просто объявите их с использованием ключевого слова void, сделав их void-функциями.*

Функции, которые возвращают указатели

Функции могут возвращать указатели. Указатели возвращаются подобно значениям любых других типов данных и не создают при этом особых проблем. Но, поскольку указатель представляет собой одно из самых сложных (или небезопасных) средств языка С++, имеет смысл посвятить ему отдельный раздел.

Чтобы вернуть указатель, функция должна объявить его тип в качестве типа возвращаемого значения. Вот как, например, объявляется тип возвращаемого значения для функции *f()*, которая должна возвращать указатель на целое число.

```
int *f();
```

Если функция возвращает указатель, то значение, используемое в ее инструкции *return*, также должно быть указателем. (Как и для всех функций, *return*-значение должно быть совместимым с типом возвращаемого значения.)

В следующей программе демонстрируется использование указателя в качестве типа возвращаемого значения. Это — новая версия приведенной выше функции *find_substr()*, только теперь она возвращает не индекс найденной подстроки, а указатель на нее. Если заданная подстрока не найдена, возвращается нулевой указатель.

```
// Новая версия функции find_substr().
// которая возвращает указатель на подстроку.
#include <iostream>
using namespace std;

char *find_substr(char *sub, char *str);
int main()
{
  char *substr;
  substr = find_substr("три", "один два три четыре");
  cout << "Найденная подстрока: " << substr;

  return 0;
}

// Функция возвращает указатель на искомую подстроку или нуль, если таковая не будет найдена.
char *find_substr(char *sub, char *str)
{
  int t;
  char *p, *p2, *start;
  for(t=0; str[t]; t++) {
    p = &str[t]; // установка указателей
    start = p;
    p2 = sub;
    while(*p2 && *p2==*p) { // проверка совпадения
```

```
    p++; p2++;
   }
   /* Если достигнут конец p2-подстроки, то эта подстрока была найдена.
*/
    if(!*p2) return start; // Возвращаем указатель на начало найденной
подстроки.
  }
  return 0; // подстрока не найдена
}
```

При выполнении этой версии программы получен следующий результат.

```
Найденная подстрока: три четыре
```

В данном случае, когда подстрока *"три"* была найдена в строке *"один два три четыре"*, функция *find_substr()* возвратила указатель на начало искомой подстроки *"три"*, который в функции main() был присвоен переменной *substr*. Таким образом, при выводе значения *substr* на экране отобразился остаток строки, т.е. *"три четыре"*.

Многие поддерживаемые C++ библиотечные функции, предназначенные для обработки строк, возвращают указатели на символы. Например, функция *strcpy()* возвращает указатель на первый аргумент.

Прототипы функций

Прототип объявляет функцию до ее первого использования.

До сих пор в приводимых здесь примерах программ прототипы функций использовались без каких-либо разъяснений. Теперь настало время поговорить о них подробно. В C++ все функции должны быть объявлены до их использования. Обычно это реализуется с помощью прототипа функции. Прототипы содержат три вида информации о функции:

■ тип возвращаемого ею значения;

■ тип ее параметров;

■ количество параметров.

Прототипы позволяют компилятору выполнить следующие три важные операции.

■ Они сообщают компилятору, код какого типа необходимо генерировать при вызове функции. Различия в типах параметров и значении, возвращаемом функцией, обеспечивают различную обработку компилятором.

■ Они позволяют C++ обнаружить недопустимые преобразования типов аргументов, используемых при вызове функции, в тип, указанный в объявлении ее параметров, и сообщить о них.

■ Они позволяют компилятору выявить различия между количеством аргументов, используемых при вызове функции, и количеством параметров, заданных в определении функции.

Общая форма прототипа функции аналогична ее определению за исключением того, что в прототипе не представлено тело функции.

```
type func_name(type parm_name1, type parm_name2,
...,
type parm_nameN);
```

Использование имен параметров в прототипе необязательно, но позволяет компилятору идентифицировать любое несовпадение типов при возникновении ошибки, поэтому лучше имена параметров все же включать в прототип функции.

Чтобы лучше понять полезность прототипов функций, рассмотрим следующую программу. Если вы попытаетесь ее скомпилировать, то получите от компилятора сообщение об

ошибке, поскольку в этой программе делается попытка вызвать функцию *sqr_it()* с целочисленным аргументом, а не с указателем на целочисленное значение (согласно прототипу функции). Ошибка состоит в недопустимости преобразования целочисленного значения в указатель.

```
/* В этой программе используется прототип функции, который позволяет
осуществить строгий контроль типов.
*/
void sqr_it(int *i); // прототип функции
int main( )
{
  int x;
  x = 10;
  sqr_it(x); // *** Ошибка *** – несоответствие типов!
  return 0;
}

void sqr_it(int *i)
{
  *i=*i * *i;
}
```

Важно! *Несмотря на то что язык С допускает прототипы, их использование не является обязательным. Дело в том, что в первых версиях С они не применялись. Поэтому при переводе старого С-кода в С++-код перед компиляцией программы необходимо обеспечить наличие прототипов абсолютно для всех функций.*

Подробнее о заголовках

В начале этой книги вы узнали о существовании стандартных заголовков С++, которые содержат информацию, необходимую для ваших программ. Несмотря на то что все вышесказанное — истинная правда, это еще не вся правда. Заголовки С++ содержат прототипы стандартных библиотечных функций, а также различные значения и определения, используемые этими функциями. Подобно функциям, создаваемым программистами, стандартные библиотечные функции также должны "заявить о себе" в форме прототипов до их использования. Поэтому любая программа, в которой используется библиотечная функция, должна включать заголовок, содержащий прототип этой функции.

Чтобы узнать, какой заголовок необходим для той или иной библиотечной функции, следует обратиться к справочному руководству, прилагаемому к вашему компилятору. Помимо описания каждой функции, там должно быть указано имя заголовка, который необходимо включить в программу для использования выбранной функции.

Сравнение старого и нового стилей объявления параметров функций

Если вам приходилось когда-либо разбираться в старом С-коде, то вы, возможно, обратили внимание на необычное (с точки зрения современного программиста) объявление параметров функции. Этот старый стиль объявления параметров, который иногда называют *классическим форматом*, устарел, но его до сих пор можно встретить в программах "эпохи раннего С". В С++ (и обновленном С-коде) используется новая форма объявлений параметров функций. Но если вам придется работать со старыми С-программами и, особенно, если понадобится переводить их в С++-код, то вам будет полезно понимать и форму объявления параметров, "выдержанную" в старом стиле.

Объявление параметра функции согласно старому стилю состоит из двух частей: списка параметров, заключенного в круглые скобки, который приводится после имени функции, и

собственно объявления параметров, которое должно находиться между закрывающей круглой скобкой и открывающей фигурной скобкой функции. Например, это "новое" объявление (т.е. по новому стилю)

```
float f(int a, int b, char ch)
{ ...
```

будет выглядеть с использованием старого стиля несколько по-другому.

```
float f(a, b, ch)
int a, b;
char ch;
{ ...
```

Обратите внимание на то, что в классической форме после указания имени типа в списке может находиться несколько параметров. В новой форме объявления это не допускается.

В общем случае, чтобы преобразовать объявление параметров из старого стиля в новый (C++-стиль), достаточно внести объявление типов параметров в круглые скобки, следующие за именем функции. При этом помните, что каждый параметр должен быть объявлен отдельно, с собственным спецификатором типа.

Рекурсия

Рекурсивная функция — *это функция, которая вызывает сама себя*.

Рекурсия — это последняя тема, которую мы рассмотрим в этой главе. Рекурсия, которую иногда называют *циклическим определением*, представляет собой процесс определения чего-либо на собственной основе. В области программирования под рекурсией понимается процесс вызова функцией самой себя. Функцию, которая вызывает саму себя, называют *рекурсивной*.

Классическим примером рекурсии является вычисление факториала числа с помощью функции *factr()*. Факториал числа *N* представляет собой произведение всех целых чисел от *1* до *N*. Например, факториал числа *3* равен *1x2x3*, или *6*. Рекурсивный способ вычисления факториала числа демонстрируется в следующей программе. Для сравнения сюда же включен и его нерекурсивный (итеративный) эквивалент.

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

int factr(int n);
int fact(int n);
int main( )
{
  // Использование рекурсивной версии.
  cout << "Факториал числа 4 равен " << factr(4);
  cout << '\n';

  // Использование итеративной версии.
  cout << "Факториал числа 4 равен " << fact(4);
  cout << '\n';

  return 0;
}

// Рекурсивная версия.
```

```
int factr(int n)
{
  int answer;
  if(n==1) return(1);
  answer = factr(n-1)*n;
  return(answer);
}

// Итеративная версия.
int fact(int n)
{
  int t, answer;
  answer =1;
  for(t=1; t<=n; t++) answer = answer* (t);
  return (answer);
}
```

Нерекурсивная версия функции *fact()* довольно проста и не требует расширенных пояснений. В ней используется цикл, в котором организовано перемножение последовательных чисел, начиная с *1* и заканчивая числом, заданным в качестве параметра: на каждой итерации цикла текущее значение управляющей переменной цикла умножается на текущее значение произведения, полученное в результате выполнения предыдущей итерации цикла.

Рекурсивная функция *factr()* несколько сложнее. Если она вызывается с аргументом, равным *1*, то сразу возвращает значение *1*. В противном случае она возвращает произведение *factr(n-1) * n*. Для вычисления этого выражения вызывается метод *factr()* с аргументом *n-1*. Этот процесс повторяется до тех пор, пока аргумент не станет равным *1*, после чего вызванные ранее методы начнут возвращать значения. Например, при вычислении факториала числа *2* первое обращение к методу *factr()* приведет ко второму обращению к тому же методу, но с аргументом, равным *1*. Второй вызов метода *factr()* возвратит значение *1*, которое будет умножено на *2* (исходное значение параметра *n*). Возможно, вам будет интересно вставить в функцию *factr()* инструкции *cout*, чтобы показать уровень каждого вызова и промежуточные результаты.

Когда функция вызывает сама себя, в системном стеке выделяется память для новых локальных переменных и параметров, и код функции с самого начала выполняется с этими новыми переменными. Рекурсивный вызов не создает новой копии функции. Новыми являются только аргументы. При возвращении каждого рекурсивного вызова из стека извлекаются старые локальные переменные и параметры, и выполнение функции возобновляется с "внутренней" точки ее вызова. О рекурсивных функциях можно сказать, что они "выдвигаются" и "задвигаются".

Следует иметь в виду, что в большинстве случаев использование рекурсивных функций не дает значительного сокращения объема кода. Кроме того, рекурсивные версии многих процедур выполняются медленнее, чем их итеративные эквиваленты, из-за дополнительных затрат системных ресурсов, связанных с многократными вызовами функций. Слишком большое количество рекурсивных обращений к функции может вызвать переполнение стека. Поскольку локальные переменные и параметры сохраняются в системном стеке и каждый новый вызов создает новую копию этих переменных, может настать момент, когда память стека будет исчерпана. В этом случае могут быть разрушены другие ("ни в чем не повинные") данные. Но если рекурсия построена корректно, об этом вряд ли стоит волноваться.

Основное достоинство рекурсии состоит в том, что некоторые типы алгоритмов рекурсивно реализуются проще, чем их итеративные эквиваленты. Например, алгоритм сортировки *Quicksort* довольно трудно реализовать итеративным способом. Кроме того, некоторые задачи (особенно те, которые связаны с искусственным интеллектом) просто созданы для рекурсивных решений. Наконец, у некоторых программистов процесс мышления организован так, что им проще думать рекурсивно, чем итеративно.

При написании рекурсивной функции необходимо включить в нее инструкцию проверки условия (например, if-инструкцию), которая бы обеспечивала выход из функции без выполнения рекурсивного вызова. Если этого не сделать, то, вызвав однажды такую функцию, из нее уже нельзя будет вернуться. При работе с рекурсией это самый распространенный тип ошибки. Поэтому при разработке программ с рекурсивными функциями не стоит скупиться на инструкции *cout*, чтобы быть в курсе того, что происходит в конкретной функции, и иметь возможность прервать ее работу в случае обнаружения ошибки.

Рассмотрим еще один пример рекурсивной функции. Функция *reverse()* использует рекурсию для отображения своего строкового аргумента в обратном порядке.

```
// Отображение строки в обратном порядке с помощью рекурсии.
#include <iostream>
using namespace std;

void reverse(char *s);
int main( )
{
  char str[] = "Это тест";
  reverse(str);
  return 0;
}

// Вывод строки в обратном порядке.
void reverse(char *s)
{
  if(*s)reverse(s+1);
  else return;
  cout << *s;
}
```

Функция *reverse()* проверяет, не передан ли ей в качестве параметра указатель на нуль, которым завершается строка. Если нет, то функция *reverse()* вызывает саму себя с указателем на следующий символ в строке. Этот "закручивающийся" процесс повторяется до тех пор, пока той же функции не будет передан указатель на нуль. Когда, наконец, обнаружится символ конца строки, пойдет процесс "раскручивания", т.е. вызванные ранее функции начнут возвращать значения, и каждый возврат будет сопровождаться "довыполнением" метода, т.е. отображением символа *s*. В результате исходная строка посимвольно отобразится в обратном порядке.

Создание рекурсивных функций часто вызывает трудности у начинающих программистов. Но с приходом опыта использование рекурсии становится для многих обычной практикой.

Глава 8: Функции, часть вторая: ссылки, перегрузка и использование аргументов по умолчанию

В этой главе мы продолжим изучение функций, а именно рассмотрим три самые важные темы, связанные с функциями C++: *ссылки*, *перегрузка функций* и *использование аргументов*

по умолчанию. Эти три средства в значительной степени расширяют возможности функций. Как будет показано ниже, ссылка — это неявный указатель. Перегрузка функций представляет собой свойство, которое позволяет одну функцию реализовать несколькими способами, причем в каждом случае возможно выполнение отдельной задачи. Поэтому есть все основания считать перегрузку функций одним из путей поддержки полиморфизма в C++. Используя возможность задания аргументов по умолчанию, можно определить значение для параметра, которое будет автоматически применено в случае, если соответствующий аргумент не задан.

Поскольку к параметрам функций часто применяются ссылки (это основная причина их существования), начнем эту главу с краткого рассмотрения способов передачи аргументов функциям.

Два способа передачи аргументов

При вызове по значению функции передается значение аргумента.

Чтобы понять происхождение ссылки, необходимо знать теорию процесса передачи аргументов. В общем случае в языках программирования, как правило, предусматривается два способа, которые позволяют передавать аргументы в подпрограммы (функции, методы, процедуры). Первый называется *вызовом по значению* (call-by-value). В этом случае *значение* аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, внесенные в параметры подпрограммы, не влияют на аргументы, используемые при ее вызове.

При вызове по ссылке функции передается адрес аргумента.

Второй способ передачи аргумента подпрограмме называется *вызовом по ссылке* (call-by-reference). В этом случае в параметр копируется *адрес* аргумента (а не его значение). В пределах вызываемой подпрограммы этот адрес используется для доступа к реальному аргументу, заданному при ее вызове. Это значит, что изменения, внесенные в параметр, окажут воздействие на аргумент, используемый при вызове подпрограммы.

Как в C++ реализована передача аргументов

По умолчанию для передачи аргументов в C++ используется метод вызова по значению. Это означает, что в общем случае код функции не может изменить аргументы, используемые при вызове функции. Во всех программах этой книги, представленных до сих пор, использовался метод вызова по значению.

Рассмотрим следующую функцию.

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

int sqr_it(int x);
int main( )
{
  int t=10;
  cout << sqr_it(t) << ' ' << t;
  return 0;
}

int sqr_it(int x)
{
  x = x*x;
  return x;
}
```

```
}
```

В этом примере значение аргумента, передаваемого функции *sqr_it()*, *10*, копируется в параметр *x*. При выполнении присваивания *x = x*x* изменяется лишь локальная переменная *x*. Переменная *t*, используемая при вызове функции *sqr_it()*, по-прежнему будет иметь значение *10*, и на нее никак не повлияют операции, выполняемые в этой функции. Следовательно, после запуска рассматриваемой программы на экране будет выведен такой результат: *100 10*.

Узелок на память. *По умолчанию функции передается копия аргумента. То, что происходит внутри функции, никак не отражается на значении переменной, используемой при вызове функции.*

Использование указателя для обеспечения вызова по ссылке

Несмотря на то что в качестве C++-соглашения о передаче параметров по умолчанию действует вызов по значению, существует возможность "вручную" заменить его вызовом по ссылке. В этом случае функции будет передаваться адрес аргумента (т.е. указатель на аргумент). Это позволит внутреннему коду функции изменить значение аргумента, которое хранится вне функции. Пример такого "дистанционного" управления значениями переменных вы видели в предыдущей главе при рассмотрении возможности вызова функции с указателями (в примере программы функции передавался указатель на целочисленную переменную). Как вы знаете, указатели передаются функциям подобно значениям любого другого типа. Безусловно, для этого необходимо объявить параметры с типом указателей.

Чтобы понять, как передача указателя позволяет вручную обеспечить вызов по ссылке, рассмотрим следующую версию функции *swap()*. (Она меняет значения двух переменных, на которые указывают ее аргументы.)

```
void swap(int *x, int *y)
{
  int temp;
  temp = *x; // Временно сохраняем значение, расположенное по адресу x.
  *x = *y; // Помещаем значение, хранимое по адресу y, в адрес x.
  *y = temp; // Помещаем значение, которое раньше хранилось по адресу x, в адрес y.
}
```

Здесь параметры **x* и **y* означают переменные, адресуемые указателями *x* и *y*, которые попросту являются адресами аргументов, используемых при вызове функции *swap()*. Следовательно, при выполнении этой функции будет совершен реальный обмен содержимым переменных, используемых при ее вызове.

Поскольку функция *swap()* ожидает получить два указателя, вы должны помнить, что функцию *swap()* необходимо вызывать с адресами переменных, значения которых вы хотите обменять. Корректный вызов этой функции продемонстрирован в следующей программе.

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

// Объявляем функцию swap(), которая использует указатели.
void swap(int *x, int *y);
int main()
{
  int i, j;
  i = 10;
```

```
  j = 20;
  cout << "Исходные значения переменных i и j: ";
  cout << i << ' ' << j << '\n';

  swap(&j, &i); // Вызываем swap( ) с адресами переменных i и j.
  cout << "Значения переменных i и j после обмена: ";
  cout << i << ' ' << j << '\n';
  return 0;
}

// Обмен аргументами.
void swap(int *x, int *y)
{
  int temp;
  temp = *x; // Временно сохраняем значение, расположенное по адресу x.
  *x = *y; // Помещаем значение, хранимое по адресу y, в адрес x.
  *y = temp; // Помещаем значение, которое раньше хранилось по адресу x,
в адрес y.
}
```

Результаты выполнения этой программы таковы.

```
Исходные значения переменных i и j: 10 20
Значения переменных i и j после обмена: 20 10
```

В этом примере переменной *i* было присвоено начальное значение *10*, а переменной *j — 20*. Затем была вызвана функция *swap()* с адресами переменных *i* и *j*. Для получения адресов здесь используется унарный оператор Следовательно, функции *swap()* при вызове были переданы адреса переменных *i* и *j*, а не их значения. После выполнения функции *swap()* переменные *i* и *j* обменялись своими значениями.

Ссылочные параметры

Ссылочный параметр автоматически получает адрес соответствующего аргумента.

Несмотря на возможность "вручную" организовать вызов по ссылке с помощью оператора получения адреса, такой подход не всегда удобен. Во-первых, он вынуждает программиста выполнять все операции с использованием указателей. Во-вторых, вызывая функцию, программист должен не забыть передать ей адреса аргументов, а не их значения. К счастью, в C++ можно сориентировать компилятор на автоматическое использование вызова по ссылке (вместо вызова по значению) для одного или нескольких параметров конкретной функции. Такая возможность реализуется с помощью *ссылочного параметра* (reference parameter). При использовании ссылочного параметра функции автоматически передается адрес (а не значение) аргумента. При выполнении кода функции, а именно при выполнении операций над ссылочным параметром, обеспечивается его автоматическое разыменование, и поэтому программисту не нужно использовать операторы, работающие с указателями.

Ссылочный параметр объявляется с помощью символа который должен предшествовать имени параметра в объявлении функции. Операции, выполняемые над ссылочным параметром, оказывают влияние на аргумент, используемый при вызове функции, а не на сам ссылочный параметр.

Чтобы лучше понять механизм действия ссылочных параметров, рассмотрим для начала простой пример. В следующей программе функция *f()* принимает один ссылочный параметр типа *int*

типа *int*.

```
// Использование ссылочного параметра.
#include <iostream>
using namespace std;

void f(int &i);
int main( )
{
  int val = 1;
  cout << "Старое значение переменной val: " << val << '\n';

  f(val); // Передаем адрес переменной val функции f( ).
  cout << "Новое значение переменной val: " << val << '\n';

  return 0;
}

void f(int &i)
{
  i = 10; // Модификация аргумента, заданного при вызове.
}
```

При выполнении этой программы получаем такой результат.

```
Старое значение переменной val: 1
Новое значение переменной val: 10
```

Обратите особое внимание на определение функции *f()*.

```
void f (int &i)
{
  i = 10; // Модификация аргумента, заданного при вызове.
}
```

Итак, рассмотрим объявление параметра *i*. Его имени предшествует символ который "превращает" переменную *i* в ссылочный параметр. (Это объявление также используется в прототипе функции.) Инструкция

```
i = 10;
```

(в данном случае она одна составляет тело функции) *не* присваивает переменной *i* значение *10*. В действительности значение *10* присваивается переменной, на которую ссылается переменная *i* (в нашей программе ею является переменная val). Обратите внимание на то, что в этой инструкции не используется оператор который необходим при работе с указателями. Применяя ссылочный параметр, вы тем самым уведомляете C++-компилятор о передаче адреса (т.е. указателя), и компилятор автоматически разыменовывает его за вас. Более того, если бы вы попытались "помочь" компилятору, использовав оператор то сразу же получили бы сообщение об ошибке (и вправду "ни одно доброе дело не остается безнаказанным").

Поскольку переменная *i* была объявлена как ссылочный параметр, компилятор автоматически передает функции *f()* адрес любого аргумента, с которым вызывается эта функция. Таким образом, в функции *main()* инструкция

```
f(val); // Передаем адрес переменной val функции f( ).
```

передает функции *f()* адрес переменной *val* (а не ее значение). Обратите внимание на то, что при вызове функции *f()* не нужно предварять переменную *val* оператором *"&"*. (Более того,

это было бы ошибкой.) Поскольку функция *f()* получает адрес переменной *val* в форме ссылки, она может модифицировать значение этой переменной.

Чтобы проиллюстрировать реальное применение ссылочных параметров (и тем самым продемонстрировать их достоинства), перепишем нашу старую знакомую функцию *swap()* с использованием ссылок. В следующей программе обратите внимание на то, как функция *swap()* объявляется и вызывается.

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

// Объявляем функцию swap( ) с использованием ссылочных параметров.
void swap(int &x, int &y);
int main( )
{
  int i, j;
  i = 10; j = 20;

  cout << " Исходные значения переменных i и j: ";
  cout << i << ' ' << j << '\n';

  swap (j, i);
  cout << " Значения переменных i и j после обмена: ";
  cout << i << ' ' << j << '\n';

  return 0;
}

/* Здесь функция swap( ) определяется в расчете на вызов по ссылке, а не на вызов по значению. Поэтому она может выполнить обмен значениями двух аргументов, с которыми она вызывается.
*/
void swap(int &x, int &y)
{
  int temp;
  temp = x; // Сохраняем значение, расположенное по адресу x.
  x = y; // Помещаем значение, хранимое по адресу y, в адрес x.
  y = temp; // Помещаем значение, которое раньше хранилось по адресу x, в адрес y.
}
```

Опять таки, обратите внимание на то, что объявление *x* и *y* ссылочными параметрами избавляет вас от необходимости использовать оператор при организации обмена значениями. Как уже упоминалось, такая "навязчивость" с вашей стороны стала бы причиной ошибки. Поэтому запомните, что компилятор автоматически генерирует адреса аргументов, используемых при вызове функции *swap()*, и автоматически разыменовывает ссылки *x* и *y*.

Итак, подведем некоторые итоги. После создания ссылочный параметр автоматически ссылается (т.е. неявно указывает) на аргумент, используемый при вызове функции. Более того, при вызове функции не нужно применять к аргументу оператор Кроме того, в теле функции ссылочный параметр используется непосредственно, т.е. без использования оператора Все операции, включающие ссылочный параметр, автоматически выполняются над аргументом, используемым при вызове функции.

Узелок на память. *Присваивая некоторое значение ссылке, вы в действительности присваиваете это значение переменной, на которую указывает эта ссылка. Поэтому, применяя ссылку в качестве аргумента функции, при вызове функции вы в действительности используете такую переменную.*

Объявление ссылочных параметров

В изданной в 1986 г. книге *Язык программирования C++* (в которой был впервые описан синтаксис C++) Бьерн Страуструп представил стиль объявления ссылочных параметров, одобренный другими программистами. В соответствии с этим стилем оператор *"&"* связывается с именем типа, а не с именем переменной. Например, вот как выглядит еще один способ записи прототипа функции *swap()*.

```
void swap(int& x, int& y);
```

Нетрудно заметить, что в этом объявлении символ *"&"* прилегает вплотную к имени типа *int*, а не к имени переменной *x*.

Некоторые программисты определяют в таком стиле и указатели, связывая символ *"*"* с типом, а не с переменной, как в этом примере.

```
float* p;
```

Приведенные объявления отражают желание некоторых программистов иметь в C++ отдельный тип ссылки или указателя. Но дело в том, что подобное связывание символа *"&"* или *"*"* с типом (а не с переменной) не распространяется, в соответствии с формальным синтаксисом C++, на весь список переменных, приведенных в объявлении, что может привести к путанице. Например, в следующем объявлении создается один указатель (а не два) на целочисленную переменную.

```
int* a, b;
```

Здесь *b* объявляется как целочисленная переменная (а не как указатель на целочисленную переменную), поскольку, как определено синтаксисом C++, используемый в объявлении символ *"*"* или *"&"* связывается с конкретной переменной, которой он предшествует, а не с типом, за которым он следует.

Важно понимать, что для C++-компилятора абсолютно безразлично, как именно вы напишете объявление: *int *p* или *int* p*. Таким образом, если вы предпочитаете связывать символ *"*"* или *"&"* с типом, а не переменной, поступайте, как вам удобно. Но, чтобы избежать в дальнейшем каких-либо недоразумений, в этой книге мы будем связывать символ *"*"* или *"&"* с именем переменной, а не с именем типа.

Важно! *В языке C ссылки не поддерживаются. Поэтому единственный способ обеспечить в языке C вызов по ссылке состоит в использовании указателей, как было показано выше (см. первую версию функции swap()). Преобразуя C-код в C++-код, вам стоит вместо параметров-указателей использовать, где это возможно, ссылки.*

Возврат ссылок

Функция может возвращать ссылку. В программировании на C++ предусмотрено несколько применений для ссылочных значений, возвращаемых функциями. Сейчас мы продемонстрируем только некоторые из них, а другие рассмотрим ниже в этой книге, когда познакомимся с перегрузкой операторов.

Если функция возвращает ссылку, это означает, что она возвращает неявный указатель на значение, передаваемое ею в инструкции *return*. Этот факт открывает поразительные возможности: функцию, оказывается, можно использовать в левой части инструкции присваивания! Например, рассмотрим следующую простую программу.

```
// Возврат ссылки.
#include <iostream>
using namespace std;

double &f();
double val = 100.0;
int main()
{
  double newval;
  cout << f() << '\n'; // Отображаем значение val.

  newval = f(); // Присваиваем значение val переменной newval.
  cout << newval << '\n'; // Отображаем значение newval.

  f() = 99.1; // Изменяем значение val.
  cout << f() << '\n'; // Отображаем новое значение val.

  return 0;
}

double &f()
{
  return val; // Возвращаем ссылку на val.
}
```

Вот как выглядят результаты выполнения этой программы.

```
100
100
99.1
```

Рассмотрим эту программу подробнее. Судя по прототипу функции *f()*, она должна возвращать ссылку на *double*-значение. За объявлением функции *f()* следует объявление глобальной переменной *val*, которая инициализируется значением *100*. При выполнении следующей инструкции выводится исходное значение переменной *val*.

```
cout << f() << '\n'; // Отображаем значение val.
```

После вызова функция *f()* возвращает ссылку на переменную *val*. Поскольку функция *f()* объявлена с "обязательством" вернуть ссылку, при выполнении строки

```
return val; // Возвращаем ссылку на val.
```

автоматически возвращается ссылка на глобальную переменную *val*. Эта ссылка затем используется инструкцией *cout* для отображения значения *val*.

При выполнении строки

```
newval = f(); //Присваиваем значение val переменной newval.
```

ссылка на переменную *val*, возвращенная функцией *f()*, используется для присвоения значения *val* переменной *newval*.

А вот самая интересная строка в программе.

```
f() = 99.1; // Изменяем значение val.
```

При выполнении этой инструкции присваивания значение переменной *val* становится равным числу *99,1*. И вот почему: поскольку функция *f()* возвращает ссылку на переменную *val*, эта ссылка и является приемником инструкции присваивания. Таким образом, значение *99,1* присваивается переменной *val* косвенно, через ссылку на нее, которую возвращает функция *f()*.

Наконец, при выполнении строки

```
cout << f() << '\n'; // Отображаем новое значение val.
```

отображается новое значение переменной *val* (после того, как ссылка на переменную *val* будет возвращена в результате вызова функции *f()* в инструкции *cout*).

Приведем еще один пример программы, в которой в качестве значения, возвращаемого функцией, используется ссылка (или значение ссылочного типа).

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

double &change_it(int i);
// Функция возвращает ссылку.
double vals[] = {1.1, 2.2, 3.3, 4.4, 5.5};
int main()
{
  int i;
  cout << "Вот исходные значения: ";
  for(i=0; i<5; i++)
    cout << vals[i] << ' ';
  cout << '\n';

  change_it(1) = 5298.23; // Изменяем 2-й элемент.
  change_it(3) = -98.8; // Изменяем 4-й элемент.

  cout << "Вот измененные значения: ";
  for(i=0; i<5; i++)
    cout << vals[i] << ' ';
  cout << '\n';

  return 0;
}

double &change_it(int i)
{
  return vals[i]; // Возвращаем ссылку на i-й элемент.
}
```

Эта программа изменяет значения второго и четвертого элементов массива *vals*. Результаты

ее выполнения таковы.

```
Вот исходные значения: 1.1 2.2 3.3 4.4 5.5
Вот измененные значения: 1.1 5298.23 3.3 -98.8 5.5
```

Давайте разберемся, как они были получены. Функция *change_it()* объявлена как возвращающая ссылку на значение типа *double*. Говоря более конкретно, она возвращает ссылку на элемент массива *vals*, который задан ей в качестве параметра *i*. Таким образом, при выполнении следующей инструкции функции main()

```
change_it(1) = 5298.23; // Изменяем 2-й элемент.
```

функция *change_it()* возвращает ссылку на элемент *vals[1]*. Через эту ссылку элементу *vals[1]* теперь присваивается значение *5298,23*. Аналогичные события происходят при выполнении и этой инструкции.

```
change_it(3) = -98.8; // Изменяем 4-й элемент.
```

Поскольку функция *change_it()* возвращает ссылку на конкретный элемент массива *vals*, ее можно использовать в левой части инструкции для присвоения нового значения соответствующему элементу массива.

Однако, организуя возврат функцией ссылки, необходимо позаботиться о том, чтобы объект, на который она ссылается, не выходил за пределы действующей области видимости. Например, рассмотрим такую функцию.

```
// Здесь ошибка: нельзя возвращать ссылку
// на локальную переменную.
int &f()
{
  int i=10;
  return i;
}
```

При завершении функции *f()* локальная переменная *i* выйдет за пределы области видимости. Следовательно, ссылка на переменную *i*, возвращаемая функцией *f()*, будет неопределенной. В действительности некоторые компиляторы не скомпилируют функцию *f()* в таком виде, и именно по этой причине. Однако проблема такого рода может быть создана опосредованно, поэтому нужно внимательно отнестись к тому, на какой объект будет возвращать ссылку ваша функция.

Создание ограниченного массива

Ссылочный тип в качестве типа значения, возвращаемого функцией, можно с успехом применить для создания ограниченного массива. Как вы знаете, при выполнении C++-кода проверка нарушения границ при индексировании массивов не предусмотрена. Это означает, что может произойти выход за границы области памяти, выделенной для массива. Другими словами, может быть задан индекс, превышающий размер массива. Однако путем создания *ограниченного*, или *безопасного*, массива выход за его границы можно предотвратить. При работе с таким массивом любой выходящий за установленные границы индекс не допускается для индексирования массива.

Один из способов создания ограниченного массива иллюстрируется в следующей программе.

```
// Простой способ организации безопасного массива.
#include <iostream>
using namespace std;

int &put(int i); // Помещаем значение в массив.
int get(int i); // Считываем значение из массива.
```

```
int vals[10];
int error = -1;

int main()
{
  put(0) = 10; // Помещаем значения в массив.
  put(1) = 20;
  put(9) = 30;

  cout << get(0) << ' ';
  cout << get(1) << ' ';
  cout << get(9) << ' ';

  // А теперь специально генерируем ошибку.
  put(12) =1; // Индекс за пределами границ массива.
return 0;
}

// Функция занесения значения в массив.
int &put(int i)
{
  if(i>=0 && i<10)
     return vals[i]; // Возвращаем ссылку на i-й элемент.
   else {
     cout << "Ошибка нарушения границ!\n";
     return error; // Возвращаем ссылку на error.
   }
}

// Функция считывания значения из массива.
int get(int i)
{
  if(i>=0 && i<10)
     return vals[i]; // Возвращаем значение i-го элемента.
   else {
     cout << "Ошибка нарушения границ!\n";
     return error; // Возвращаем значение переменной error.
   }
}
```

Результат, полученный при выполнении этой программы, выглядит так.

```
10 20 30 Ошибка нарушения границ!
```

В этой программе создается безопасный массив, предназначенный для хранения десяти целочисленных значений. Чтобы поместить в него значение, используйте функцию *put()*, а чтобы прочитать нужный элемент массива, вызовите функцию *get()*. При использовании

обеих функций индекс интересующего вас элемента задается в виде аргумента. Как видно из текста программы, функции *get()* и *put()* не допускают выход за границы области памяти, выделенной для массива. Обратите внимание на то, что функция *put()* возвращает ссылку на заданный элемент и поэтому законно используется в левой части инструкции присваивания.

Несмотря на то что метод реализации безопасного массива, представленный в предыдущей программе, вполне корректен, возможен более удачный вариант. Как будет показано ниже в этой книге (при рассмотрении темы перегрузки операторов), программист может создать собственный безопасный массив, при работе с которым достаточно использовать стандартную систему обозначений.

Независимые ссылки

Понятие ссылки включено в C++ главным образом для поддержки способа передачи параметров "по ссылке" и для использования в качестве ссылочного типа значения, возвращаемого функцией. Несмотря на это, можно объявить независимую переменную ссылочного типа, которая и называется *независимой ссылкой*. Однако, справедливости ради, необходимо сказать, что эти независимые ссылочные переменные используются довольно редко, поскольку они могут "сбить с пути истинного" вашу программу. Сделав (для очистки совести) эти замечания, мы все же можем уделить независимым ссылкам некоторое внимание.

Независимая ссылка — *это просто еще одно название для переменных некоторого иного типа*.

Независимая ссылка должна указывать на некоторый объект. Следовательно, независимая ссылка должна быть инициализирована при ее объявлении. В общем случае это означает, что ей будет присвоен адрес некоторой ранее объявленной переменной. После этого имя такой ссылочной переменной можно применять везде, где может быть использована переменная, на которую она ссылается. И в самом деле, между ссылкой и переменной, на которую она ссылается, практически нет никакой разницы. Рассмотрим, например, следующую программу.

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

int main( )
{
  int j, k;
  int &i = j; // независимая ссылка

  j = 10;
  cout << j << " " << i; // Выводится: 10 10
  k = 121;
  i = k; // Копирует в переменную j значение переменной k, а не адрес переменной k.
  cout << "\n" << j; // Выводится: 121
  return 0;
}
```

При выполнении эта программа выводит следующие результаты.

```
10 10
121
```

Адрес, который содержит ссылочная переменная, фиксирован и его нельзя изменить. Следовательно, при выполнении инструкции *i = k* в переменную *j* (адресуемую ссылкой *i*) копируется значение переменной *k*, а не ее адрес. В качестве еще одного примера отметим, что после выполнения инструкции *i++* ссылочная переменная *i* не станет содержать новый

адрес, как можно было бы предположить. В данном случае на *1* увеличится содержимое переменной *j*.

Как было отмечено выше, независимые ссылки лучше не использовать, поскольку чаще всего им можно найти замену, а их неаккуратное применение может исказить ваш код. Согласитесь: наличие двух имен для одной и той же переменной, по сути, уже создает ситуацию, потенциально порождающую недоразумения.

Ограничения при использовании ссылок

На применение ссылочных переменных накладывается ряд следующих ограничений.

■ Нельзя ссылаться на ссылочную переменную.

■ Нельзя создавать массивы ссылок.

■ Нельзя создавать указатель на ссылку, т.е. нельзя к ссылке применять оператор *"&"*

■ Ссылки не разрешено использовать для битовых полей структур. (Битовые поля рассматриваются ниже в этой книге.)

Перегрузка функций

Перегрузка функций — *это механизм, который позволяет двум родственным функциям иметь одинаковые имена*.

В этом разделе мы узнаем об одной из самых удивительных возможностей языка C++ — перегрузке функций. В C++ несколько функций могут иметь одинаковые имена, но при условии, что их параметры будут различными. Такую ситуацию называют *перегрузкой функций* (function overloading), а функции, которые в ней задействованы, — *перегруженными* (overloaded). Перегрузка функций — один из способов реализации полиморфизма в C++.

Рассмотрим простой пример перегрузки функций.

```
// "Трехкратная" перегрузка функции f().
#include <iostream>
using namespace std;

void f(int i); // один целочисленный параметр
void f(int i, int j); // два целочисленных параметра
void f(double k); // один параметр типа double

int main()
{
  f (10); // вызов функции f(int)
  f(10, 20); // вызов функции f (int, int)
  f(12.23); // вызов функции f(double)

  return 0;
}

void f(int i)
{
  cout << "В функции f(int), i равно " << i << '\n';
}

void f(int i, int j)
{
```

```
  cout << "В функции f(int, int), i равно " << i;
  cout << ", j равно " << j << '\n';
}

void f(double k)
{
  cout << "В функции f(double), k равно " << k << ' \n';
}
```

При выполнении эта программа генерирует следующие результаты.

```
В функции f(int), i равно 10
В функции f(int, int), i равно 10, j равно 20
В функции f(double), к равно 12.23
```

Как видите, функция *f()* перегружается три раза. Первая версия принимает один целочисленный параметр, вторая — два целочисленных параметра, а третья — один double-параметр. Поскольку списки параметров для всех трех версий различны, компилятор обладает достаточной информацией, чтобы вызвать правильную версию каждой функции. В общем случае для создания перегрузки некоторой функции достаточно объявить различные ее версии.

Для определения того, какую версию перегруженной функции вызвать, компилятор использует *тип* и/или *количество аргументов*. Таким образом, перегруженные функции должны отличаться *типами* и/или *числом параметров*. Несмотря на то что перегруженные методы могут отличаться и типами возвращаемых значений, этого вида информации недостаточно для C++, чтобы во всех случаях компилятор мог решить, какую именно функцию нужно вызвать.

Чтобы лучше понять выигрыш от перегрузки функций, рассмотрим три функции из стандартной библиотеки: *abs()*, *labs()* и *fabs()*. Они были впервые определены в языке C, а затем ради совместимости включены в C++. Функция *abs()* возвращает абсолютное значение (модуль) целого числа, функция *labs()* возвращает модуль длинного целочисленного значения (типа *long*), а *fabs()* — модуль значения с плавающей точкой (типа *double*). Поскольку язык C не поддерживает перегрузку функций, каждая функция должна иметь собственное имя, несмотря на то, что все три функции выполняют, по сути, одно и то же действие. Это делает ситуацию сложнее, чем она есть на самом деле. Другими словами, при одних и тех же действиях программисту необходимо помнить имена всех трех (в данном случае) функций вместо одного. Но в C++, как показано в следующем примере, можно использовать только одно имя для всех трех функций.

```
// Создание функций myabs( ) — перегруженной версии функции abs( ).
#include <iostream>
using namespace std;

// Функция myabs( ) перегружается тремя способами.
int myabs(int i);
double myabs(double d);
long myabs(long l);

int main( )
{
  cout << myabs(-10) << "\n";
  cout << myabs(-11.0) << "\n";
```

```
  cout << myabs(-9L) << "\n";
  return 0;
}
int myabs(int i)
{
  cout << "Использование int-функции myabs(): ";
  if(i<0) return -i;
  else return i;
}
double myabs(double d)
{
  cout << "Использование double-функции myabs(): ";
  if(d<0.0) return -d;
  else return d;
}
long myabs(long l)
{
  cout << "Использование long-функции myabs(): ";
  if(1<0) return -1;
  else return 1;
}
```

Результаты выполнения этой программы таковы.

```
Использование int-функции myabs(): 10
Использование double-функции myabs(): 11
Использование long-функции myabs(): 9
```

При выполнении эта программа создает три похожие, но все же различные функции, вызываемые с использованием "общего" (одного на всех) имени *myabs*. Каждая из них возвращает абсолютное значение своего аргумента. Во всех ситуациях вызова компилятор "знает", какую именно функцию ему использовать. Для принятия решения ему достаточно "взглянуть" на тип аргумента, передаваемого функции. Принципиальная значимость перегрузки состоит в том, что она позволяет обращаться к связанным функциям посредством одного, общего для всех, имени. Следовательно, имя *myabs* представляет общее действие, которое выполняется во всех случаях. Компилятору остается правильно выбрать конкретную версию при конкретных обстоятельствах. Благодаря полиморфизму программисту нужно помнить не три различных имени, а только одно. Несмотря на простоту приведенного примера, он позволяет понять, насколько перегрузка способна упростить процесс программирования.

Каждая версия перегруженной функции может выполнять любые действия. Другими словами, не существует правила, которое бы обязывало программиста связывать перегруженные функции общими действиями. Однако с точки зрения стилистики перегрузка функций все-таки подразумевает определенное "родство" его версий. Таким образом, несмотря на то, что одно и то же имя можно использовать для перегрузки не связанных общими действиями функций, этого делать не стоит. Например, в принципе можно использовать имя *sqr* для создания функции, которая возвращает квадрат целого числа, и функции, которая возвращает значение квадратного корня из вещественного числа

(типа double). Но, поскольку эти операции фундаментально различны, применение механизма перегрузки методов в этом случае сводит на нет его первоначальную цель. (Такой стиль программирования, наверное, подошел бы лишь для того, чтобы ввести в заблуждение конкурента.) На практике перегружать имеет смысл только тесно связанные операции.

Анахронизм в виде ключевого слова overload

На заре создания C++ перегруженные функции необходимо было явным образом объявлять таковыми с помощью ключевого слова *overload*. Это ключевое слово больше не требуется в C++. В действительности стандартом C++ оно даже не включено в список ключевых слов. Однако время от времени его можно встретить в каком-нибудь C++-коде, особенно в старых книгах и статьях.

Общая форма использования ключевого слова *overload* такова.

```
overload func_name;
```

Здесь элемент *func_name* представляет собой имя перегружаемой функции. Эта инструкция должна предшествовать объявлениям перегруженных функций. (В общем случае оно встречается в начале программы.) Например, если функция *Counter()* является перегруженной, то в программу могла быть включена такая строка.

```
overload Counter;
```

Если вы встретите *overload*-объявления при работе со старыми программами, их можно просто удалить: они больше не нужны. Поскольку ключевое слово *overload* — анахронизм, его не следует использовать в новых C++-программах. На самом деле большинство компиляторов его попросту не воспримет.

Аргументы, передаваемые функции по умолчанию

В C++ мы можем придать параметру некоторое значение, которое будет автоматически использовано, если при вызове функции не задается аргумент, соответствующий этому параметру. Аргументы, передаваемые функции по умолчанию, можно использовать, чтобы упростить обращение к сложным функциям, а также в качестве "сокращенной формы" перегрузки функций.

Задание аргументов, передаваемых функции по умолчанию, синтаксически аналогично инициализации переменных. Рассмотрим следующий пример, в котором объявляется функция *myfunc()*, принимающая один аргумент типа *double* с действующим по умолчанию значением *0.0* и один символьный аргумент с действующим по умолчанию значением *'X'*.

```
void myfunc(double num = 0.0, char ch = 'X')
{
  .
  .
  .
}
```

После такого объявления функцию *myfunc()* можно вызвать одним из трех следующих способов.

```
myfunc(198.234, 'A'); // Передаем явно заданные значения.
myfunc(10.1); // Передаем для параметра num значение 10.1, а для параметра ch позволяем применить аргумент, задаваемый по умолчанию ('X').
myfunc(); // Для обоих параметров num и ch позволяем применить аргументы, задаваемые по умолчанию.
```

При первом вызове параметру *num* передается значение *198.234*, а параметру *ch* — символ *'A'*. Во время второго вызова параметру *num* передается значение *10.1*, а параметр *ch* по

'A'. Во время второго вызова параметру *num* передается значение *10.1*, а параметр *ch* по умолчанию устанавливается равным символу *'X'*. Наконец, в результате третьего вызова как параметр *num*, так и параметр *ch* по умолчанию устанавливаются равными значениям, заданным в объявлении функции.

Включение в C++ возможности передачи аргументов по умолчанию позволяет программистам упрощать код программ. Чтобы предусмотреть максимально возможное количество ситуаций и обеспечить их корректную обработку, функции часто объявляются с большим числом параметров, чем необходимо в наиболее распространенных случаях. Поэтому благодаря применению аргументов по умолчанию программисту нужно указывать не все аргументы (используемые в общем случае), а только те, которые имеют смысл для определенной ситуации.

Аргумент, передаваемый функции по умолчанию, представляет собой значение, которое будет автоматически передано параметру функции в случае, если аргумент, соответствующий этому параметру, явным образом не задан.

Насколько полезна возможность передачи аргументов по умолчанию, показано на примере функции *clrscr()*, представленной в следующей программе. Функция *clrscr()* очищает экран путем вывода последовательности символов новой строки (это не самый эффективный способ, но он очень подходит для данного примера). Поскольку в наиболее часто используемом режиме представления видеоизображений на экран дисплея выводится *25* строк текста, то в качестве аргумента по умолчанию используется значение *25*. Но так как в других видеорежимах на экране может отображаться больше или меньше *25* строк, аргумент, действующий по умолчанию, можно переопределить, явно указав нужное значение.

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

void clrscr(int size=25);
int main( )
{
  int i;
  for(i=0; i<30; i++ ) cout << i << '\n';
  clrscr( ); // Очищаем 25 строк.

  for(i=0; i<30; i++ ) cout << i << '\n';
  clrscr(10); // Очищаем 10 строк.

  return 0;
}

void clrscr(int size)
{
  for(; size; size--) cout << '\n';
}
```

Как видно из кода этой программы, если значение, действующее по умолчанию, соответствует ситуации, при вызове функции *clrscr()* аргумент указывать не нужно. Но в других случаях аргумент, действующий по умолчанию, можно переопределить и передать параметру *size* нужное значение.

При создании функций, имеющих значения аргументов, передаваемых по умолчанию, необходимо помнить две вещи. Эти значения по умолчанию должны быть заданы только однажды, причем при первом объявлении функции в файле. В предыдущем примере

аргумент по умолчанию был задан в прототипе функции *clrscr()*. При попытке определить новые (или даже те же) передаваемые по умолчанию значения аргументов в определении функции *clrscr()* компилятор отобразит сообщение об ошибке и не скомпилирует вашу программу.

Несмотря на то что передаваемые по умолчанию аргументы должны быть определены только один раз, для каждой версии перегруженной функции для передачи по умолчанию можно задавать различные аргументы. Таким образом, разные версии перегруженной функции могут иметь различные значения аргументов, действующие по умолчанию.

Важно понимать, что все параметры, которые принимают значения по умолчанию, должны быть расположены справа от остальных. Например, следующий прототип функции содержит ошибку.

```
// Неверно!
void f(int a = 1, int b);
```

Если вы начали определять параметры, которые принимают значения по умолчанию, нельзя после них указывать параметры, задаваемые при вызове функции только явным образом. Поэтому следующее объявление также неверно и не будет скомпилировано.

```
int myfunc(float f, char *str, int i=10, int j);
```

Поскольку для параметра *i* определено значение по умолчанию, для параметра *j* также нужно задать значение по умолчанию.

Сравнение возможности передачи аргументов по умолчанию с перегрузкой функций

Как упоминалось в начале этого раздела, одним из применений передачи аргументов по умолчанию является "сокращенная форма" перегрузки функций. Чтобы понять это, представьте, что вам нужно создать две "адаптированные" версии стандартной функции *strcat()*. Одна версия должна присоединять все содержимое одной строки к концу другой. Вторая же принимает третий аргумент, который задает количество конкатенируемых (присоединяемых) символов. Другими словами, эта версия должна конкатенировать только заданное количество символов одной строки к концу другой.

Допустим, что вы назвали свои функции именем *mystrcat()* и предложили такой вариант их прототипов.

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

Первая версия должна скопировать *len* символов из строки *s2* в конец строки *s1*. Вторая версия копирует всю строку, адресуемую указателем *s2*, в конец строки, адресуемой указателем *s1*, т.е. действует подобно стандартной функции *strcat()*.

Несмотря на то что для достижения поставленной цели можно реализовать две версии функции *mystrcat()*, есть более простой способ решения этой задачи. Используя возможность передачи аргументов по умолчанию, можно создать только одну функцию *mystrcat()*, которая заменит обе задуманные ее версии. Реализация этой идеи продемонстрирована в следующей программе.

```
// Применение пользовательской версии функции strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);
int main()
{
  char str1[80] = "Это тест.";
```

```
  char str2[80] = "0123456789";

  mystrcat(str1, str2, 5); // Присоединяем 5 символов.
  cout << str1 << '\n';

  strcpy(str1, "Это тест."); // Восстанавливаем str1.

  mystrcat(str1, str2); // Присоединяем всю строку.
  cout << str1 << '\n';

  return 0;
}

// Пользовательская версия функции strcat().
void mystrcat(char *s1, char *s2, int len)
{
  // Находим конец строки s1.
  while(*s1) s1++;
  if(len == -1) len = strlen(s2);
  while(*s2 && len) {
    *s1 = *s2; // Копируем символы.
    s1++; s2++; len--;
  }
  *s1 = '\0'; // Завершаем строку s1 нулевым символом.
}
```

Здесь функция *mystrcat()* присоединяет *len* символов строки, адресуемой параметром *s2*, к концу строки, адресуемой параметром *s1*. Но если значение *len* равно *-1*, как и в случае разрешения передачи этого аргумента по умолчанию, функция *mystrcat()* присоединит к строке *s1* всю строку, адресуемую параметром *s2*. (Другими словами, если значение *len* равно *-1*, функция *mystrcat()* действует подобно стандартной функции *strcat()*.) Используя для параметра *len* возможность передачи аргумента по умолчанию, обе операции можно объединить в одной функции.

Этот пример позволил продемонстрировать, как аргументы, передаваемые функции по умолчанию, обеспечивают основу для сокращенной формы объявления перегруженных функций.

Об использовании аргументов, передаваемых по умолчанию

Несмотря на то что аргументы, передаваемые функции по умолчанию, — очень мощное средство программирования (при их корректном использовании), с ними могут иногда возникать проблемы. Их назначение — позволить функции эффективно выполнять свою работу, обеспечивая при всей простоте этого механизма значительную гибкость. В этом смысле все передаваемые по умолчанию аргументы должны отражать способ наиболее общего использования функции или альтернативного ее применения. Если не существует некоторого единого значения, которое обычно присваивается тому или иному параметру, то и нет смысла объявлять соответствующий аргумент по умолчанию. На самом деле объявление аргументов, передаваемых функции по умолчанию, при недостаточном для этого основании деструктуризирует код, поскольку такие аргументы способны сбить с толку любого, кому придется разбираться в такой программе. Наконец, основным принципом использования аргументов по умолчанию должен быть, как у врачей, принцип *"не навредить"*. Другими словами, случайное использование аргумента по умолчанию не должно привести к необратимым отрицательным последствиям. Ведь такой аргумент можно просто забыть указать при вызове некоторой функции, и, если это случится,

Можно просто забыть указать при вызове некоторой функции, и, если это случится, подобный промах не должен вызвать, например, потерю важных данных!

Перегрузка функций и неоднозначность

Неоднозначность возникает тогда, когда компилятор не может определить различие между двумя перегруженными функциями.

Прежде чем завершить эту главу, мы должны исследовать вид ошибок, уникальный для C++: неоднозначность. Возможны ситуации, в которых компилятор не способен сделать выбор между двумя (или более) корректно перегруженными функциями. Такие ситуации и называют *неоднозначными*. Инструкции, создающие неоднозначность, являются ошибочными, а программы, которые их содержат, скомпилированы не будут.

Основной причиной неоднозначности в C++ является автоматическое преобразование типов. В C++ делается попытка автоматически преобразовать тип аргументов, используемых для вызова функции, в тип параметров, определенных функцией. Рассмотрим пример.

```
int myfunc(double d);
.
.
.
cout << myfunc('c'); // Ошибки нет, выполняется преобразование типов.
```

Как отмечено в комментарии, ошибки здесь нет, поскольку C++ автоматически преобразует символ *'c'* в его *double*-эквивалент. Вообще говоря, в C++ запрещено довольно мало видов преобразований типов. Несмотря на то что автоматическое преобразование типов — это очень удобно, оно, тем не менее, является главной причиной неоднозначности. Рассмотрим следующую программу.

```
// Неоднозначность вследствие перегрузки функций.
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main( )
{
  // Неоднозначности нет, вызывается функция myfunc(double).
  cout << myfunc (10.1) << " ";

  // Неоднозначность.
  cout << myfunc(10);
  return 0;
}

float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
```

```
{
  return -i;
}
```

Здесь благодаря перегрузке функция *myfunc()* может принимать аргументы либо типа *float*, либо типа *double*. При выполнении строки кода

```
cout << myfunc (10.1) << " ";
```

не возникает никакой неоднозначности: компилятор "уверенно" обеспечивает вызов функции *myfunc(double)*, поскольку, если не задано явным образом иное, все литералы с плавающей точкой в C++ автоматически получают тип *double*. Но при вызове функции *myfunc()* с аргументом, равным целому числу *10*, в программу вносится неоднозначность, поскольку компилятору неизвестно, в какой тип ему следует преобразовать этот аргумент: *float* или *double*. Оба преобразования допустимы. В такой неоднозначной ситуации будет выдано сообщение об ошибке, и программа не скомпилируется.

На примере предыдущей программы хотелось бы подчеркнуть, что неоднозначность в ней вызвана не перегрузкой функции *myfunc()*, объявленной дважды для приема *double*- и *float*-аргумента, а использованием при конкретном вызове функции *myfunc()* аргумента неопределенного для преобразования типа. Другими словами, ошибка состоит не в перегрузке функции *myfunc()*, а в конкретном ее вызове.

А вот еще один пример неоднозначности, вызванной автоматическим преобразованием типов в C++.

```
// Еще одна ошибка, вызванная неоднозначностью.
#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main( )
{
  cout << myfunc('c'); // Здесь вызывается myfunc(char).
  cout << myfunc(88) << " "; // Вносится неоднозначность.
  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
}

char myfunc(char ch)
{
  return ch+1;
}
```

В C++ типы *unsigned char* и *char* не являются существенно неоднозначными. (Это — различные типы.) Но при вызове функции *myfunc()* с целочисленным аргументом *88* компилятор *"не знает"*, какую функцию ему выполнить, т.е. в значение какого типа ему следует преобразовать число 88: типа *char* или типа *unsigned char*? Оба преобразования здесь вполне допустимы.

Неоднозначность может быть также вызвана использованием в перегруженных функциях аргументов, передаваемых по умолчанию. Для примера рассмотрим следующую программу.

```
// Еще один пример неоднозначности.
#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
  cout << myfunc(4, 5) << " "; // неоднозначности нет
  cout << myfunc(10); // неоднозначность
  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

Здесь в первом обращении к функции *myfunc()* задается два аргумента, поэтому у компилятора нет никаких сомнений в выборе нужной функции, а именно *myfunc(int i, int j)*, т.е. никакой неоднозначности в этом случае не привносится. Но при втором обращении к функции *myfunc()* мы получаем неоднозначность, поскольку компилятор *"не знает"*, то ли ему вызвать версию функции *myfunc()*, которая принимает один аргумент, то ли использовать возможность передачи аргумента по умолчанию к версии, которая принимает два аргумента.

Программируя на языке C++, вам еще не раз придется столкнуться с ошибками неоднозначности, которые, к сожалению, очень легко "проникают" в программы, и только опыт и практика помогут вам избавиться от них.

Глава 9: Еще о типах данных и операторах

Прежде чем переходить к более сложным средствам C++, имеет смысл подробнее познакомиться с некоторыми типами данных и операторами. Кроме уже рассмотренных нами типов данных, в C++ определены и другие. Одни из них состоят из модификаторов, добавляемых к уже известным вам типам. Другие включают перечисления, а третьи используют ключевое слово *typedef*. C++ также поддерживает ряд операторов, которые значительно расширяют область действия языка и позволяют решать задачи программирования в весьма широком диапазоне. Речь идет о поразрядных операторах, операторах сдвига, а также операторах *"?"* и *sizeof*. Кроме того, в этой главе рассматриваются такие специальные операторы, как *new* и *delete*. Они предназначены для поддержки C++-системы динамического распределения памяти.

Спецификаторы типа const и volatile

Спецификаторы типа const и volatile управляют доступом к переменной

Спецификаторы типа const и volatile управляют доступом к переменной.

В C++ определено два спецификатора типа, которые оказывают влияние на то, каким образом можно получить доступ к переменным или модифицировать их. Это спецификаторы *const* и *volatile*. Официально они именуются *cv-спецификаторами* и должны предшествовать базовому типу при объявлении переменной.

Спецификатор типа const

Переменные, объявленные с использованием спецификатора *const*, не могут изменить свои значения во время выполнения программы. Однако любой *const*-переменной можно присвоить некоторое начальное значение. Например, при выполнении инструкции

```
const double version = 3.2;
```

создается *double*-переменная *version*, которая содержит значение *3.2*, и это значение программа изменить уже не может. Но эту переменную можно использовать в других выражениях. Любая *const*-переменная получает значение либо во время явно задаваемой инициализации, либо при использовании аппаратно-зависимых средств. Применение спецификатора *const* к объявлению переменной гарантирует, что она не будет модифицирована другими частями вашей программы.

Спецификатор const предотвращает модификацию переменной при выполнении программы.

Спецификатор *const* имеет ряд важных применений. Возможно, чаще всего его используют для создания *const*-параметров типа указатель. Такой параметр-указатель защищает объект, на который он ссылается, от модификации со стороны функции. Другими словами, если параметр-указатель предваряется ключевым словом *const*, никакая инструкция этой функции не может модифицировать переменную, адресуемую этим параметром. Например, функция *code()* в следующей короткой программе сдвигает каждую букву в сообщении на одну алфавитную позицию (т.е. вместо буквы *'А'* ставится буква *'Б'* и т.д.), отображая таким образом сообщение в закодированном виде. Использование спецификатора *const* в объявлении параметра не позволяет коду функции модифицировать объект, на который указывает этот параметр.

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

void code(const char *str);

int main( )
{
  code("Это тест.");
  return 0;
}

/* Использование спецификатора const гарантирует, что str не может изменить аргумент, на который он указывает.
*/
void code(const char *str)
{
  while(*str) {
    cout << (char) (*str+1);
    str++;
  }
}
```

```
}
```

Поскольку параметр *str* объявляется как *const*-указатель, у функции *code()* нет никакой возможности внести изменения в строку, адресуемую параметром *str*. Но если вы попытаетесь написать функцию *code()* так, как показано в следующем примере, то обязательно получите сообщение об ошибке, и программа не скомпилируется.

```
// Этот код неверен.
void code(const char *str)
{
  while(*str) {
    *str = *str + 1; // Ошибка, аргумент модифицировать нельзя.
    cout << (char) *str;
    str++;
  }
}
```

Поскольку параметр *str* является *const*-указателем, его нельзя использовать для модификации объекта, на который он ссылается.

Спецификатор *const* можно также использовать для ссылочных параметров, чтобы не допустить в функции модификацию переменных, на которые ссылаются эти параметры. Например, следующая программа некорректна, поскольку функция *f()* пытается модифицировать переменную, на которую ссылается параметр *i*.

```
// Нельзя модифицировать const-ссылки.
#include <iostream>
using namespace std;

void f(const int &i);

int main( )
{
  int к = 10;
  f(к);
  return 0;
}

// Использование ссылочного const-параметра.
void f (const int &i)
{
  i = 100; // Ошибка, нельзя модифицировать const-ссылку.
  cout << i;
}
```

Спецификатор *const* еще можно использовать для подтверждения того, что ваша программа не изменяет значения некоторой переменной. Вспомните, что переменная типа *const* может быть модифицирована внешними устройствами, т.е. ее значение может быть установлено каким-нибудь аппаратным устройством (например, датчиком). Объявив переменную с помощью спецификатора *const*, можно доказать, что любые изменения, которым подвергается эта переменная, вызваны исключительно внешними событиями.

Наконец, спецификатор *const* используется для создания именованных констант. Часто в программах многократно применяется одно и то же значение для различных целей. Например, необходимо объявить несколько различных массивов таким образом, чтобы все

они имели одинаковый размер. Когда нужно использовать подобное "магическое число", имеет смысл реализовать его в виде *const*-переменной. Затем вместо реального значения можно использовать имя этой переменной, а если это значение придется впоследствии изменить, вы измените его только в одном месте программы. Следующая программа позволяет попробовать этот вид применения спецификатора *const* "на вкус".

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

const int size = 10;

int main( )
{
  int A1[size], A2[size], A3[size];
  // . . .
}
```

Если в этом примере понадобится использовать новый размер для массивов, вам потребуется изменить только объявление переменной *size* и перекомпилировать программу. В результате все три массива автоматически получат новый размер.

Спецификатор типа volatile

Спецификатор volatile информирует компилятор о том, что данная переменная может быть изменена внешними (по отношению к программе) факторами.

Спецификатор *volatile* сообщает компилятору о том, что значение соответствующей переменной может быть изменено в программе неявным образом. Например, адрес некоторой глобальной переменной может передаваться управляемой прерываниями подпрограмме тактирования, которая обновляет эту переменную с приходом каждого импульса сигнала времени. В такой ситуации содержимое переменной изменяется без использования явно заданных инструкций программы. Существуют веские основания для того, чтобы сообщить компилятору о внешних факторах изменения переменной. Дело в том, что C++-компилятору разрешается автоматически оптимизировать определенные выражения в предположении, что содержимое той или иной переменной остается неизменным, если оно не находится в левой части инструкции присваивания. Но если некоторые факторы (внешние по отношению к программе) изменят значение этого поля, такое предположение окажется неверным, в результате чего могут возникнуть проблемы.

Например, в следующем фрагменте программы предположим, что переменная *clock* обновляется каждую миллисекунду часовым механизмом компьютера. Но, поскольку переменная *clock* не объявлена с использованием спецификатора *volatile*, этот фрагмент кода может иногда работать недолжным образом. (Обратите особое внимание на строки, обозначенные буквами *"А"* и *"Б"*.)

```
int clock, timer;
  // ...
timer = clock; // строка А
// ... Какие-нибудь действия.

cout << "Истекшее время " << clock-timer; // строка Б
```

В этом фрагменте переменная *clock* получает свое значение, когда она присваивается переменной *timer* в строке А. Но, поскольку переменная *clock* не объявлена с использованием спецификатора *volatile*, компилятор волен оптимизировать этот код, причем таким способом, при котором значение переменной *clock*, возможно, не будет опрошено в инструкции *cout* (строка Б), если между строками *А* и *Б* не будет ни одного промежуточного присваивания значения переменной *clock*. (Другими словами, в строке *Б*

компилятор может просто еще раз использовать значение, которое получила переменная *clock* в строке *А*.) Но если между моментами выполнения строк *А* и *Б* поступят очередные импульсы сигнала времени, то значение переменной *clock* обязательно изменится, а строка *Б* в этом случае не отразит корректный результат.

Для решения этой проблемы необходимо объявить переменную *clock* с ключевым словом *volatile*.

```
volatile int clock;
```

Теперь значение переменной *clock* будет опрашиваться при каждом ее использовании.

И хотя на первый взгляд это может показаться странным, спецификаторы *const* и *volatile* можно использовать вместе. Например, следующее объявление абсолютно допустимо. Оно создает *const*-указатель на *volatile*-объект.

```
const volatile unsigned char *port = (const volatile char *) 0x2112;
```

В этом примере для преобразования целочисленного литерала *0x2112* в *const*-указатель на *volatile*-символ необходимо применить операцию приведения типов.

Спецификаторы классов памяти

C++ поддерживает пять спецификаторов классов памяти:

`auto`

`extern`

`register`

`static`

`mutable`

Спецификаторы классов памяти определяют, как должна храниться переменная.

С помощью этих ключевых слов компилятор получает информацию о том, как должна храниться переменная. Спецификатор классов памяти необходимо указывать в начале объявления переменной.

Спецификатор *mutable* применяется только к объектам классов, о которых речь впереди. Остальные спецификаторы мы рассмотрим в этом разделе.

Спецификатор класса памяти auto

Редко используемый спецификатор auto объявляет локальную переменную.

Спецификатор *auto* объявляет локальную переменную. Но он используется довольно редко (возможно, вам никогда и не доведется применить его), поскольку локальные переменные являются "автоматическими" по умолчанию. Вряд ли вам попадется это ключевое слово и в чужих программах.

Спецификатор класса памяти extern

Все программы, которые мы рассматривали до сих пор, имели довольно скромный размер. Реальные же компьютерные программы гораздо больше. По мере увеличения размера файла, содержащего программу, время компиляции становится иногда раздражающе долгим. В этом случае следует разбить программу на несколько отдельных файлов. После этого небольшие изменения, вносимые в один файл, не потребуют перекомпиляции всей программы. При разработке больших проектов такой многофайловый подход может сэкономить существенное время. Реализовать этот подход позволяет ключевое слово *extern*.

В программах, которые состоят из двух или более файлов, каждый файл должен "знать" имена и типы глобальных переменных, используемых программой в целом. Однако нельзя просто объявить копии глобальных переменных в каждом файле. Дело в том, что в C++ программа может включать только одну копию каждой глобальной переменной. Следовательно, если вы попытаетесь объявить необходимые глобальные переменные в каждом файле, возникнут проблемы. Когда компоновщик попытается скомпоновать эти

файлы, он обнаружит дублированные глобальные переменные, и компоновка программы не состоится. Чтобы выйти из этого затруднительного положения, достаточно объявить все глобальные переменные в одном файле, а в других использовать *extern*-объявления, как показано на рис. 9.1.

Спецификатор extern объявляет переменную, но не выделяет для нее области памяти.

Файл F1

```
int x, y:
char ch;

int main()
{
   // ...
}

void func1()
{
   x = 123;
}
```

Файл F2

```
extern int x, y:
extern char ch;

void func22()
{
   x = y/10;
}

void func23()
{
   y = 10;
}
```

Рис. 9.1. Использование глобальных переменных в отдельно компилируемых модулях

В файле *F1* объявляются и определяются переменные *x*, *y* и *ch*. В файле *F2* используется скопированный из файла *F1* список глобальных переменных, к объявлению которых добавлено ключевое слово *extern*. Спецификатор *extern* делает переменную известной для модуля, но в действительности не создает ее. Другими словами, ключевое слово *extern* предоставляет компилятору информацию о типе и имени глобальных переменных, повторно не выделяя для них памяти. Во время компоновки этих двух модулей все ссылки на эти внешние переменные будут определены.

До сих пор мы не уточняли, в чем состоит различие между объявлением и определением переменной, но здесь это очень важно. При объявлении, переменной присваивается *имя* и *mun*, а посредством определения для переменной выделяется память. В большинстве случаев объявления переменных одновременно являются определениями. Предварив имя переменной спецификатором *extern*, можно объявить переменную, не определяя ее.

Существует еще одно применение для ключевого слова *extern*, которое не связано с многофайловыми проектами. Не секрет, что много времени уходит на объявления глобальных переменных, которые, как правило, приводятся в начале программы, но это не всегда обязательно. Если функция использует глобальную переменную, которая определяется ниже (в том же файле), в теле функции ее можно специфицировать как внешнюю (с помощью ключевого слова *extern*). При обнаружении определения этой переменной компилятор вычислит соответствующие ссылки на нее.

Рассмотрим следующий пример. Обратите внимание на то, что глобальные переменные *first* и *last* объявляются не перед, а после функции *main()*.

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

```
int main( )
{
  extern int first, last; // Использование глобальных переменных.
  cout << first << " " << last << "\n";
  return 0;
}

// Глобальное определение переменных first и last.
int first = 10, last = 20;
```

При выполнении этой программы на экран будут выведены числа *10* и *20*, поскольку глобальные переменные *first* и *last*, используемые в инструкции *cout*, инициализируются этими значениями. Поскольку *extern*-объявление в функции *main()* сообщает компилятору о том, что переменные *first* и *last* объявляются где-то в другом месте (в данном случае ниже, но в том же файле), программу можно скомпилировать без ошибок, несмотря на то, что переменные *first* и *last* используются до их определения.

Важно понимать, что *extern*-объявления переменных, показанные в предыдущей программе, необходимы здесь только по той причине, что переменные *first* и *last* не были определены до их использования в функции *main()*. Если бы их определения компилятор обнаружил раньше определения функции *main()*, необходимости в *extern*-инструкции не было бы. Помните, если компилятор обнаруживает переменную, которая не была объявлена в текущем блоке, он проверяет, не совпадает ли она с какой-нибудь из переменных, объявленных внутри других включающих блоков. Если нет, компилятор просматривает ранее объявленные глобальные переменные. Если обнаруживается совпадение их имен, компилятор предполагает, что ссылка была именно на эту глобальную переменную. Спецификатор *extern* необходим только в том случае, если вы хотите использовать переменную, которая объявляется либо ниже в том же файле, либо в другом.

И еще. Несмотря на то что спецификатор *extern* объявляет, но не определяет переменную, существует одно исключение из этого правила. Если в *extern*-объявлении переменная инициализируется, то такое *extern*-объявление становится определением. Это очень важный момент, поскольку любой объект может иметь несколько объявлений, но только одно определение.

Статические переменные

Переменные типа *static* — это переменные "долговременного" хранения, т.е. они хранят свои значения в пределах своей функции или файла. От глобальных они отличаются тем, что за рамками своей функции или файла они неизвестны. Поскольку спецификатор *static* по-разному определяет "судьбу" локальных и глобальных переменных, мы рассмотрим их в отдельности.

Локальные static-переменные

Локальная static-переменная поддерживает свое значение между вызовами функции.

Если к локальной переменной применен модификатор *static*, то для нее выделяется постоянная область памяти практически так же, как и для глобальной переменной. Это позволяет статической переменной поддерживать ее значение между вызовами функций. (Другими словами, в отличие от обычной локальной переменной, значение *static*-переменной не теряется при выходе из функции.) Ключевое различие между статической локальной и глобальной переменными состоит в том, что статическая локальная переменная известна только блоку, в котором она объявлена. Таким образом, статическую локальную переменную в некоторой степени можно назвать глобальной переменной, которая имеет ограниченную область видимости.

Чтобы объявить статическую переменную, достаточно предварить ее тип ключевым словом *static*. Например, при выполнении этой инструкции переменная *count* объявляется статической

статической:

```
static int count;
```

Статической переменной можно присвоить некоторое начальное значение. Например, в этой инструкции переменной *count* присваивается начальное значение *200*:

```
static int count = 200;
```

Локальные *static*-переменные инициализируются только однажды, в начале выполнения программы, а не при каждом входе в функцию, в которой они объявлены.

Возможность использования статических локальных переменных важна для создания независимых функций, поскольку существуют такие типы функций, которые должны сохранять их значения между вызовами. Если бы статические переменные не были предусмотрены в C++, пришлось бы использовать вместо них глобальные, что открыло бы путь для всевозможных побочных эффектов.

Рассмотрим пример использования *static*-переменной. Она служит для хранения текущего среднего значения от чисел, вводимых пользователем.

```
/* Вычисляем текущее среднее значение от чисел, вводимых пользователем.
*/
#include <iostream>
using namespace std;

int r_avg(int i);

int main( )
{
  int num;
  do {
    cout << "Введите числа (-1 означает выход): ";
      cin >> num;
    if(num != -1)
      cout << "Текущее среднее равно: " << r_avg(num);
    cout << '\n';
  }while(num > -1);
  return 0;
}

// Вычисляем текущее среднее.
int r_avg(int i)
{
  static int sum=0, count=0;
  sum = sum + i;
  count++;
  return sum / count;
}
```

Здесь обе локальные переменные *sum* и *count* объявлены статическими и инициализированы значением *0*. Помните, что для статических переменных инициализация выполняется только один раз (при первом выполнении функции), а не при каждом входе в функцию. В этой программе функция *r_avg()* используется для вычисления текущего среднего значения от чисел, вводимых пользователем. Поскольку обе переменные *sum* и

count являются статическими, они поддерживают свои значения между вызовами функции *r_avg()*, что позволяет нам получить правильный результат вычислений. Чтобы убедиться в необходимости модификатора *static*, попробуйте удалить его из программы. После этого программа не будет работать корректно, поскольку промежуточная сумма будет теряться при каждом выходе из функции *r_avg()*.

Глобальные static-переменные

Глобальная static-переменная известна только для файла, в котором она объявлена.

Если модификатор *static* применен к глобальной переменной, то компилятор создаст глобальную переменную, которая будет известна только для файла, в котором она объявлена. Это означает, что, хотя эта переменная является глобальной, другие функции в других файлах не имеют о ней "ни малейшего понятия" и не могут изменить ее содержимое. Поэтому она и не может стать "жертвой" несанкционированных изменений. Следовательно, для особых ситуаций, когда локальная статичность оказывается бессильной, можно создать небольшой файл, который будет содержать лишь функции, использующие глобальные *static*-переменные, отдельно скомпилировать этот файл и работать с ним, не опасаясь вреда от побочных эффектов "всеобщей глобальности".

Рассмотрим пример, который представляет собой переработанную версию программы (из предыдущего раздела), вычисляющей текущее среднее значение. Эта версия состоит из двух файлов и использует глобальные *static*-переменные для хранения значений промежуточной суммы и счетчика вводимых чисел.

```
//--------------------Первый файл--------------------
#include <iostream>
using namespace std;

int r_avg(int i);
void reset();

int main()
{
  int num;
  do {
    cout <<"Введите числа (-1 для выхода, -2 для сброса): ";
      cin >> num;
    if(num==-2) {
      reset();
      continue;
    }
    if(num != -1)
      cout << "Среднее значение равно: " << r_avg(num);
    cout << '\n';
  }while(num != -1);
  return 0;
}

//--------------------Второй файл--------------------
#include <iostream>
static int sum=0, count=0;
```

```
static int sum = 0, count = 0;

int r_avg(int i)
{
  sum = sum + i;
  count++;
  return sum / count;
}

void reset()
{
  sum = 0;
  count = 0;
}
```

В этой версии программы переменные *sum* и *count* являются глобально статическими, т.е. их глобальность ограничена вторым файлом. Итак, они используются функциями *r_avg()* и *reset()*, причем обе они расположены во втором файле. Этот вариант программы позволяет сбрасывать накопленную сумму (путем установки в исходное положение переменных *sum* и *count*), чтобы можно было усреднить другой набор чисел. Но ни одна из функций, расположенных вне второго файла, не может получить доступ к этим переменным. Работая с данной программой, можно обнулить предыдущие накопления, введя число *-2*. В этом случае будет вызвана функция *reset()*. Проверьте это. Кроме того, попытайтесь получить из первого файла доступ к любой из переменных *sum* или *count*. (Вы получите сообщение об ошибке.)

Итак, имя локальной *static*-переменной известно только функции или блоку кода, в котором она объявлена, а имя глобальной *static*-переменной — только файлу, в котором она *"обитает"*. По сути, модификатор *static* позволяет переменным существовать так, что о них знают только функции, использующие их, тем самым "держа в узде" и ограничивая возможности негативных побочных эффектов. Переменные типа *static* позволяют программисту "скрывать" одни части своей программы от других частей. Это может оказаться просто супердостоинством, когда вам придется разрабатывать очень большую и сложную программу.

Важно! *Несмотря на то что глобальные static-переменные по-прежнему допустимы и широко используются в C++-коде, стандарт C++ возражает против их применения. Для управления доступом к глобальным переменным рекомендуется другой метод, который заключается в использовании пространств имен. Этот метод описан ниже в этой книге.*

Регистровые переменные

Возможно, чаще всего используется спецификатор класса памяти *register*. Для компилятора модификатор *register* означает предписание обеспечить такое хранение соответствующей переменной, чтобы доступ к ней можно было получить максимально быстро. Обычно переменная в этом случае будет храниться либо в регистре центрального процессора (ЦП), либо в кэше (быстродействующей буферной памяти небольшой емкости). Вероятно, вы знаете, что доступ к регистрам ЦП (или к кэш-памяти) принципиально быстрее, чем доступ к основной памяти компьютера. Таким образом, переменная, сохраняемая в регистре, будет обслужена гораздо быстрее, чем переменная, сохраняемая, например, в оперативной памяти (ОЗУ). Поскольку скорость, с которой к переменным можно получить доступ, определяет, по сути, скорость выполнения вашей программы, для получения удовлетворительных результатов программирования важно разумно использовать спецификатор *register*.

Спецификатор register в объявлении переменной означает требование оптимизировать код для получения максимально возможной скорости доступа к ней.

Формально спецификатор *register* представляет собой лишь запрос, который компилятор вправе проигнорировать. Это легко объяснить: ведь количество регистров (или устройств памяти с малым временем выборки) ограничено, причем для разных сред оно может быть различным. Поэтому, если компилятор исчерпает память быстрого доступа, он будет хранить *register*-переменные обычным способом. В общем случае неудовлетворенный *register*-запрос не приносит вреда, но, конечно же, и не дает никаких преимуществ хранения в регистровой памяти.

Поскольку в действительности только для ограниченного количества переменных можно обеспечить быстрый доступ, важно тщательно выбрать, к каким из них применить модификатор *register*. (Только правильный выбор может повысить быстродействие программы.) Как правило, чем чаще к переменной требуется доступ, тем большая выгода будет получена в результате оптимизации кода с помощью спецификатора *register*. Поэтому объявлять регистровыми имеет смысл управляющие переменные цикла или переменные, к которым выполняется доступ в теле цикла. На примере следующей функции показано, как *register*-переменная типа *int* используется для управления циклом. Эта функция вычисляет результат выражения m^e для целочисленных значений с сохранением знака исходного числа (т.е. при *m = -2* и *e = 2* результат будет равен *-4*).

```
int signed_pwr(register int m, register int e)
{
  register int temp;
  int sign;

  if(m < 0) sign = -1;
  else sign = 1;

  temp = 1;
  for( ; e; e--) temp = temp * m;

  return temp * sign;
}
```

В этом примере переменные *m*, *e* и *temp* объявлены как регистровые, поскольку все они используются в теле цикла, и потому к ним часто выполняется доступ. Однако переменная *sign* объявлена без спецификатора *register*, поскольку она не является частью цикла и используется реже.

Происхождение модификатора register

Модификатор *register* был впервые определен в языке C. Первоначально он применялся только к переменным типа int и char или к указателям и заставлял хранить переменные этого типа в регистре ЦП, а не в ОЗУ, где хранятся обычные переменные. Это означало, что операции с регистровыми переменными могли выполняться намного быстрее, чем операции с остальными (хранимыми в памяти), поскольку для опроса или модификации их значений не требовался доступ к памяти.

После стандартизации языка C было принято решение расширить определение спецификатора *register*. Согласно ANSI-стандарту C модификатор *register* можно применять к любому типу данных. Его использование стало означать для компилятора требование сделать доступ к переменной типа *register* максимально быстрым. Для ситуаций, включающих символы и целочисленные значения, это по-прежнему означает помещение их в регистры ЦП, поэтому традиционное определение все еще в силе. Поскольку язык C++ построен на ANSI-стандарте C, он также поддерживает расширенное определение спецификатора *register*.

Как упоминалось выше, точное количество *register*-переменных, которые реально будут оптимизированы в любой одной функции, определяется как типом процессора, так и

конкретной реализацией C++, которую вы используете. В общем случае можно рассчитывать по крайней мере на две. Однако не стоит беспокоиться о том, что вы могли объявить слишком много *register*-переменных, поскольку C++ автоматически превратит регистровые переменные в нерегистровые, когда их лимит будет исчерпан. (Это гарантирует переносимость C++-кода в рамках широкого диапазона процессоров.)

Чтобы показать влияние, оказываемое *register*-переменными на быстродействие программы, в следующем примере измеряется время выполнения двух циклов *for*, которые отличаются друг от друга только типом управляющих переменных. В программе используется стандартная библиотечная C++-функция *clock()*, которая возвращает количество импульсов сигнала времени системных часов, подсчитанных с начала выполнения этой программы. Программа должна включать заголовок *<ctime>*.

```
/* Эта программа демонстрирует влияние, которое может оказать
использование register-переменной на скорость выполнения программы.
*/
#include <iostream>
#include <ctime>
using namespace std;

unsigned int i; //не register-переменная
unsigned int delay;

int main()
{
  register unsigned int j;
  long start, end;
  start = clock();
  for(delay=0; delay<50; delay++)
    for(i=0; i<64000000; i++);
  end = clock();
  cout << "Количество тиков для не register-цикла: ";
  cout << end-start << ' \n';

  start = clock();
  for(delay=0; delay<50; delay++)
    for(j=0; j<64000000; j++);
  end = clock();
  cout << "Количество тиков для register-цикла: ";
  cout << end-start << '\n';

  return 0;
}
```

При выполнении этой программы вы убедитесь, что цикл с "регистровым" управлением выполняется приблизительно в два раза быстрее, чем цикл с "нерегистровым" управлением. Если вы не увидели ожидаемой разницы, это может означать, что ваш компилятор оптимизирует все переменные. Просто "поиграйте" программой до тех пор, пока разница не станет очевидной.

На заметку. *При написании этой книги была использована среда Visual C++, которая игнорирует ключевое слово register. Visual C++ применяет оптимизацию "как считает нужным". Поэтому вы можете не заметить влияния спецификатора register на выполнение*

нужны. Поэтому вы можете не заметить влияния спецификатора register на выполнение предыдущей программы. Однако ключевое слово register все еще принимается компилятором без сообщения об ошибке. Оно просто не оказывает никакого воздействия.

Перечисления

В C++ можно определить список именованных целочисленных констант. Такой список называется перечислением (enumeration). Эти константы можно затем использовать везде, где допустимы целочисленные значения (например, в целочисленных выражениях). Перечисления определяются с помощью ключевого слова *enum*, а формат их определения имеет такой вид:

```
enum type_name { список_перечисления } список_переменных;
```

Под элементом *список_перечисления* понимается список разделенных запятыми имен, которые представляют значения перечисления. Элемент *список_переменных* необязателен, поскольку переменные можно объявить позже, используя имя типа перечисления. В следующем примере определяется перечисление *apple* и две переменные типа *apple* с именами *red* и *yellow*.

```
enum apple {Jonathan, Golden_Del, Red_Del, Winesap, Cortland, McIntosh}
red, yellow;
```

Определив перечисление, можно объявить другие переменные этого типа, используя имя перечисления. Например, с помощью следующей инструкции объявляется одна переменная *fruit* перечисления *apple*.

```
apple fruit;
```

Эту инструкцию можно записать и так.

```
enum apple fruit;
```

Ключевое слово enum объявляет перечисление.

Однако использование ключевого слова *enum* здесь излишне. В языке C (который также поддерживает перечисления) обязательной была вторая форма, поэтому в некоторых программах вы можете встретить подобную запись.

С учетом предыдущих объявлений следующие типы инструкций совершенно допустимы.

```
fruit = Winesap;
```

```
if(fruit==Red_Del) cout << "Red Delicious\n";
```

Важно понимать, что каждый символ списка перечисления означает целое число, причем каждое следующее число (представленное идентификатором) на единицу больше предыдущего. По умолчанию значение первого символа перечисления равно нулю, следовательно, значение второго — единице и т.д. Поэтому при выполнении этой инструкции

```
cout << Jonathan << ' ' << Cortland;
```

на экран будут выведены числа *0 4*.

Несмотря на то что перечислимые константы автоматически преобразуются в целочисленные, обратное преобразование автоматически не выполняется. Например, следующая инструкция некорректна.

```
fruit =1; // ошибка
```

Эта инструкция вызовет во время компиляции ошибку, поскольку автоматического преобразования целочисленных значений в значения типа *apple* не существует. Откорректировать предыдущую инструкцию можно с помощью операции приведения типов.

```
fruit = (apple) 1; // Теперь все в порядке, но стиль не совершенен.
```

Теперь переменная *fruit* будет содержать значение *Golden_Del*, поскольку эта *apple*-константа связывается со значением *1*. Как отмечено в комментарии, несмотря на то, что эта

инструкция стала корректнои, ее стиль оставляет желать лучшего, что простительно лишь в особых обстоятельствах.

Используя инициализатор, можно указать значение одной или нескольких перечислимых констант. Это делается так: после соответствующего элемента списка перечисления ставится знак равенства и нужное целое число. При использовании инициализатора следующему (после инициализированного) элементу списка присваивается значение, на единицу превышающее предыдущее значение инициализатора. Например, при выполнении следующей инструкции константе *Winesap* присваивается значение *10*.

```
enum apple {Jonathan, Golden_Del, Red_Del, Winesap=10, Cortland, McIntosh};
```

Jonathan	0
Golden_Del	1
Red_Del	2
Winesap	10
Cortland	11
McIntosh	12

Часто в отношении перечислений ошибочно предполагается, что символы перечисления можно вводить и выводить как строки. Например, следующий фрагмент кода выполнен не будет.

```
// Слово "McIntosh" на экран таким образом не попадет.
fruit = McIntosh;
cout << fruit;
```

Не забывайте, что символ *McIntosh* — это просто имя для некоторого целочисленного значения, а не строка. Следовательно, при выполнении предыдущего кода на экране отобразится числовое значение константы *McIntosh*, а не строка *"McIntosh"*. Конечно, можно создать код ввода и вывода символов перечисления в виде строк, но он выходит несколько громоздким. Вот, например, как можно отобразить на экране названия сортов яблок, связанных с переменной *fruit*.

```
switch(fruit) {
  case Jonathan: cout << "Jonathan";
    break;
  case Golden_Del: cout << "Golden Delicious";
    break;
  case Red_Del: cout << "Red Delicious";
    break;
  case Winesap: cout << "Winesap";
    break;
  case Cortland: cout << "Cortland";
    break;
  case McIntosh: cout << "McIntosh";
    break;
}
```

Иногда для перевода значения перечисления в соответствующую строку можно объявить массив строк и использовать значение перечисления в качестве индекса. Например, следующая программа выводит названия трех сортов яблок.

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

enum apple {Jonathan, Golden_Del, Red_Del, Winesap, Cortland, McIntosh};

// Массив строк, связанных с перечислением apple.
char name[][20] = {
  "Jonathan",
  "Golden Delicious",
  "Red Delicious",
  "Winesap",
  "Cortland",
  "McIntosh",
};

int main( )
{
  apple fruit;

  fruit = Jonathan;
  cout << name[fruit] << '\n';

  fruit = Winesap;
  cout << name[fruit] << '\n';

  fruit = McIntosh;
  cout << name[fruit] << '\n';

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Jonathan
Winesap
McIntosh
```

Использованный в этой программе метод преобразования значения перечисления в строку можно применить к перечислению любого типа, если оно не содержит инициализаторов. Для надлежащего индексирования массива строк перечислимые константы должны начинаться с нуля, быть строго упорядоченными по возрастанию, и каждая следующая константа должна быть больше предыдущей точно на единицу.

Из-за того, что значения перечисления необходимо вручную преобразовывать в удобные для восприятия человеком строки, они, в основном, используются там, где такое преобразование не требуется. Для примера рассмотрите перечисление, используемое для определения таблицы символов компилятора.

Ключевое слово typedef

Ключевое слово typedef позволяет создать новое имя для существующего типа данных.

В C++ разрешается определять новые имена типов данных с помощью ключевого слова *typedef*. При использовании *typedef*-имени новый тип данных не создается, а лишь определяется новое имя для уже существующего типа. Благодаря *typedef*-именам можно сделать машинозависимые программы более переносимыми: для этого иногда достаточно

изменить *typedef*-инструкции. Это средство также позволяет улучшить читабельность кода, поскольку для стандартных типов данных с его помощью можно использовать описательные имена. Общий формат записи инструкции *typedef* таков,

```
typedef тип новое_имя;
```

Здесь элемент *mun* означает любой допустимый тип данных, а элемент *новое_имя* — новое имя для этого типа. При этом заметьте: новое имя определяется вами в качестве дополнения к существующему имени типа, а не для его замены.

Например, с помощью следующей инструкции можно создать новое имя для типа *float*,

```
typedef float balance;
```

Эта инструкция является предписанием компилятору распознавать идентификатор *balance* как еще одно имя для типа *float*. После этой инструкции можно создавать *float*-переменные с использованием имени *balance*.

```
balance over_due;
```

Здесь объявлена переменная с плавающей точкой *over_due* типа *balance*, который представляет собой стандартный тип *float*, но имеющий другое название.

Еще об операторах

Выше в этой книге вы уже познакомились с большинством операторов, которые не уникальны для C++. Но, в отличие от других языков программирования, в C++ предусмотрены и другие специальные операторы, которые значительно расширяют возможности языка и повышают его гибкость. Этим операторам и посвящена оставшаяся часть данной главы.

Поразрядные операторы

Поразрядные операторы обрабатывают отдельные биты.

Поскольку C++ нацелен на то, чтобы позволить полный доступ к аппаратным средствам компьютера, важно, чтобы он имел возможность непосредственно воздействовать на отдельные биты в рамках байта или машинного слова. Именно поэтому C++ и содержит поразрядные операторы. Поразрядные операторы предназначены для тестирования, установки или сдвига реальных битов в байтах или словах, которые соответствуют символьным или целочисленным C++-типам. Поразрядные операторы не используются для операндов типа *bool*, *float*, *double*, *long double*, *void* или других еще более сложных типов данных. Поразрядные операторы (они перечислены в табл. 9.1) очень часто используются для решения широкого круга задач программирования системного уровня, например, при опросе информации о состоянии устройства или ее формировании. Теперь рассмотрим каждый оператор этой группы в отдельности.

Таблица 9.1. Поразрядные операторы

Оператор	Значение
&	Поразрядное И (AND)
|	Поразрядное ИЛИ (OR)
^	Поразрядное исключающее ИЛИ (XOR)
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы *И*, *ИЛИ*, *исключающее ИЛИ* и *НЕ* (обозначаемые символами &, |, ^ и ~ соответственно) выполняют те же операции, что и их логические эквиваленты (т.е. они действуют согласно той же таблице истинности). Различие состоит лишь в том, что поразрядные операции работают на побитовой основе. В следующей таблице показан

результат выполнения каждой поразрядной операции для всех возможных сочетаний операндов (нулей и единиц).

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Как видно из таблицы, результат применения оператора *XOR* (исключающее ИЛИ) будет равен значению ИСТИНА (1) только в том случае, если истинен (равен значению 1) лишь один из операндов; в противном случае результат принимает значение ЛОЖЬ (0).

Поразрядный оператор *И* можно представить как способ подавления битовой информации. Это значит, что *0* в любом операнде, обеспечит установку в *0* соответствующего бита результата. Вот пример.

1101 0011

& *1010 1010*

1000 0010

Следующая программа считывает символы с клавиатуры и преобразует любой строчный символ в его прописной эквивалент путем установки шестого бита равным значению *0*. Набор символов *ASCII* определен так, что строчные буквы имеют почти такой же код, что и прописные, за исключением того, что код первых отличается от кода вторых ровно на *32*[только для латинского алфавита]. Следовательно, как показано в этой программе, чтобы из строчной буквы сделать прописную, достаточно обнулить ее шестой бит.

```
// Получение прописных букв.
#include <iostream>
using namespace std;

int main( )
{
  char ch;
  do {
    cin >> ch;

    // Эта инструкция обнуляет 6-й бит.
    ch = ch & 223; // В переменной ch теперь прописная буква.
    cout << ch;
  }while(ch! = 'Q');

  return 0;
}
```

Значение *223*, используемое в инструкции поразрядного *И*, является десятичным представлением двоичного числа *1101 1111*. Следовательно, эта операция *И* оставляет все биты в переменной *ch* нетронутыми, за исключением шестого (он сбрасывается в нуль).

Оператор *И* также полезно использовать, если нужно определить, установлен ли интересующий вас бит (т.е. равен ли он значению *1*) или нет. Например, при выполнении следующей инструкции вы узнаете, установлен ли *4-й* бит в переменной *status*,

```
if(status & 8) cout << "Бит 4 установлен";
```

```
if(status & 8) cout << "Бит 4 установлен";
```

Чтобы понять, почему для тестирования четвертого бита используется число *8*, вспомните, что в двоичной системе счисления число *8* представляется как *0000 1000*, т.е. в числе *8* установлен только четвертый разряд. Поэтому условное выражение инструкции *if* даст значение ИСТИНА только в том случае, если четвертый бит переменной *status* также установлен (равен 1). Интересное использование этого метода показано на примере функции *disp_binary()*. Она отображает в двоичном формате конфигурацию битов своего аргумента. Мы будем использовать функцию *disp_binary()* ниже в этой главе для исследования возможностей других поразрядных операций.

```
// Отображение конфигурации битов в байте.
void disp_binary(unsigned u)
{
  register int t;
  for(t=128; t>0; t=t/2)
    if(u & t) cout << "1";
    else cout << "0 ";
  cout << "\n";
}
```

Функция *disp_binary()*, используя поразрядный оператор *И*, последовательно тестирует каждый бит младшего байта переменной *u*, чтобы определить, установлен он или сброшен. Если он установлен, отображается цифра *1*, в противном случае — цифра *0*. Интереса ради попробуйте расширить эту функцию так, чтобы она отображала все биты переменной *u*, а не только ее младший байт.

Поразрядный оператор *ИЛИ*, в противоположность поразрядному *И*, удобно использовать для установки нужных битов в единицу. При выполнении операции *ИЛИ* наличие в любом операнде бита, равного *1*, означает, что в результате соответствующий бит также будет равен единице. Вот пример.

1101 0011

| *1010 1010*

1111 1011

Можно использовать оператор *ИЛИ* для превращения рассмотренной выше программы (которая преобразует строчные символы в их прописные эквиваленты) в ее "противоположность", т.е. теперь, как показано ниже, она будет преобразовывать прописные буквы в строчные.

```
// Получение строчных букв.
#include <iostream>
using namespace std;

int main()
{
  char ch;
  do {
    cin >> ch;
    /* Эта инструкция делает букву строчной, устанавливая ее 6-й бит.*/
    ch = ch | 32;
    cout << ch;
  }while(ch != 'q');
```

```
  return 0;
}
```

Установка шестого бита превращает прописную букву в ее строчный эквивалент. Поразрядное исключающее *ИЛИ* (XOR) устанавливает в единицу бит результата только в том случае, если соответствующие биты операндов отличаются один от другого, т.е. не равны. Вот пример:

0111 1111

^ *1011 1001*

1100 0110

Унарный оператор *НЕ* (или оператор дополнения до 1) инвертирует состояние всех битов своего операнда. Например, если целочисленное значение (хранимое в переменной *A*), представляет собой двоичный код *1001 0110*, то в результате операции *~A* получим двоичный код *0110 1001*.

В следующей программе демонстрируется использование оператора *НЕ* посредством отображения некоторого числа и его дополнения до *1* в двоичном коде с помощью приведенной выше функции *disp_binary()*.

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

void disp_binary(unsigned u);
int main( )
{
  unsigned u;
  cout << "Введите число между 0 и 255: ";
    cin >> u;

  cout << "Исходное число в двоичном коде: ";
  disp_binary(u);

  cout << "Его дополнение до единицы: ";
  disp_binary(~u);

  return 0;
}

// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
  register int t;
  for(t=128; t>0; t=t/2)
    if(u & t) cout << "1";
    else cout << "0";
  cout << "\n";
}
```

Вот как выглядят результаты выполнения этой программы.

```
Введите число между 0 и 255: 99
```

```
Исходное число в двоичном коде: 01100011
Его дополнение до единицы: 10011100
```

И еще. Не путайте логические и поразрядные операторы. Они выполняют различные действия. Операторы &, | и ~ применяются непосредственно к каждому биту значения в отдельности. Эквивалентные логические операторы обрабатывают в качестве операндов значения ИСТИНА/ЛОЖЬ (не нуль/нуль). Поэтому поразрядные операторы нельзя использовать вместо их логических эквивалентов в условных выражениях. Например, если значение *x* равно *7*, то выражение *x && 8* имеет значение ИСТИНА, в то время как выражение *x & 8* дает значение ЛОЖЬ.

Узелок на память. *Оператор отношения или логический оператор всегда генерирует результат, который имеет значение ИСТИНА или ЛОЖЬ, в то время как аналогичный поразрядный оператор генерирует значение, получаемое согласно таблице истинности конкретной операции.*

Операторы сдвига

Операторы сдвига, *">>"* и *"<<"* сдвигают все биты в значении вправо или влево.

Общий формат использования оператора сдвига вправо выглядит так.

```
значение >> число_битов
```

А оператор сдвига влево используется так.

```
значение << число_битов
```

Операторы сдвига предназначены для сдвига битов в рамках целочисленного значения.

Здесь элемент *число_битов* указывает, на сколько позиций должно быть сдвинуто *значение*. При каждом сдвиге влево все биты, составляющее значение, сдвигаются влево на одну позицию, а в младший разряд записывается нуль. При каждом сдвиге вправо все биты сдвигаются, соответственно, вправо. Если сдвигу вправо подвергается значение без знака, в старший разряд записывается нуль. Если же сдвигу вправо подвергается значение со знаком, значение знакового разряда сохраняется. Как вы помните, отрицательные целые числа представляются установкой старшего разряда числа равным единице. Таким образом, если сдвигаемое значение отрицательно, при каждом сдвиге вправо в старший разряд записывается единица, а если положительно — нуль. Не забывайте, сдвиг, выполняемый операторами сдвига, не является циклическим, т.е. при сдвиге как вправо, так и влево крайние биты теряются, и содержимое потерянного бита узнать невозможно.

Операторы сдвига работают только со значениями целочисленных типов, например, *символами*, *целыми числами* и *длинными целыми числами*. Они не применимы к значениям с плавающей точкой.

Побитовые операции сдвига могут оказаться весьма полезными для декодирования входной информации, получаемой от внешних устройств (например, цифроаналоговых преобразователей), и обработки информация о состоянии устройств. Поразрядные операторы сдвига можно также использовать для выполнения ускоренных операций умножения и деления целых чисел. С помощью сдвига влево можно эффективно умножать на два, сдвиг вправо позволяет не менее эффективно делить на два.

Следующая программа наглядно иллюстрирует результат использования операторов сдвига.

```
// Демонстрация выполнения поразрядного сдвига.
#include <iostream>
using namespace std;

void disp_binary(unsigned u);

int main( )
```

```
{
  int i=1, t;
  for(t=0; t<8; t++) {
    disp_binary(i);
    i = i << 1;
  }
  cout << "\n";
  for(t=0; t<8; t++) {
    i = i >> 1;
    disp_binary(i);
  }
  return 0;
}

// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
  register int t;
  for(t=128; t>0; t=t/2)
    if(u & t) cout << "1";
    else cout << "0 ";
  cout << "\n";
}
```

Результаты выполнения этой программы таковы.

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Оператор "знак вопроса"

Одним из самых замечательных операторов C++ является оператор *"?"*. Оператор *"?"* можно использовать в качестве замены *if*-else-инструкций, употребляемых в следующем общем формате.

```
if(условие)
  переменная = выражение 1;
else
  переменная = выражение 2;
```

Здесь значение, присваиваемое переменной, зависит от результата вычисления элемента *условие*, управляющего инструкцией *if*.

Оператор *"?"* называется тернарным, поскольку он работает с тремя операндами. Вот его общий формат записи:

```
Выражение1? Выражение2 : Выражение3;
```

Все элементы здесь являются выражениями. Обратите внимание на использование и расположение двоеточия.

Значение *?*-выражения определяется следующим образом. Вычисляется *Выражение1*. Если оно оказывается истинным, вычисляется *Выражение2*, и результат его вычисления становится значением всего *?*-выражения. Если результат вычисления элемента *Выражение1* оказывается ложным, значением всего *?*-выражения становится результат вычисления элемента *Выражение3*. Рассмотрим следующий пример.

```
while(something) {
  x = count > 0 ? 0 : 1;
  // ...
}
```

Здесь переменной *x* будет присваиваться значение *0* до тех пор, пока значение переменной *count* не станет меньше или равно нулю. Аналогичный код (но с использованием if-else-инструкции) выглядел бы так.

```
while(something) {
  if(count >0) x = 0;
  else x = 1;
  // ...
}
```

А вот еще один пример практического применения оператора *?*. Следующая программа делит два числа, но не допускает деления на нуль.

```
/* Эта программа использует оператор ? для предотвращения деления на
нуль.
*/
#include <iostream>
using namespace std;

int div_zero();

int main()
{
  int i, j, result;
  cout << "Введите делимое и делитель: ";
```

```
    cin >> i >> j;
  // Эта инструкция не допустит возникновения ошибки деления на нуль.
  result = j ? i/j : div_zero();
  cout << "Результат: " << result;
  return 0;
}
int div_zero()
{
  cout << "Нельзя делить на нуль. \n";
  return 0;
}
```

Здесь, если значение переменной *j* не равно нулю, выполняется деление значения переменной *i* на значение переменной *j*, а результат присваивается переменной *result*. В противном случае вызывается обработчик ошибки деления на нуль *div_zero()*, и переменной *result* присваивается нулевое значение.

Составные операторы присваивания

В C++ предусмотрены специальные составные операторы присваивания, в которых объединено присваивание с еще одной операцией. Начнем с примера и рассмотрим следующую инструкцию.

```
x = x + 10;
```

Используя составной оператор присваивания, ее можно переписать в таком виде.

```
x += 10;
```

Пара операторов *+=* служит указанием компилятору присвоить переменной *x* сумму текущего значения переменной *x* и числа *10*. Этот пример служит иллюстрацией того, что составные операторы присваивания упрощают программирование определенных инструкций присваивания. Кроме того, они позволяют компилятору сгенерировать более эффективный код.

Составные версии операторов присваивания существуют для всех бинарных операторов (т.е. для всех операторов, которые работают с двумя операндами). Таким образом, при таком общем формате бинарных операторов присваивания

```
переменная = переменная op выражение;
```

общая форма записи их составных версий выглядит так:

```
переменная op = выражение;
```

Здесь элемент *op* означает конкретный арифметический или логический оператор, объединяемый с оператором присваивания.

А вот еще один пример. Инструкция

```
x = x - 100;
```

аналогична такой:

```
x -= 100;
```

Обе эти инструкции присваивают переменной *x* ее прежнее значение, уменьшенное на *100*.

Составные операторы присваивания можно часто встретить в профессионально написанных C++-программах, поэтому каждый C++-программист должен быть с ними на "ты" .

Оператор "запятая"

Не менее интересным, чем описанные выше операторы, является такой оператор C++, как

"запятая". Вы уже видели несколько примеров его использования в цикле *for*, где с его помощью была организована инициализация сразу нескольких переменных. Но оператор *"запятая"* также может составлять часть выражения. Его назначение в этом случае — связать определенным образом несколько выражений. Значение списка выражений, разделенных запятыми, определяется в этом случае значением крайнего справа выражения. Значения других выражений отбрасываются. Следовательно, значение выражения справа становится значением всего выражения-списка. Например, при выполнении этой инструкции

```
var = (count=19, incr=10, count+1);
```

переменной *count* сначала присваивается число *19*, переменной *incr* — число *10*, а затем к значению переменной *count* прибавляется единица, после чего переменной *var* присваивается значение крайнего справа выражения, т.е. *count+1*, которое равно *20*. Круглые скобки здесь обязательны, поскольку оператор *"запятая"* имеет более низкий приоритет, чем оператор присваивания.

Чтобы понять назначение оператора *"запятая"*, попробуем выполнить следующую программу.

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

int main( )
{
  int i, j;
  j = 10;
  i = (j++, j+100, 999+j);
  cout << i;
  return 0;
}
```

Эта программа выводит на экран число *1010*. И вот почему: сначала переменной *j* присваивается число *10*, затем переменная *j* инкрементируется до *11*. После этого вычисляется выражение *j+100*, которое нигде не применяется. Наконец, выполняется сложение значения переменной *j* (оно по-прежнему равно 11) с числом *999*, что в результате дает число *1010*.

По сути, назначение оператора *"запятая"* — обеспечить выполнение заданной последовательности операций. Если эта последовательность используется в правой части инструкции присваивания, то переменной, указанной в ее левой части, присваивается значение последнего выражения из списка выражений, разделенных запятыми. Оператор "запятая" по его функциональной нагрузке можно сравнить со словом *"и"*, используемым в фразе: *"сделай это, и то, и другое..."*.

Несколько присваиваний "в одном"

Язык C++ позволяет применить очень удобный метод одновременного присваивания многим переменным одного и того же значения. Речь идет об объединении сразу нескольких присваиваний в одной инструкции. Например, при выполнении этой инструкции переменным *count*, *incr* и *index* будет присвоено число *10*.

```
count = incr = index = 10;
```

Этот формат присвоения нескольким переменным общего значения можно часто встретить в профессионально написанных программах.

Использование ключевого слова sizeof

Иногда полезно знать размер (в байтах) одного из типов данных. Поскольку размеры встроенных C++-типов данных в разных вычислительных средах могут быть различными, а

знание размера переменной во всех ситуациях имеет важное значение, то для решения этой проблемы в C++ включен оператор (действующий во время компиляции программы), который используется в двух следующих форматах.

```
sizeof (type)
sizeof value
```

Оператор sizeof во время компиляции программы получает размер типа или значения.

Первая версия возвращает размер заданного *типа данных*, а вторая — размер заданного *значения*. Если вам нужно узнать размер некоторого типа данных (например, *int*), заключите название этого типа в круглые скобки. Если же вас интересует размер области памяти, занимаемой конкретным значением, можно обойтись без круглых скобок, хотя при желании их можно использовать.

Чтобы понять, как работает оператор *sizeof*, испытайте следующую короткую программу. Для многих 32-разрядных сред она должна отобразить значения *1*, *4*, *4* и *8*.

```
// Демонстрация использования оператора sizeof.
#include <iostream>
using namespace std;

int main( )
{
  char ch;
  int i;

  cout << sizeof ch << ' '; // размер типа char
  cout << sizeof i << ' '; // размер типа int
  cout << sizeof (float) << ' '; // размер типа float
  cout << sizeof (double) << ' '; // размер типа double

  return 0;
}
```

Как упоминалось выше, оператор *sizeof* действует во время компиляции программы. Вся информация, необходимая для вычисления размера указанной переменной или заданного типа данных, известна уже во время компиляции.

Оператор *sizeof* можно применить к любому типу данных. Например, в случае применения к массиву он возвращает количество байтов, занимаемых массивом. Рассмотрим следующий фрагмент кода.

```
int nums[4];
cout << sizeof nums; // Будет выведено число 16.
```

Для *4*-байтных значений типа *int* при выполнении этого фрагмента кода на экране отобразится число *16* (которое получается в результате умножения 4 байт на 4 элемента массива).

Оператор *sizeof* главным образом используется при написании кода, который зависит от размера C++-типов данных. Помните: поскольку размеры типов данных в C++ определяются конкретной реализацией, не стоит полагаться на размеры типов, определенные в реализации, в которой вы работаете в данный момент.

Динамическое распределение памяти с использованием операторов new и delete

Для C++-программы существует два основных способа хранения информации в основной памяти компьютера. Первый состоит в использовании переменных. Область памяти,

предоставляемая переменным, закрепляется за ними во время компиляции и не может быть изменена при выполнении программы. Второй способ заключается в использовании C++-системы динамического распределения памяти. В этом случае память для данных выделяется по мере необходимости из раздела свободной памяти, который расположен между вашей программой (и ее постоянной областью хранения) и стеком. Этот раздел называется *"кучей"* (heap). (Расположение программы в памяти схематично показано на рис. 9.2.)

Система динамического распределения памяти — это средство получения программой некоторой области памяти во время ее выполнения.



Рис. 9.2. Использование памяти в C++-программе

Динамическое выделение памяти — это получение программой памяти во время ее выполнения. Другими словами, благодаря этой системе программа может создавать переменные во время выполнения, причем в нужном (в зависимости от ситуации) количестве. Эта система динамического распределения памяти особенно ценна для таких структур данных, как связные списки и двоичные деревья, которые изменяют свой размер по мере их использования. Динамическое выделение памяти для тех или иных целей — важная составляющая почти всех реальных программ.

Чтобы удовлетворить запрос на динамическое выделение памяти, используется так называемая *"куча"*. Нетрудно предположить, что в некоторых чрезвычайных ситуациях свободная память *"кучи"* может исчерпаться. Следовательно, несмотря на то, что динамическое распределение памяти (по сравнению с фиксированным) обеспечивает большую гибкость, но и в этом случае оно имеет свои пределы.

Оператор new позволяет динамически выделить область памяти.

Язык C++ содержит два оператора, *new* и *delete*, которые выполняют функции по выделению и освобождению памяти. Приводим их общий формат.

```
переменная-указатель = new тип_переменной;
delete переменная-указатель;
```

Оператор delete освобождает ранее выделенную динамическую память.

Здесь элемент *переменная-указатель* представляет собой указатель на значение, тип которого задан элементом *тип_переменной*. Оператор *new* выделяет область памяти, достаточную для хранения значения заданного типа, и возвращает указатель на эту область памяти. С помощью оператора *new* можно выделить память для значений любого допустимого типа. Оператор *delete* освобождает область памяти, адресуемую заданным указателем. После освобождения эта память может быть снова выделена в других целях при последующем *new*-запросе на выделение памяти.

Поскольку объем *"кучи"* конечен, она может когда-нибудь исчерпаться. Если для удовлетворения очередного запроса на выделение памяти не существует достаточно свободной памяти, оператор *new* потерпит фиаско, и будет сгенерировано исключение. *Исключение*— это ошибка специального типа, которая возникает во время выполнения программы (в C++ предусмотрена целая подсистема, предназначенная для обработки таких ошибок). (Исключения описаны в главе 17.) В общем случае ваша программа должна обработать подобное исключение и по возможности выполнить действие, соответствующее конкретной ситуации. Если это исключение не будет обработано вашей программой, ее выполнение будет прекращено.

Такое поведение оператора *new* в случае невозможности удовлетворить запрос на выделение памяти определено стандартом C++. На такую реализацию настроены также все современные компиляторы, включая последние версии *Visual C++* и *C++ Builder*. Однако дело в том, что некоторые более ранние компиляторы обрабатывают *new*-инструкции по-другому. Сразу после изобретения языка C++ оператор *new* при неудачном выполнении возвращал нулевой указатель. Позже его реализация была изменена так, чтобы в случае неудачи генерировалось исключение, как было описано выше. Поскольку в этой книге мы придерживаемся стандарта C++, то во всех представленных здесь примерах предполагается именно генерирование исключения. Если же вы используете более старый компилятор, обратитесь к прилагаемой к нему документации и уточните, как реализован оператор *new* (при необходимости внесите в примеры соответствующие изменения).

Поскольку исключения рассматриваются ниже в этой книге (после темы классов и объектов), мы не будем пока отвлекаться на обработку исключений, генерируемых в случае неудачного выполнения оператора *new*. Кроме того, ни один из примеров в этой и последующих главах не должен вызвать неудачного выполнения оператора *new*, поскольку в этих программах запрашивается лишь несколько байтов. Но если такая ситуация все же возникнет, то в худшем случае это приведет к завершению программы. В главе 17, посвященной обработке исключений, вы узнаете, как обработать исключение, сгенерированное оператором *new*.

Рассмотрим пример программы, которая иллюстрирует использование операторов *new* и *delete*.

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

int main( )
{
  int *p;
  p = new int; // Выделяем память для int-значения.
  *p = 20; // Помещаем в эту область памяти значение 20.
  cout << *p; // Убеждаемся (путем вывода на экран) в работоспособности
```

```
  cout << *p; // Убеждаемся (путем вывода на экран) в работоспособности
этого кода.
  delete p; // Освобождаем память.

  return 0;
}
```

Эта программа присваивает указателю p адрес (взятой из *"кучи"*) области памяти, которая будет иметь размер, достаточный для хранения целочисленного значения. Затем в эту область памяти помещается число *20*, после чего на экране отображается ее содержимое. Наконец, динамически выделенная память освобождается.

Благодаря такому способу организации динамического выделения памяти оператор *delete* необходимо использовать только с тем указателем на память, который был возвращен в результате *new*-запроса на выделение памяти. Использование оператора *delete* с другим типом адреса может вызвать серьезные проблемы.

Инициализация динамически выделенной памяти

Используя оператор *new*, динамически выделяемую память можно инициализировать. Для этого после имени типа задайте начальное значение, заключив его в круглые скобки. Например, в следующей программе область памяти, адресуемая указателем *p*, инициализируется значением *99*.

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

int main( )
{
  int *p;
  p = new int (99); // Инициализируем память числом 99.
  cout << *p; // На экран выводится число 99.
  delete p;
  return 0;
}
```

Выделение памяти для массивов

С помощью оператора *new* можно выделять память и для массивов. Вот как выглядит общий формат операции выделения памяти для одномерного массива:

```
переменная-указатель = new тип [размер];
```

Здесь элемент *размер* задает количество элементов в массиве.

Чтобы освободить память, выделенную для динамически созданного массива, используйте такой формат оператора *delete*:

```
delete [] переменная-указатель;
```

Здесь элемент *переменная-указатель* представляет собой адрес, полученный при выделении памяти для массива (с помощью оператора *new*). Квадратные скобки означают для C++, что динамически созданный массив удаляется, а вся область памяти, выделенная для него, автоматически освобождается.

Важно! *Более старые C++-компиляторы могут требовать задания размера удаляемого массива, поскольку в ранних версиях C++ для освобождения памяти, занимаемой удаляемым массивом, необходимо было применять такой формат оператора delete:*

```
delete [размер] переменная-указатель;
```

Здесь элемент размер задает количество элементов в массиве. Стандарт C++ больше не

требует указывать размер при его удалении.

При выполнении следующей программы выделяется память для *10*-элементного массива типа *double*, который затем заполняется значениями от *100* до *109*, после чего содержимое этого массива отображается на экране.

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

int main( )
{
  double *p;
  int i;
  p = new double [10]; // Выделяем память для 10-элементного массива.

  //Заполняем массив значениями от 100 до 109.
  for(i=0; i<10; i++) p[i] = 100.00 + i;

  // Отображаем содержимое массива.
  for(i=0; i<10; i++) cout << p[i] << " ";
  delete [] p; // Удаляем весь массив.
  return 0;
}
```

При динамическом выделении памяти для массива важно помнить, что его нельзя одновременно и инициализировать.

Динамическое распределение памяти в языке С: функции malloc() и free()

Язык С не содержит операторов *new* или *delete*. Вместо них в С используются библиотечные функции, предназначенные для выделения и освобождения памяти. В целях совместимости С++ по-прежнему поддерживает С-систему динамического распределения памяти и не зря: в С++-программах все еще используются С-ориентированные средства динамического распределения памяти. Поэтому им стоит уделить внимание.

Ядро С-системы распределения памяти составляют функции *malloc()* и *free()*. Функция *malloc()* предназначена для выделения памяти, а функция *free()* — для ее освобождения. Другими словами, каждый раз, когда с помощью функции *malloc()* делается запрос, часть свободной памяти выделяется в соответствии с этим запросом. При каждом вызове функции *free()* соответствующая область памяти возвращается системе. Любая программа, которая использует эти функции, должна включать заголовок *<cstdlib>*.

Функция *malloc()* имеет такой прототип,

```
void *malloc(size_t num_bytes);
```

Здесь *num_bytes* означает количество байтов запрашиваемой памяти. (Тип *size_t* представляет собой разновидность целочисленного типа без знака.) Функция *malloc()* возвращает указатель типа *void*, который играет роль обобщенного указателя. Чтобы из этого обобщенного указателя получить указатель на нужный вам тип, необходимо использовать операцию приведения типов. В результате успешного вызова функция *malloc()* возвратит указатель на первый байт области памяти, выделенной из *"кучи"*. Если для удовлетворения запроса свободной памяти в системе недостаточно, функция *malloc()* возвращает нулевой указатель.

Функция *free()* выполняет действие, обратное действию функции *malloc()* в том, что она возвращает системе ранее выделенную ею память. После освобождения память можно снова использовать последующим обращением к функции *malloc()*. Функция *free()* имеет

такои прототип.

```
void free(void *ptr);
```

Здесь параметр *ptr* представляет собой указатель на память, ранее выделенную с помощью функции *malloc()*. Никогда не следует вызывать функцию *free()* с недействительным аргументом; это может привести к разрушению списка областей памяти, подлежащих освобождению.

Использование функций *malloc()* и *free()* иллюстрируется в следующей программе.

```
// Демонстрация использования функций malloc( ) и free( ).
#include <iostream>
#include <cstdlib>
using namespace std;

int main( )
{
  int *i;
  double *j;

  i = (int *) malloc(sizeof(int));
  if(!i) {
    cout << "Выделить память не удалось.\n";
    return 1;
  }

  j = (double *) malloc(sizeof(double));
  if(! j ) {
    cout << "Выделить память не удалось.\n";
    return 1;
  }

  *i = 10;
  *j = 100.123;
  cout << *i << ' ' << *j;

  // Освобождение памяти.
  free (i);
  free (j);

  return 0;
}
```

Несмотря на то что функции *malloc()* и *f₹ee()* — полностью пригодны для динамического распределения памяти, есть ряд причин, по которым в C++ определены собственные средства динамического распределения памяти. Во-первых, оператор *new* автоматически вычисляет размер выделяемой области памяти для заданного типа, т.е. вам не нужно использовать оператор *sizeof*, а значит, налицо экономия в коде и трудовых затратах программиста. Но важнее то, что автоматическое вычисление не допускает выделения неправильного объема памяти. Во-вторых, C++-оператор *new* автоматически возвращает корректный тип указателя, что освобождает программиста от необходимости использовать операцию приведения типов. В-третьих, используя оператор *new*, можно инициализировать объект, для которого выделяется память. Наконец, как будет показано ниже в этой книге,

программист может создать собственные версии операторов *new* и *delete*.

И последнее. Из-за возможной несовместимости не следует смешивать функции *malloc()* и *free()* с операторами *new* и *delete* в одной программе.

Сводная таблица приоритетов C++-операторов

В табл. 9.2 показан приоритет выполнения всех C++-операторов (от высшего до самого низкого). Большинство операторов ассоциированы слева направо. Но *унарные операторы*, *операторы присваивания* и оператор *"?"* ассоциированы справа налево. Обратите внимание на то, что эта таблица включает несколько операторов, которые мы пока не использовали в наших примерах, поскольку они относятся к объектно-ориентированному программированию (и описаны ниже).

Таблица 9.2. Приоритет C++-операторов

Наивысший	`() [] -> :: .`
	`! ~ ++ -- - * & sizeof new delete typeid` операторы приведения типа
	`.* ->*`
	`* / %`
	`+ -`
	`<< >>`
	`< <= > >=`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	`?:`
	`= += -= *= /= %= >>= <<= &= ^= |=`
Низший	`,`

Глава 10: Структуры и объединения

В языке C++ определено несколько составных типов данных, т.е. типов, которые состоят из двух или более элементов. С одним из составных типов — *массивом* — вы уже знакомы. С двумя другими — *структурами* и *объединениями* — вы познакомитесь в этой главе, а знакомство с еще одним типом — *классом* — мы отложим до главы 11. И хотя структура и объединение предназначены для удовлетворения различных потребностей программистов, оба они представляют собой удобные средства управления группами связанных переменных. При этом важно понимать, что создание структуры (или объединения) означает создание нового *определенного программистом* типа данных. Сама возможность создания собственных типов данных является признаком могущества языка C++.

В C++ структуры и объединения имеют как *объектно-ориентированные*, так и *не объектно-ориентированные* атрибуты. В этой главе рассматриваются только последние. Об объектно-ориентированных и их свойствах речь пойдет в следующей главе после введения понятия о классах и объектах.

Структуры

Структура — *это группа связанных переменных*.

В C++ *структура* представляет собой коллекцию объединенных общим именем переменных, которая обеспечивает удобное средство хранения родственных данных в одном месте. Структуры — это *совокупные типы данных*, поскольку они состоят из нескольких различных, но логически связанных переменных. По тем же причинам их иногда называют *составными* или *конгломератными типами данных*.

Прежде чем будет создан объект структуры, должен быть определен ее формат. Это делается посредством объявления структуры. Объявление структуры позволяет понять, переменные какого типа она содержит. Переменные, составляющие структуру, называются ее *членами*. Члены структуры также называют элементами или *полями*.

Член структуры — *это переменная, которая является частью структуры*.

В общем случае все члены структуры должны быть логически связаны друг с другом. Например, структуры обычно используются для хранения такой информации, как почтовые адреса, таблицы имен компилятора, элементы карточного каталога и т.п. Безусловно, отношения между членами структуры совершенно субъективны и определяются программистом. Компилятор *"ничего о них не знает"* (или *"не хочет знать"*).

Начнем рассмотрение структуры с примера. Определим структуру, которая может содержать информацию о товарах, хранимых на складе компании. Одна запись инвентарной ведомости обычно состоит из нескольких данных, например: наименование товара, стоимость и количество, имеющееся в наличии. Поэтому для управления подобной информацией удобно использовать именно структуру. В следующем фрагменте кода объявляется структура, которая определяет следующие элементы: *наименование товара*, *стоимость*, *розничная цена*, *имеющееся в наличии количество* и *время до пополнения запасов*. Этих данных часто вполне достаточно для управления складским хозяйством. О начале объявления структуры компилятору сообщает ключевое слово *struct*.

```
struct inv_type {
  char item[40]; // наименование товара
  double cost; // стоимость
  double retail; // розничная цена
  int on_hand; // имеющееся в наличии количество
  int lead_time; // число дней до пополнения запасов
};
```

Имя структуры — *это ее спецификатор типа*.

Обратите внимание на то, что объявление завершается точкой с запятой. Дело в том, что объявление структуры представляет собой инструкцию. Именем типа структуры здесь является *inv_type*. Другими словами, имя *inv_type* идентифицирует конкретную структуру данных и является ее спецификатором типа.

В предыдущем объявлении в действительности не было создано ни одной переменной. Был определен лишь формат данных. Чтобы с помощью этой структуры объявить реальную переменную (т.е. физический объект), нужно записать инструкцию, подобную следующей.

```
inv_type inv_var;
```

Вот теперь объявляется структурная переменная типа *inv_type* с именем *inv_var*. Помните: определяя структуру, вы определяете новый тип данных, но он не будет реализован до тех пор, пока вы не объявите переменную того типа, который уже реально существует.

При объявлении структурной переменной C++ автоматически выделит объем памяти, достаточный для хранения всех членов структуры. На рис. 10.1 показано, как переменная *inv_var* будет размещена в памяти компьютера (в предположении, что double-значение занимает 8 байт, а int-значение — 4).

Одновременно с определением структуры можно объявить одну или несколько переменных, как показано в этом примере.

```
struct inv_type {
  char item[40]; // наименование товара
  double cost; // стоимость
  double retail; // розничная цена
  int on_hand; // имеющееся в наличии количество
  int lead_time; // число дней до пополнения запасов
} inv_varA, inv_varB, inv_varC;
```

Этот фрагмент кода определяет структурный тип *inv_type* и объявляет переменные *inv_varA*, *inv_varB* и *inv_varC* этого типа. Важно понимать, что каждая структурная переменная содержит собственные копии членов структуры. Например, поле *cost* структуры *inv_varA* изолировано от поля *cost* структуры *inv_varB*. Следовательно, изменения, вносимые в одно поле, никак не влияют на содержимое другого поля.

Если для программы достаточно только одной структурной переменной, в ее определение необязательно включать имя структурного типа. Рассмотрим следующий пример:

```
struct {
  char item[40]; // наименование товара
  double cost; // стоимость
  double retail; // розничная цена
  int on_hand; // имеющееся в наличии количество
  int lead_time; // число дней до пополнения запасов
} temp;
```

Этот фрагмент кода объявляет одну переменную *temp* в соответствии с предваряющим ее определением структуры.

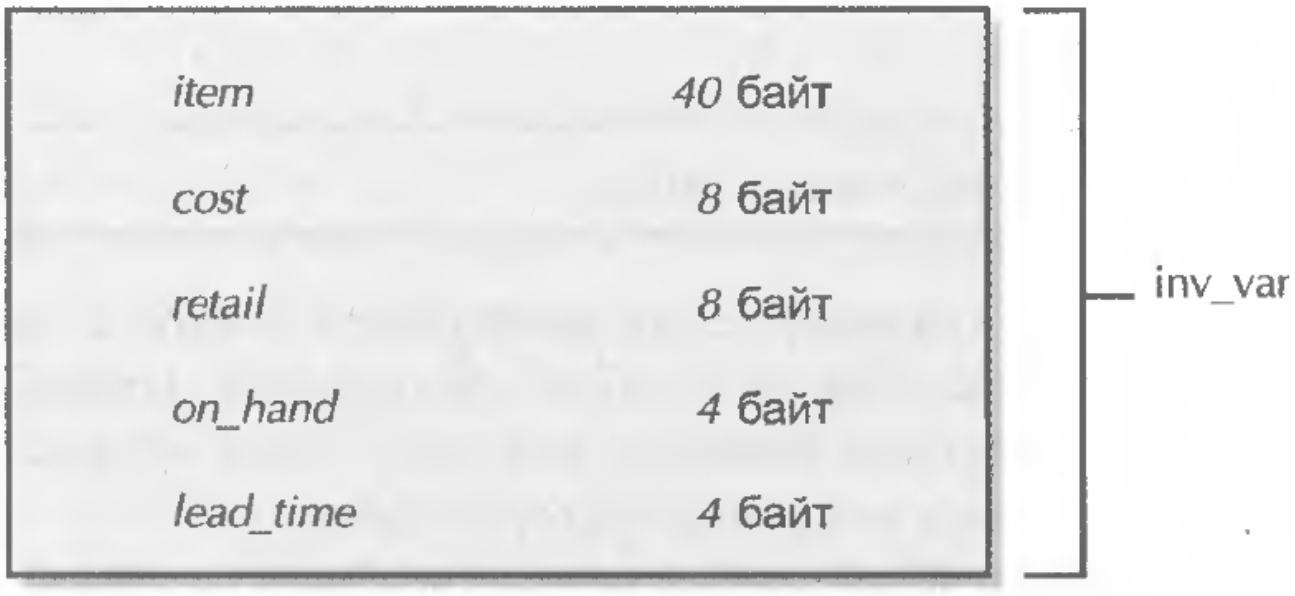


Рис. 10.1. Размещение структурной переменной `inv_var` в памяти компьютера

Ключевое слово struct означает начало объявления структуры.

Общий формат объявления структуры выглядит так.

```
struct имя_типа_структуры {
  тип имя_элемента1;
```

```
  тип имя_элемента2;
  тип имя_элемента3;
  .
  .
  .
  тип имя_элементаN;
} структурные_переменные;
```

Доступ к членам структуры

К отдельным членам структуры доступ осуществляется с помощью оператора *"точка"*. Например, при выполнении следующего кода значение *10.39* будет присвоено полю *cost* структурной переменной *inv_var*, которая была объявлена выше.

```
inv_var.cost = 10.39;
```

Чтобы обратиться к члену структуры, нужно перед его именем поставить имя структурной переменной и оператор *"точка"*. Так осуществляется доступ ко всем элементам структуры. Общий формат доступа записывается так.

```
имя_структурной_переменной.имя_члена
```

Оператор "точка" (.) позволяет получить доступ к любому члену любой структуры.

Следовательно, чтобы вывести значение поля *cost* на экран, необходимо выполнить следующую инструкцию.

```
cout << inv_var.cost;
```

Аналогичным способом можно использовать символьный массив *inv_var.item* в вызове функции *gets()*.

```
gets(inv_var.item);
```

Здесь функции *gets()* будет передан символьный указатель на начало области памяти, отведенной элементу *item*.

Если вам нужно получить доступ к отдельным элементам массива *inv_var.item*, используйте индексацию. Например, с помощью этого кода можно посимвольно вывести на экран содержимое массива *inv_var.item*.

```
int t;
for(t=0; inv_var.itern[t]; t++)
  cout << inv_var.item[t];
```

Массивы структур

Структуры могут быть элементами массивов. И в самом деле, массивы структур используются довольно часто. Чтобы объявить массив структур, необходимо сначала определить структуру, а затем объявить массив элементов этого структурного типа. Например, чтобы объявить *100*-элементный массив структур типа *inv_type* (который определен выше), достаточно записать следующее.

```
inv_type invtry[100];
```

Чтобы получить доступ к конкретной структуре в массиве структур, необходимо индексировать имя структуры. Например, чтобы отобразить на экране содержимое члена *on_hand* третьей структуры, используйте такую инструкцию.

```
cout << invtry[2].on_hand;
```

Подобно всем переменным массивов, у массивов структур индексирование начинается с нуля.

Простой пример инвентаризации склада

Простой пример инвентаризации склада

Чтобы продемонстрировать применение структур, разработаем простую программу управления складом, в которой для хранения информации о товарах, размещенных на складе компании, используется массив структур типа *inv_type*. Различные функции, определенные в этой программе, взаимодействуют со структурой и ее элементами по-разному.

Инвентарная ведомость будет храниться в структурах типа *inv_type*, организованных в массиве *invtry*.

```
const int SIZE = 100;

struct inv_type {
  char item[40]; // наименование товара
  double cost; // стоимость
  double retail; // розничная цена
  int on_hand; // имеющееся в наличии количество
  int lead_time; // число дней до пополнения запасов
} invtry[SIZE];
```

Размер массива выбран произвольно. При желании его можно легко изменить. Обратите внимание на то, что размерность массива задана с использованием *const*-переменной. А поскольку размер массива во всей программе используется несколько раз, применение *const*-переменной для этой цели весьма оправданно. Чтобы изменить размер массива, достаточно изменить значение константной переменной *SIZE*, а затем перекомпилировать программу. Использование *const*-переменной для определения *"магического числа"*, которое часто употребляется в программе, — обычная практика в профессиональном C++-коде.

Разрабатываемая программа должна обеспечить выполнение следующих действий:

- ввод информации о товарах, хранимых на складе;
- отображение инвентарной ведомости;
- модификация заданного элемента.

Прежде всего напишем функцию *main()*, которая должна иметь примерно такой вид.

```
int main( )
{
  char choice;
  init_list( );
  for(;;) {
    choice = menu( );
    switch(choice) {
      case 'e': enter( );
        break;
      case 'd': display( );
        break;
      case 'u': update( );
        break;
      case 'q': return 0;
    }
  }
```

```
}
```

Функция *main()* начинается с вызова функции *init_list()*, которая инициализирует массив структур. Затем организован цикл, который отображает меню и обрабатывает команду, выбранную пользователем. Приведем код функции *init_list()*.

```
// Инициализация массива структур.
void init_list()
{
  int t;
  // Имя нулевой длины означает пустое имя.
  for(t=0; t<SIZE; t++) *invtry[t].item = '\0';
}
```

Функция *init_list()* подготавливает массив структур для использования, помещая в первый байт поля *item* нулевой символ. Предполагается, что если поле *item* пустое, то структура, в которой оно содержится, попросту не используется.

Функция *menu_select()* отображает команды меню и принимает вариант, выбранный пользователем.

```
// Получение команды меню, выбранной пользователем.
int menu()
{
  char ch;
  cout << '\n';
  do {
    cout << "(E)nter\n"; // Ввести новый элемент.
    cout << "(D)isplay\n"; // Отобразить всю ведомость.
    cout << "(U)pdate\n"; // Изменить элемент.
    cout << "(Q)uit\n\n"; // Выйти из программы.
    cout << "Выберите команду: ";
      cin >> ch;
  }while(!strchr("eduq", tolower(ch)));
  return tolower(ch);
}
```

Пользователь выбирает из предложенного меню команду, вводя нужную букву. Например, чтобы отобразить всю инвентарную ведомость, нажмите букву *"D"*.

Функция *menu()* вызывает библиотечную функцию C++ *strchr()*, которая имеет такой прототип.

```
char *strchr(const char *str, int ch);
```

Эта функция просматривает строку, адресуемую указателем *str*, на предмет вхождения в нее символа, который хранится в младшем байте переменной *ch*. Если такой символ обнаружится, функция возвратит указатель на него. И в этом случае значение, возвращаемое функцией, по определению будет истинным. Но если совпадения символов не произойдет, функция возвратит нулевой указатель, который по определению представляет собой значение ЛОЖЬ. Так здесь организована проверка того, являются ли значения, вводимые пользователем, допустимыми командами меню.

Функция *enter()* предваряет вызов функции *input()*, которая "подсказывает" пользователю порядок ввода данных и принимает их. Рассмотрим код обеих функций.

```
// Ввод элементов в инвентарную ведомость.
void enter()
{
  int i;
  // Находим первую свободную структуру.
  for(i=0; i<SIZE; i++)
    if( !*invtry[i].item) break;

  // Если массив полон, значение i будет равно SIZE.
  if(i==SIZE) {
    cout << "Список полон.\n";
    return;
  }
  input (i);
}

// Ввод информации.
void input(int i)
{
  cout << "Товар: ";
    cin >> invtry[i].item;

  cout << "Стоимость: ";
    cin >> invtry[i].cost;

  cout << "Розничная цена: ";
    cin >> invtry[i].retail;

  cout << "В наличии: ";
    cin >> invtry[i].on_hand;

  cout << "Время до пополнения запасов (в днях): ";
    cin >> invtry[i].lead_time;
}
```

Функция *enter()* сначала находит пустую структуру. Для этого проверяется поле *item* каждого (по очереди) элемента массива *invtry*, начиная с первого. Если поле *item* оказывается пустым, то предполагается, что структура, к которой оно относится, еще ничем не занята. Если не отыщется ни одной свободной структуры при проверке всего массива структур, управляющая переменная цикла *i* станет равной его размеру. Это говорит о том, что массив полон, и в него уже нельзя ничего добавить. Если же в массиве найдется свободный элемент, будет вызвана функция *input()* для получения информации о товаре, вводимой пользователем. Если вас интересует, почему код ввода данных о новом товаре не является частью функции *enter()*, то ответ таков: функция *input()* используется также и функцией *update()*, о которой речь впереди.

Поскольку информация о товарах на складе может меняться, программа ведения

инвентарнои ведомости должна позволять вносить изменения в ее отдельные элементы. Это реализуется путем вызова функции *update()*.

```
// Модификация существующего элемента.
void update( )
{
  int i;
  char name[80];

  cout << "Введите наименование товара: ";
    cin >> name;

  for(i=0; i<SIZE; i++)
    if(!strcmp(name, invtry[i].item)) break;

  if(i==SIZE) {
    cout << "Товар не найден.\n";
    return;
  }

  cout << "Введите новую информацию.\n";
  input(i);
}
```

Эта функция предлагает пользователю ввести наименование товара, информацию о котором ему нужно изменить. Затем она просматривает весь список существующих элементов, и если указанный товар в нем имеется, то вызывается функция *input()*, которая обеспечивает прием от пользователя новой информации.

Нам осталось рассмотреть функцию *display()*. Она выводит на экран инвентарную ведомость в полном объеме. Код функции *display()* выглядит так.

```
// Отображение на экране инвентарной ведомости.
void display( )
{
  int t;
  for(t=0; t<SIZE; t++ ) {
    if(*invtry[t].item) {
      cout << invtry[t].item << '\n';
      cout << "Стоимость: $" << invtry[t].cost;
      cout << "\nВ розницу: $";
      cout << invtry[t].retail << '\n';
      cout << "В наличии: " << invtry[t].on_hand;
      cout << "\nДо пополнения осталось: ";
      cout << invtry[t].lead_time << " дней\n\n";
    }
  }
}
```

Ниже приведена законченная программа ведения инвентарной ведомости. Вам следует ввести эту программу в свой компьютер и исследовать ее работу. Внесите некоторые

изменения и понаблюдайте, как они отразятся на ее выполнении. Попробуйте также расширить программу, добавив функции поиска в списке заданного товара, удаления уже ненужного элемента или полной очистки инвентарной ведомости.

```
/* Простая программа ведения инвентарной ведомости, в которой
используется массив структур.
*/
#include <iostream>
#include <cctype>
#include <cstring>
#include <cstdlib>
using namespace std;

const int SIZE = 100;

struct inv_type {
  char item[40]; // наименование товара
  double cost; // стоимость
  double retail; // розничная цена
  int on_hand; // имеющееся в наличии количество
  int lead_time; // число дней до пополнения запасов
} invtry[SIZE];
void enter(), init_list(), display();
void update(), input(int i);

int menu();

int main()
{
  char choice;
  init_list();
  for (;;) {
    choice = menu();
    switch(choice) {
      case 'e' : enter();
        break;
      case 'd' : display();
        break;
      case 'u': update();
        break;
      case 'q': return 0;
    }
  }
}

// Инициализация массива структур.
```

```
void init_list()
{
  int t;
  // Имя нулевой длины означает пустое имя.
  for(t=0; t<SIZE; t++ ) *invtry[t].item = '\0';
}

//Получение команды меню, выбранной пользователем.
int menu()
{
  char ch;
  cout << '\n';
  do {
    cout << "(E)nter\n"; // Ввести новый элемент.
    cout << "(D)isplay\n"; // Отобразить всю ведомость.
    cout << " (U) pdate\n"; // Изменить элемент.
    cout << " (Q) uit\n\n"; // Выйти из программы.
    cout << "Выберите команду: ";
      cin >> ch;
  }while(!strchr("eduq", tolower(ch)));
  return tolower(ch);
}

//Ввод элементов в инвентарную ведомость.
void enter()
{
  int i;

  // Находим первую свободную структуру.
  for(i=0; i<SIZE; i++)
    if(!*invtry[i].item) break;

  // Если массив полон, значение i будет равно SIZE.
  if(i==SIZE) {
    cout << "Список полон.\n";
    return;
  }
  input(i);
}

// Ввод информации.
void input(int i)
{
  cout << "Товар: ";
    cin >> invtry[i].item;
```

```
  cout << "Стоимость: ";
    cin >> invtry[i].cost;
  cout << "Розничная цена: ";
    cin >> invtry[i].retail;
  cout << "В наличии: ";
    cin >> invtry[i].on_hand;
  cout << "Время до пополнения запасов (в днях): ";
    cin >> invtry[i].lead_time;
}

// Модификация существующего элемента.
void update()
{
  int i;
  char name[80];
  cout << "Введите наименование товара: ";
    cin >> name;
  for(i=0; i<SIZE; i++)
    if(!strcmp(name, invtry[i].item)) break;
  if(i==SIZE) {
    cout << "Товар не найден.\n";
    return;
  }
  cout << "Введите новую информацию.\n";
  input (i);
}

// Отображение на экране инвентарной ведомости.
void display()
{
  int t;
  for(t=0; t<SIZE; t++) {
    if(*invtry[t].item) {
      cout << invtry[t].item << '\n';
      cout << "Стоимость: $" << invtry[t].cost;
      cout << "\nВ розницу: $";
      cout << invtry[t].retail << '\n';
      cout << "В наличии: " << invtry[t].on_hand;
      cout << "\nДо пополнения осталось: ";
      cout << invtry[t].lead time << " дней\n\n";
```

```
    }
  }
}
```

Передача структур функциям

При передаче функции структуры в качестве аргумента используется механизм передачи параметров по значению. Это означает, что любые изменения, внесенные в содержимое структуры в теле функции, которой она передана, не влияют на структуру, используемую в качестве аргумента. Однако следует иметь в виду, что передача больших структур требует значительных затрат системных ресурсов. (Как правило, чем больше данных передается функции, тем больше расходуется системных ресурсов.)

Используя структуру в качестве параметра, помните, что тип аргумента должен соответствовать типу параметра. Например, в следующей программе сначала объявляется структура *sample*, а затем функция *f1()* принимает параметр типа *sample*.

```
// Передача функции структуры в качестве аргумента.
#include <iostream>
using namespace std;

// Определяем тип структуры.
struct sample {
  int a;
  char ch;
};
void f1(sample parm);

int main()
{
  struct sample arg; // Объявляем переменную arg типа sample.
  arg.a = 1000;
  arg.ch = 'x';
  f1(arg);

  return 0;
}

void f1(sample parm)
{
  cout << parm.a << " " << parm.ch << "\n";
}
```

Здесь как аргумент *arg* в функции *main()*, так и параметр *parm* в функции *f1()* имеют одинаковый тип. Поэтому аргумент *arg* можно передать функции *f1()*. Если бы типы этих структур были различны, при компиляции программы было бы выдано сообщение об ошибке.

Присваивание структур

Содержимое одной структуры можно присвоить другой, если обе эти структуры имеют одинаковый тип. Например, следующая программа присваивает значение структурной переменной *svar1* переменной *svar2*.

```
// Демонстрация присваивания значений структур.
#include <iostream>
using namespace std;

struct stype {
  int a, b;
};

int main( )
{
  stype svar1, svar2;
  svar1.a = svar1.b = 10;
  svar2.a = svar2.b = 20;

  cout << "Структуры до присваивания.\n";
  cout << "svar1: " << svar1.a << ' ' << svar1.b;
  cout <<'\n';
  cout << "svar2: " << svar2.a << ' ' << svar2.b;
  cout <<"\n\n";

  svar2 = svar1; // присваивание структур
  cout << "Структуры после присваивания.\n";
  cout << "svar1: " << svar1.a << ' ' << svar1.b;
  cout << '\n';
  cout << "svar2: " << svar2.a << ' ' << svar2.b;

  return 0;
}
```

Эта программа генерирует следующие результаты.

```
Структуры до присваивания.
svar1: 10 10
svar2: 20 20
Структуры после присваивания,
svar1: 10 10
svar2: 10 10
```

В C++ каждое новое объявление структуры определяет новый тип. Следовательно, даже если две структуры физически одинаковы, но имеют разные имена типов, компилятор будет считать их разными и не позволит присвоить значение одной из них другой. Рассмотрим следующий фрагмент кода. Он некорректен и поэтому не скомпилируется.

```
struct stype1 {
  int a, b;
};

struct stype2 {
  int a, b;
};
```

```
stype1 svar1;
stype2 svar2;
svar2 = svar1; // Ошибка из-за несоответствия типов.
```

Несмотря на то что структуры *stype1* и *stype2* физически одинаковы, с точки зрения компилятора они являются отдельными типами.

Узелок на память. *Одну структуру можно присвоить другой только в том случае, если обе они имеют одинаковый тип.*

Использование указателей на структуры и оператора "стрелка"

В C++ указатели на структуры можно использовать таким же способом, как и указатели на переменные любого другого типа. Однако использование указателей на структуры имеет ряд особенностей, которые необходимо учитывать.

Указатель на структуру объявляется так же, как указатель на любую другую переменную, т.е. с помощью символа *"*"*, поставленного перед именем структурной переменной. Например, используя определенную выше структуру *inv_type*, можно записать следующую инструкцию, которая объявляет переменную *inv_pointer* указателем на данные типа *inv_type*:

```
inv_type *inv_pointer;
```

Чтобы найти адрес структурной переменной, необходимо перед ее именем разместить оператор *"&"*. Например, предположим, с помощью следующего кода мы определяем структуру, объявляем структурную переменную и указатель на структуру определенного нами типа.

```
struct bal {
  float balance;
  char name[80];
}
person;
bal *p; // Объявляем указатель на структуру.
```

Тогда при выполнении инструкции

```
p = &person;
```

в указатель *p* будет помещен адрес структурной переменной *person*.

К членам структуры можно получить доступ с помощью указателя на эту структуру. Но в этом случае используется не оператор *"точка"*, а оператор *"->"*. Например, при выполнении этой инструкции мы получаем доступ к полю *balance* через указатель *p*:

```
p->balance
```

Оператор *"->"* называется оператором *"стрелка"*. Он образуется с использованием знаков *"минус"* и *"больше"*.

Оператор "стрелка" (->) позволяет получить доступ к членам структуры с помощью указателя.

Указатель на структуру можно использовать в качестве параметра функции. Важно помнить о таком способе передачи параметров, поскольку он работает гораздо быстрее, чем в случае, когда функции "собственной персоной" передается объемная структура. (Передача указателя всегда происходит быстрее, чем передача самой структуры.)

Узелок на память. *Чтобы получить доступ к членам структуры, используйте оператор "точка". Чтобы получить доступ к членам структуры с помощью указателя, используйте оператор "стрелка".*

Пример использования указателей на структуры

В качестве интересного примера использования указателей на структуры можно рассмотреть C++-функции *времени* и *даты*. Эти функции считывают значения текущего системного *времени* и *даты*. Для их использования в программу необходимо включить заголовок *<ctime>*. Этот заголовок поддерживает два типа даты, требуемые упомянутыми функциями. Один из этих типов, *time_t*, предназначен для представления системного времени и даты в виде длинного целочисленного значения, которое используется в качестве *календарного времени*. Второй тип представляет собой структуру *tm*, которая содержит отдельные элементы даты и времени. Такое представление времени называют *поэлементным*. Структура *tm* имеет следующий формат.

```
struct tm {
  int tm_sec; /* секунды, 0-59 */
  int tm_min; /* минуты, 0-59 */
  int tm_hour; /* часы, 0-23 */
  int tm_mday; /* день месяца, 1-31 */
  int tm_mon; /* месяц, начиная с января, 0-11 */
  int tm_year; /* год после 1900 */
  int tm_wday; /* день, начиная с воскресенья, 0-6 */
  int tm_yday; /* день, начиная с 1-го января, 0-365 */
  int tm_isdst /* индикатор летнего времени */
}
```

Значение *tm_isdst* положительно, если действует режим летнего времени (Daylight Saving Time), равно нулю, если не действует, и отрицательно, если информация об этом недоступна.

Основным средством определения времени и даты в C++ является функция *time()*, которая имеет такой прототип:

```
time_t time(time_t *curtime);
```

Функция *time()* возвращает текущее календарное время системы. Если в системе отсчет времени не производится, возвращается значение *-1*. Функцию *time()* можно вызывать либо с нулевым указателем, либо с указателем на переменную *curtime* типа *time_t*. В последнем случае этой переменной будет присвоено значение текущего календарного времени.

Чтобы преобразовать календарное время в поэлементное, используйте функцию *localtime()*, которая имеет такой прототип:

```
struct tm *localtime(const time_t *curtime);
```

Функция *localtime()* возвращает указатель на поэлементную форму параметра *curtime*, представленного в виде структуры *tm*. Значение *curtime* представляет локальное время. Его обычно получают с помощью функции *time()*.

Структура, используемая функцией *localtime()* для хранения времени в поэлементной форме, размещается в памяти статически и перезаписывается при каждом вызове этой функции. Если нужно сохранить содержимое этой структуры, скопируйте его в какую-нибудь другую область памяти.

Следующая программа демонстрирует использование функций *time()* и *localtime()*, отображая на экране текущее системное время.

```
// Эта программа отображает текущее системное время.
#include <iostream>
#include <ctime>
using namespace std;
```

```
int main( )
{
  struct tm *ptr;
  time_t lt;

  lt = time('\0');

  ptr = localtime(&lt);

  cout << ptr->tm_hour << ':' << ptr->tm_min;
  cout << ':' << ptr->tm_sec;

  return 0;
}
```

Вот один из возможных результатов выполнения этой программы:

```
14:52:30
```

Несмотря на то что ваши программы могут использовать поэлементную форму представления времени и даты (как показано в предыдущем примере), проще всего сгенерировать строку времени и даты с помощью функции *asctime()*, прототип который выглядит так:

```
char *asctime(const struct tm *ptr);
```

Функция *asctime()* возвращает указатель на строку, которая содержит результат преобразования информации, хранимой в адресуемой параметром *ptr* структуре, и имеет следующую форму.

```
день месяц число часы:минуты:секунды год\n\0
```

Указатель на структуру, передаваемый функции *asctime()*, часто получают с помощью функции *localtime()*.

Область памяти, используемая функцией *asctime()* для хранения форматированной строки результата, представляет собой символьный массив (статически выделяемый в памяти), который перезаписывается при каждом вызове этой функции. Если нужно сохранить содержимое данной строки, скопируйте его в какую-нибудь другую область памяти.

В следующей программе демонстрируется использование функции *asctime()* для отображения системного времени и даты.

```
// Эта программа отображает текущее системное время.
#include <iostream>
#include <ctime>
using namespace std;

int main( )
{
  struct tm *ptr;
  time_t lt;

  lt = time('\0');

  ptr = localtime(&lt);
  cout << asctime(ptr);

  return 0;
```

```
}
```

Вот один из возможных результатов выполнения этой программы.

```
Wed Jul 28 15:05:51 2004
```

В языке C++ предусмотрены и другие функции даты и времени, с которыми можно познакомиться, обратившись к документации, прилагаемой к вашему компилятору.

Ссылки на структуры

Для доступа к структуре можно использовать ссылку. Ссылка на структуру часто используется в качестве параметра функции или значения, возвращаемого функцией. При получении доступа к членам структуры с помощью ссылки используйте оператор *"точка"*. (Оператор *"стрелка"* зарезервирован для доступа к членам структуры с помощью указателя.)

В следующей программе показано, как можно использовать структуру при передаче функции параметров по ссылке.

```
// Демонстрируем использование ссылки на структуру.
#include <iostream>
using namespace std;

struct mystruct {
  int a; int b;
};
mystruct &f(mystruct &var);

int main( )
{
  mystruct x, y;
  x.a = 10; x.b = 20;

  cout << "Исходные значения полей x.a and x.b: ";
  cout << x.a << ' ' << x.b << '\n';

  y = f (x);

  cout << "Модифицированные значения полей x.a и x.b: ";
  cout << x.a << ' ' << x.b << '\n';

  cout << "Модифицированные значения полей y.a и y.b: ";
  cout << y.a << ' ' << y.b << '\n';

  return 0;
}

// Функция, которая получает и возвращает ссылку на структуру.
mystruct &f(mystruct &var)
{
  var.a = var.a * var.a;
  var.b = var.b / var.b;
  return var;
```

```
}
```

Вот результаты выполнения этой программы.

```
Исходные значения полей x.a and x.b: 10 20
Модифицированные значения полей x.a и x.b: 100 1
Модифицированные значения полей y.a и y.b: 100 1
```

Ввиду существенных затрат системных ресурсов на передачу структуры функции (или при возвращении ее функцией) многие C++-программисты для выполнения таких задач используют ссылки на структуры.

Использование в качестве членов структур массивов и структур

Член структуры может иметь любой допустимый тип данных, в том числе и такие составные типы, как массивы и другие структуры. Поскольку эта тема нередко вызывает у программистов затруднения, имеет смысл остановиться на ней подробнее.

Массив, используемый в качестве члена структуры, обрабатывается вполне ожидаемым способом. Рассмотрим такую структуру.

```
struct stype {
  int nums[10][10]; // Целочисленный массив размерностью 10 x 10.
  float b;
} var;
```

Чтобы обратиться к элементу массива *nums* с "координатами" *3,7* в структуре *var* типа *stype*, следует записать такой код:

```
var.nums[3][7]
```

Как показано в этом примере, если массив является членом структуры, то для доступа к элементам этого массива индексируется имя массива, а не имя структуры.

Если некоторая структура является членом другой структуры, она называется вложенной структурой. В следующем примере структура *addr* вложена в структуру *emp*.

```
struct addr {
  char name[40];
  char street[40];
  char city[40];
  char zip[10];
}
struct emp {
  addr address;
  float wage;
} worker;
```

Здесь структура *emp* имеет два члена. Первым членом является структура типа *addr*, которая будет содержать адрес служащего. Вторым членом является переменная *wage*, которая хранит его оклад. При выполнении следующего фрагмента кода полю *zip* структуры *address*, которая является членом структуры *worker*, будет присвоен почтовый индекс *98765*:

```
worker.address.zip = 98765;
```

Как видите, члены структур указываются слева направо, от самой крайней внешней до самой дальней внутренней.

Структура также может содержать в качестве своего члена указатель на эту же структуру. И

в самом деле для структуры вполне допустимо содержать член, которыи является указателем на нее саму. Например, в следующей структуре переменная *sptr* объявляется как указатель на структуру типа *mystruct*, т.е. на объявляемую здесь структуру.

```
struct mystruct {
  int a;
  char str[80];
  mystruct *sptr; // указатель на объекты типа mystruct
};
```

Структуры, содержащие указатели на самих себя, часто используются при создании таких структур данных, как *связные списки*. По мере изучения языка C++ вы встретите приложения, в которых применяются подобные вещи.

Сравнение C- и C++-структур

C++-структуры — потомки C-структур. Следовательно, любая C-структура также является действительной C++-структурой. Между ними, однако, существуют важные различия. Во-первых, как будет показано в следующей главе, C++-структуры имеют некоторые уникальные атрибуты, которые позволяют им поддерживать *объектно-ориентированное* программирование. Во-вторых, в языке C структура не определяет в действительности новый тип данных. Этим может "похвалиться" лишь C++-структура. Как вы уже знаете, определяя структуру в C++, вы определяете новый тип, который называется по имени этой структуры. Этот новый тип можно использовать для объявления переменных, определения значений, возвращаемых функциями, и т.п. Однако в C имя структуры называется ее *тегом* (или *дескриптором*). А *тег* сам по себе не является именем типа.

Чтобы понять это различие, рассмотрим следующий фрагмент C-кода.

```
struct C_struct {
  int a; int b;
}
// объявление переменной C_struct
struct C_struct svar;
```

Обратите внимание на то, что приведенное выше определение структуры в точности такое же, как в языке C++. Теперь внимательно рассмотрите объявление структурной переменной *svar*. Оно также начинается с ключевого слова *struct*. В языке C после определения структуры для полного задания типа данных все равно нужно использовать ключевое слово *struct* совместно с тегом этой структуры (в данном случае с идентификатором *C_struct*).

Если вам придется преобразовывать старые C-программы в код C++, не беспокойтесь о различиях между C- и C++-структурами, поскольку C++ по-прежнему принимает C-ориентированные объявления. Например, предыдущий фрагмент C-кода корректно скомпилируется как часть любой C++-программы. С точки зрения компилятора C++ в объявлении переменной *svar* всего лишь избыточно использовано ключевое слово *struct*, без которого в C++ можно обойтись.

Битовые поля структур

Битовое поле — *это бит-ориентированный член структуры*.

В отличие от многих других компьютерных языков, в C++ предусмотрен встроенный способ доступа к конкретному разряду байта. Побитовый доступ возможен путем использования битовых полей. Битовые поля могут оказаться полезными в различных ситуациях. Приведем всего три примера. Во-первых, если вы имеете дело с ограниченным объемом памяти, можно хранить несколько булевых (логических) значений в одном байте. Во-вторых, некоторые интерфейсы устройств передают информацию, закодированную именно в битах. И, в-третьих, существуют подпрограммы кодирования, которым нужен доступ к отдельным битам в рамках байта. Реализация всех этих функций возможна с помощью поразрядных операторов, как было показано в предыдущей главе, но битовое поле может сделать вашу программу более прозрачной и читабельной, а также повысить ее переносимость.

программу более прозрачной и читабельной, а также повысить ее переносимость.

Метод, который использован в языке C++ для доступа к битам, основан на применении структур. Битовое поле — это в действительности специальный тип члена структуры, который определяет свой размер в битах. Общий формат определения битовых полей таков.

```
struct имя_типа_структуры {
  тип имя1 : длина;
  тип имя2 : длина;
  .
  .
  .
  тип имяN : длина;
};
```

Здесь элемент *тип* означает тип битового поля, а элемент *длина* — количество битов в этом поле. Битовое поле должно быть объявлено как значение целочисленного типа или перечисления. Битовые поля длиной *1 бит* объявляются как значения типа без знака (*unsigned*), поскольку единственный бит не может иметь знакового разряда.

Битовые поля обычно используются для анализа входных данных, принимаемых от устройств, входящих в состав оборудования системы. Например, порт состояний последовательного адаптера связи может возвращать байт состояния, организованный таким образом.

Бит	Значение в установленном состоянии
0	Изменение в линии установки в исходное состояние
1	Изменение в линии готовности данных
2	Обнаружен задний фронт
3	Изменение в линии приема данных
4	Установка в исходное состояние
5	Данные готовы
6	Телефонный сигнал вызова
7	Сигнал принят

Для представления информации, которая содержится в байте состояний, можно использовать следующие битовые поля.

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge: 1;
  unsigned delta_rec: 1;
  unsigned cts: 1;
  unsigned dsr: 1;
  unsigned ring: 1;
  unsigned rec_line: 1;
} status;
```

Чтобы определить, когда можно отправить или получить данные, используется код, подобный следующему.

```
status = get_port_status( );
if(status.cts) cout << "Установка в исходное состояние";
if(status.dsr) cout << "Данные готовы";
```

Чтобы присвоить битовому полю значение, достаточно использовать такую же форму, которая обычно применяется для элемента структуры любого другого типа. Например, следующая инструкция очищает битовое поле *ring*:

```
status.ring = 0;
```

Как видно из этих примеров, доступ к каждому битовому полю можно получить с помощью оператора *"точка"*. Но если общий доступ к структуре осуществляется через указатель, необходимо использовать оператор *"->"*.

Следует иметь в виду, что совсем необязательно присваивать имя каждому битовому полю. Это позволяет обращаться только к нужным битам, *"обходя"* остальные. Например, если вас интересуют только биты *cts* и *dsr*, вы могли бы объявить структуру *status_type* следующим образом.

```
struct status_type {
  unsigned : 4;
  unsigned cts: 1;
  unsigned dsr: 1;
} status;
```

Обратите здесь внимание на то, что биты после последнего именованного *dsr* нет необходимости вообще упоминать.

В структуре можно смешивать *"обычные"* члены с битовыми полями. Вот пример.

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off: 1; // работает или нет
  unsigned hourly: 1: // почасовая оплата или оклад
  unsigned deductions: 3: // удержание налога
};
```

Эта структура определяет запись по каждому служащему, в которой используется только один байт для хранения трех элементов информации: статус служащего, характер оплаты его труда (почасовая оплата или твердый оклад) и налоговая ставка. Без использования битовых полей для хранения этой информации пришлось бы занять три байта.

Использование битовых полей имеет определенные ограничения. Программист не может получить адрес битового поля или ссылку на него. Битовые поля нельзя хранить в массивах. Их нельзя объявлять статическими. При переходе от одного компьютера к другому невозможно знать наверняка порядок следования битовых полей: справа налево или слева направо. Это означает, что любая программа, в которой используются битовые поля, может страдать определенной зависимостью от марки компьютера. Возможны и другие ограничения, связанные с особенностями реализации компилятора C++, поэтому имеет смысл прояснить этот вопрос в соответствующей документации.

В следующем разделе представлена программа, в которой используются битовые поля для отображения символьных *ASCII*-кодов в двоичной системе счисления.

Объединения

Объединение состоит из нескольких переменных, которые разделяют одну и ту же область памяти.

Объединение состоит из нескольких переменных, которые разделяют одну и ту же область памяти. Следовательно, объединение обеспечивает возможность интерпретации одной и той же конфигурации битов двумя (или более) различными способами. Объявление объединения, как нетрудно убедиться на следующем примере, подобно объявлению структуры.

```
union utype {
  short int i;
  char ch;
};
```

Объявление объединения начинается с ключевого слова union.

Здесь объявляется объединение, в котором значение типа *short int* и значение типа *char* разделяют одну и ту же область памяти. Необходимо сразу же прояснить один момент: невозможно сделать так, чтобы это объединение хранило и целочисленное значение, и символ одновременно, поскольку переменные *i* и *ch* накладываются (в памяти) друг на друга. Но программа в любой момент может обрабатывать информацию, содержащуюся в этом объединении, как целочисленное значение или как символ. Следовательно, объединение обеспечивает два (или больше) способа представления одной и той же порции данных. Как видно из этого примера, объединение объявляется с помощью ключевого слова *union*.

Как и при использовании структур, при объявлении объединения не определяется ни одна переменная. Переменную можно объявить, разместив ее имя в конце объявления либо воспользовавшись отдельной инструкцией объявления. Чтобы объявить переменную объединения именем *u_var* типа *utype*, достаточно записать следующее:

```
utype u_var;
```

В переменной объединения *u_var* как переменная *i* типа *short int*, так и символьная переменная *ch* разделяют одну и ту же область памяти. (Безусловно, переменная *i* занимает два байта, а символьная переменная *ch* использует только один.) Как переменные *i* и *ch* разделяют одну область памяти, показано на рис. 10.2.

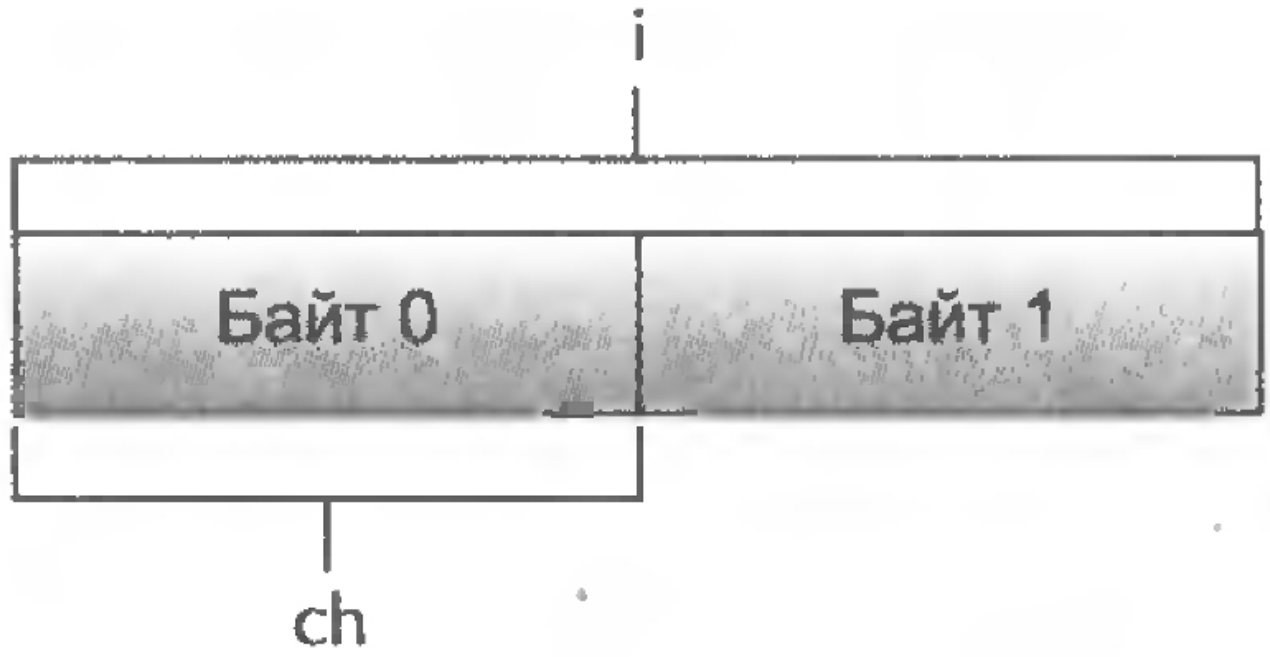


Рис. 10.2. Переменные `i` и `ch` вместе используют объединение `u_var`

При объявлении объединения компилятор автоматически выделяет область памяти, достаточную для хранения в объединении переменных самого большого по объему типа.

Чтобы получить доступ к элементу объединения, используйте тот же синтаксис, который применяется и для структур: операторы *"точка"* и *"стрелка"*. При непосредственном обращении к объединению (или посредством ссылки) используется оператор *"точка"*. Если же доступ к переменной объединения осуществляется через указатель, используется оператор *"стрелка"*. Например, чтобы присвоить букву *'A'* элементу *ch* объединения *u_var*, достаточно использовать такую запись.

```
u_var.ch = 'A';
```

В следующем примере функции передается указатель на объединение *u_var*. В теле этой функции с помощью указателя переменной *i* присваивается значение *10*.

```
// ...
  func1(&u_var); // Передаем функции func1() указатель  на объединение u_var.
// ...
}

void fund (utype *un)
{
  un->i = 10; /* Присваиваем число 10 члену объединения u_var с помощью указателя. */
}
```

Поскольку объединения позволяют вашей программе интерпретировать одни и те же данные по-разному, они часто используются в случаях, когда требуется необычное преобразование типов. Например, следующая программа использует объединение для перестановки двух байтов, которые составляют короткое целочисленное значение. Здесь для отображения содержимого целочисленных переменных используется функция *disp_binary()*, разработанная в главе 9. (Эта программа написана в предположении, что короткие целочисленные значения имеют длину два байта.)

```
// Использование объединения для перестановки двух байтов в рамках короткого целочисленного значения.
#include <iostream>
using namespace std;

void disp_binary(unsigned u);
union swap_bytes {
  short int num;
  char ch[2];
};

int main()
{
  swap_bytes sb;
  char temp;

  sb.num = 15; // двоичный код: 0000 0000 0000 1111

  cout << "Исходные байты: ";
```

```
  cout << "Исходные байты: ";
  disp_binary(sb.ch[1]);
  cout << " ";
  disp_binary(sb.ch[0]);

  cout << "\n\n";
  // Обмен байтов.
  temp = sb.ch[0];
  sb.ch[0] = sb.ch[1];
  sb.ch[1] = temp;

  cout << "Байты после перестановки: ";
  disp_binary(sb.ch[1]);
  cout << " ";
  disp_binary(sb.ch[0]);
  cout << "\n\n";

  return 0;
}

// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
  register int t;
  for(t=128; t>0; t=t/2)
    if(u & t) cout << "1 ";
    else cout << "0 ";
}
```

При выполнении программа генерирует такие результаты.

```
Исходные байты: 0000 0000 0000 1111
Байты после перестановки: 0000 1111 0000 0000
```

В этой программе целочисленной переменной *sb.num* присваивается число *15*. Перестановка двух байтов, составляющих это значение, выполняется путем обмена двух символов, которые образуют массив *ch*. В результате старший и младший байты целочисленной переменной *num* меняются местами. Эта операция возможна лишь потому, что как переменная *num*, так и массив *ch* разделяют одну и ту же область памяти.

В следующей программе демонстрируется еще один пример использования объединения. Здесь объединения связываются с битовыми полями, используемыми для отображения в двоичной системе счисления *ASCII*-кода, генерируемого при нажатии любой клавиши. Эта программа также демонстрирует альтернативный способ отображения отдельных битов, составляющих байт. Объединение позволяет присвоить значение нажатой клавиши символьной переменной, а битовые поля используются для отображения отдельных битов.

```
// Отображение ASCII-кода символов в двоичной системе счисления.
#include <iostream>
#include <conio.h>
using namespace std;

// Битовые поля, которые будут расшифрованы
```

```
// Битовые поля, которые будут расшифрованы.
struct byte {
  unsigned a : 1;
  unsigned b : 1;
  unsigned c : 1;
  unsigned d : 1;
  unsigned e : 1;
  unsigned f : 1;
  unsigned g : 1;
  unsigned h : 1;
};

union bits {
  char ch;
  struct byte bit;
}ascii;

void disp_bits(bits b);

int main( )
{
  do {
    cin >> ascii.ch;
    cout << ":";
    disp_bits(ascii);
  }while(ascii.ch!='q'); // Выход при вводе буквы "q".
  return 0;
}

// Отображение конфигурации битов для каждого символа.
void disp_bits(bits b)
{
  if(b.bit.h) cout << "1";
    else cout << "0";
  if(b.bit.g) cout << "1";
    else cout << "0";
  if(b.bit.f) cout << "1";
    else cout << "0 ";
  if(b.bit.e) cout << "1";
    else cout << "0";
  if(b.bit.d) cout << "1";
    else cout << "0";
  if(b.bit.c) cout << "1";
    else cout << "0";
```

```
  if(b.bit.b) cout << "1";
    else cout << "0";
  if(b.bit.a) cout << "1";
    else cout << "0";
  cout << "\n";
}
```

Вот как выглядит один из возможных вариантов выполнения этой программы.

```
a: 0 1 1 0 0 0 0 1
b: 0 1 1 0 0 0 1 0
c: 0 1 1 0 0 0 1 1
d: 0 1 1 0 0 1 0 0
e: 0 1 1 0 0 1 0 1
f: 0 1 1 0 0 1 1 0
g: 0 1 1 0 0 1 1 1
h: 0 1 1 0 1 0 0 0
i: 0 1 1 0 1 0 0 1
j: 0 1 1 0 1 0 1 0
k: 0 1 1 0 1 0 1 1
1: 0 1 1 0 1 1 0 0
m: 0 1 1 0 1 1 0 1
n: 0 1 1 0 1 1 1 0
o: 0 1 1 0 1 1 1 1
p: 0 1 1 1 0 0 0 0
q: 0 1 1 1 0 0 0 1
```

Важно! *Поскольку объединение предполагает, что несколько переменных разделяют одну и ту же область памяти, это средство предоставляет программисту возможность хранить информацию, которая (в зависимости от ситуации) может содержать различные типы данных, и получать доступ к этой информации. По сути, объединения обеспечивают низкоуровневую поддержку принципов полиморфизма. Другими словами, объединение обеспечивает единый интерфейс для нескольких различных типов данных, воплощая таким образом концепцию "один интерфейс — множество методов" в своей самой простой форме.*

Анонимные объединения

Анонимные объединения позволяют объявлять переменные, которые разделяют одну и ту же область памяти.

В C++ предусмотрен специальный тип объединения, который называется *анонимным*. Анонимное объединение не имеет наименования типа, и поэтому объект такого объединения объявить невозможно. Но анонимное объединение сообщает компилятору о том, что его члены разделяют одну и ту же область памяти. При этом обращение к самим переменным объединения происходит непосредственно, без использования оператора *"точка"*. Рассмотрим такой пример.

```
//Демонстрация использования анонимного объединения.
#include <iostream>
using namespace std;
```

```
int main()
{
  // Это анонимное объединение.
  union {
    short int count;
    char ch[2];
  };
  // Вот как происходит непосредственное обращение к членам анонимного объединения.
  ch[0] = 'X';
  ch[1] = 'Y';
  cout << "Объединение в виде символов: " << ch[0] <<ch[1] << '\n';
  cout << "Объединение в виде целого значения: " <<count << '\n';
  return 0;
}
```

Эта программа отображает следующий результат.

```
Объединение в виде символов: XY
Объединение в виде целого значения: 22872
```

Число *22872* получено в результате помещения символов *X* и *Y* в младший и старший байты переменной *count* соответственно. Как видите, к обеим переменным, входящим в состав объединения, как *count*, так и *ch*, можно получить доступ так же, как к обычным переменным, а не как к составляющим объединения. Несмотря на то что они объявлены как часть анонимного объединения, их имена находятся на том же уровне области видимости, что и другие локальные переменные, объявленные на уровне объединения. Таким образом, член анонимного объединения не может иметь имя, совпадающее с именем любой другой переменной, объявленной в той же области видимости.

Анонимное объединение представляет собой средство, с помощью которого программист может сообщить компилятору о своем намерении, чтобы две (или больше) переменные разделяли одну и ту же область памяти. За исключением этого момента, члены анонимного объединения ведут себя подобно любым другим переменным.

Использование оператора sizeof для гарантии переносимости программного кода

Как было показано, структуры и объединения создают объекты различных размеров, которые зависят от размеров и количества их членов. Более того, размеры таких встроенных типов, как *int*, могут изменяться при переходе от одного компьютера к другому. Иногда компилятор заполняет структуру или объединение так, чтобы выровнять их по границе четного слова или абзаца. (Абзац содержит 16 байт.) Поэтому, если в программе нужно определить размер (в байтах) структуры или объединения, используйте оператор *sizeof*. Не пытайтесь вручную выполнять сложение отдельных членов. Из-за заполнения или иных аппаратно-зависимых факторов размер структуры или объединения может оказаться больше суммы размеров отдельных их членов.

И еще. Объединение всегда будет занимать область памяти, достаточную для хранения его самого большого члена. Рассмотрим пример.

```
union x {
  char ch;
  int i;
  double f;
```

```
} u_var;
```

Здесь при выполнении оператора *sizeof u_var* получим результат *8* (при условии, что *double*-значение занимает *8 байт*). Во время выполнения программы не имеет значения, что реально будет храниться в переменной *u_var;* здесь важен размер самой большой переменной, входящей в состав объединения, поскольку объединение должно иметь размер самого большого его элемента.

Переходим к объектно-ориентированному программированию

Эта глава заключает описание *не объектно-ориентированных* атрибутов C++. Начиная со следующей главы, мы будем рассматривать средства, которые поддерживают *объектно-ориентированное программирование* (Object Oriented Programming— OOP), или *ООП*. Чтобы понять объектно-ориентированные средства C++ и научиться их эффективно применять, необходимо глубокое понимание материала этой и предыдущих девяти глав. Поэтому, возможно, вам стоит повторить пройденный материал. Особое внимание при повторении уделите указателям, структурам, функциям и перегрузке функций.

Глава 11: Введение в классы

В этой главе мы познакомимся с классом. *Класс* — это фундамент, на котором построена C++-поддержка объектно-ориентированного программирования, а также ядро многих более сложных программных средств. *Класс* — это базовая единица инкапсуляции, которая обеспечивает механизм создания объектов.

Основы понятия класса

Объектно-ориентированное программирование построено на понятии класса.

Начнем с определения терминов класса и объекта. Класс определяет новый тип данных, который задает формат *объекта*. Класс включает как данные, так и код, предназначенный для выполнения над этими данными. Следовательно, *класс связывает данные с кодом*. В C++ спецификация класса используется для построения объектов. *Объекты* — это экземпляры класса. По сути, класс представляет собой набор планов, которые определяют, как строить объект. Важно понимать, что *класс* — это логическая абстракция, которая реально не существует до тех пор, пока не будет создан объект этого класса, т.е. то, что станет физическим представлением этого класса в памяти компьютера.

Определяя *класс*, вы объявляете данные, которые он содержит, и код, который выполняется над этими данными. Хотя очень простые классы могут содержать только код или только данные, большинство реальных классов содержат оба компонента. В классе данные объявляются в виде переменных, а код оформляется в виде функций. Функции и переменные, составляющие класс, называются его *членами*. Таким образом, переменная, объявленная в классе, называется *членом данных*, а функция, объявленная в классе, называется *функцией-членом*. Иногда вместо термина *член данных* используется термин *переменная экземпляра* (или *переменная реализации*).

Объявление класса начинается с ключевого слова class.

Класс создается с помощью ключевого слова *class*. Объявление класса синтаксически подобно объявлению структуры. Рассмотрим пример. Следующий класс определяет тип *queue*, который предназначен для реализации очереди. (Под очередью понимается список с дисциплиной обслуживания в порядке поступления, т.е. "первым прибыл — первым обслужен".)

```
// Создание класса queue.
class queue {
   int q[100];
   int sloc, rloc;
 public:
   void init()
```

```
    void init( );
    void qput(int i);
    int qget( );
};
```

Рассмотрим подробно объявление этого класса.

По умолчанию члены класса являются закрытыми (private-членами).

Все члены класса *queue* объявлены в теле инструкции *class*. Членами данных класса *queue* являются переменные *q*, *sloc* и *rloc*. Кроме того, здесь определено три функции-члена: *init()*, *qput()* и *qget()*.

Любой класс может содержать как закрытые, так и открытые члены. По умолчанию все элементы, определенные в классе, являются закрытыми. Например, переменные *q*, *sloc* и *rloc* являются закрытыми. Это означает, что к ним могут получить доступ только другие члены класса *queue;* никакие другие части программы этого сделать не могут. В этом состоит одно из проявлений инкапсуляции: программист в полной мере может управлять доступом к определенным элементам данных. Закрытыми можно объявить и функции (в этом примере таких нет), и тогда их смогут вызывать только другие члены этого класса.

Ключевое слово public используется для объявления открытых членов класса.

Чтобы сделать части класса открытыми (т.е. доступными для других частей программы), необходимо объявить их после ключевого слова *public*. Все переменные или функции, определенные после спецификатора *public*, доступны для всех других функций программы. Итак, в классе *queue* функции *init()*, *qput()* и *qget()* являются открытыми. Обычно в программе организуется доступ к закрытым членам класса через его открытые функции. Обратите внимание на то, что после ключевого слова *public* стоит двоеточие.

Следует иметь в виду, что объект образует своего рода связку между кодом и данными. Так, любая функция-член имеет доступ к закрытым элементам класса. Это означает, что функции *init()*, *qput()* и *qget()* имеют доступ к переменным *q*, *sloc* и *rloc*. Чтобы добавить функцию-член в класс, определите ее прототип в объявлении этого класса.

Определив класс, можно создать объект этого "классового" типа, используя имя класса. Таким образом, имя класса становится спецификатором нового типа. Например, при выполнении следующей инструкции создается два объекта *Q1* и *Q2* типа *queue*,

```
queue Q1, Q2;
```

После создания объект класса будет иметь собственную копию членов данных, которые составляют класс. Это означает, что каждый из объектов *Q1* и *Q2* будет иметь собственные отдельные копии переменных *q*, *sloc* и *rloc*. Следовательно, данные, связанные с объектом *Q1*, отделены (изолированы) от данных, связанных с объектом *Q2*.

Чтобы получить доступ к открытому члену класса через объект этого класса, используйте оператор "точка" (именно так это делается и при работе со структурами). Например, чтобы вывести на экран значение переменной *sloc*, принадлежащей объекту *Q1*, используйте следующую инструкцию.

```
cout << Q1.sloc;
```

Давайте вспомним: в C++ класс создает новый тип данных, который можно использовать для создания объектов. В частности, класс создает логическую конструкцию, которая определяет отношения между ее членами. Объявляя переменную класса, мы создаем объект. Объект характеризуется физическим существованием и является конкретным экземпляром класса. (Другими словами, объект занимает определенную область памяти, а определение типа — нет.) Более того, каждый объект класса имеет собственную копию данных, определенных в этом классе.

В объявлении класса *queue* содержатся прототипы функций-членов. Поскольку функции-члены обеспечены своими прототипами в определении класса, их не нужно помещать больше ни в какое другое место программы

больше ни в какое другое место программы.

Чтобы реализовать функцию, которая является членом класса, необходимо сообщить компилятору, какому классу она принадлежит, квалифицировав имя этой функции с именем класса. Например, вот как можно записать код функции *qput()*.

```
void queue::qput(int i)
{
  if(sloc==100) {
    cout << "Очередь заполнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}
```

Оператор разрешения области видимости квалифицирует имя члена вместе с именем его класса.

Оператор *"::"* называется *оператором разрешения области видимост*и. По сути, он сообщает компилятору, что данная версия функции *qput()* принадлежит классу *queue*. Другими словами, оператор *"::"* заявляет о том, что функция *qput()* находится в области видимости класса *queue*. Различные классы могут использовать одинаковые имена функций. Компилятор же определит, к какому классу принадлежит функция, с помощью оператора разрешения области видимости и имени класса.

Функции-члены можно вызывать только относительно заданного объекта. Чтобы вызвать функцию-член из части программы, которая находится вне класса, необходимо использовать имя объекта и оператор *"точка"*. Например, при выполнении этого кода будет вызвана функция *init()* для объекта *ob1*.

```
queue ob1, ob2;
ob1.init();
```

При вызове функции *ob1.init()* действия, определенные в функции *init()*, будут направлены на копии данных, относящиеся к объекту *ob1*. Следует иметь в виду, что *ob1* и *ob2* — это два отдельных объекта. Это означает, что, например, инициализация объекта *ob1* отнюдь не приводит к инициализации объекта *ob2*. Единственное, что связывает объекты *ob1* и *ob2*, состоит в том, что они имеют один и тот же тип.

Если одна функция-член вызывает другую функцию-член того же класса, не нужно указывать имя объекта и использовать оператор *"точка"*. В этом случае компилятор уже точно знает, какой объект подвергается обработке. Имя объекта и оператор *"точка"* необходимы только тогда, когда функция-член вызывается кодом, расположенным вне класса. По тем же причинам функция-член может непосредственно обращаться к любому члену данных своего класса, но код, расположенный вне класса, должен обращаться к переменной класса, используя имя объекта и оператор *"точка"*.

В приведенной ниже программе класс *queue* иллюстрируется полностью (для этого объединены все уже знакомые вам части кода и добавлены недостающие детали).

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

// Создание класса queue.
class queue {
    int q[100];
    int sloc, rloc;
  public:
```

```
  .
    void init();
    void qput(int i);
    int qget();
};

// Инициализация класса queue.
void queue::init()
{
  rloc = sloc = 0;
}

// Занесение в очередь целочисленного значения.
void queue::qput(int i)
{
  if(sloc==100) {
    cout << "Очередь заполнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}

// Извлечение из очереди целочисленного значения.
int queue::qget()
{
  if(rloc == sloc) {
    cout << "Очередь пуста.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}

int main()
{
  queue a, b; // Создание двух объектов класса queue.
  a.init();
  b.init();

  a.qput(10);
  b.qput(19);

  a.qput(20);
  b.qput(1);

  cout << "Содержимое очереди a: ";
```

```
  cout << "Содержимое очереди a: ";
  cout << a.qget() << " ";
  cout << a.qget() << "\n";

  cout << "Содержимое очереди b: ";
  cout << b.qget() << " ";
  cout << b.qget() << "\n";

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Содержимое очереди a: 10 20
Содержимое очереди b: 19 1
```

Не забывайте, что закрытые члены класса доступны только функциям, которые являются членами этого класса. Например, такую инструкцию

```
a.rloc = 0;
```

нельзя включить в функцию main() нашей программы.

Общий формат объявления класса

Все классы объявляются подобно приведенному выше классу *queue*. Общий формат объявления класса имеет следующий вид.

```
class имя_класса {
    закрытые данные и функции
  public:
    открытые данные и функции
} список_объектов;
```

Здесь элемент *имя_класса* означает имя класса. Это имя становится именем нового типа, которое можно использовать для создания объектов класса. Объекты класса можно создать путем указания их имен непосредственно за закрывающейся фигурной скобкой объявления класса (в качестве элемента *список_объектов*), но это необязательно. После объявления класса его элементы можно создавать по мере необходимости.

Доступ к членам класса

Получение доступа к членам класса — вот что часто приводит в замешательство начинающих программистов. Поэтому остановимся на этой теме подробнее. Итак, рассмотрим следующий простой класс.

```
// Демонстрация доступа к членам класса.
#include <iostream>
using namespace std;

class myclass {
    int a; // закрытые данные
  public:
    int b; // открытые данные
    void setab(int i); // открытые функции
    int geta();
    void reset();
```

```
  void reset();
};

void myclass::setab(int i)
{
  a = i; // прямое обращение к переменной a
  b = i*i; // прямое обращение к переменной b
}

int myclass::geta()
{
  return a; // прямое обращение к переменной a
}

void myclass::reset()
{
// Прямой вызов функции setab()
setab(0); // для уже известного объекта.
}

int main()
{
  myclass ob;

  ob.setab(5); // Устанавливаем члены данных ob.a и ob.b.
  cout << "Объект ob после вызова функции setab(5): ";
  cout << ob.geta() << ' ';
  cout << ob.b; // К члену b можно получить прямой доступ, поскольку он является public-членом.
  cout << '\n';

  ob.b = 20; // Член b можно установить напрямую, поскольку он является public-членом.
  cout << "Объект ob после установки члена ob.b=20: ";
  cout << ob.geta() <<' ';
  cout << ob.b;
  cout << '\n';

  ob.reset();

  cout << "Объект ob после вызова функции ob.reset(): ";
  cout << ob.geta() << ' ';
  cout << ob.b;
  cout << '\n';

  return 0;
}
```

При выполнении этой программы получаем следующие результаты.

```
Объект ob после вызова функции setab(5): 5 25
Объект ob после установки члена ob.b=20: 5 20
Объект ob после вызова функции ob.reset(): 0 0
```

Теперь рассмотрим, как осуществляется доступ к членам класса *myclass*. Прежде всего обратите внимание на то, что для присвоения значений переменным *a* и *b* в функции *setab()* используются следующие строки кода.

```
a = i; // прямое обращение к переменной a
b = i*i; // прямое обращение к переменной b
```

Поскольку функция *setab()* является членом класса, она может обращаться к членам данных *a* и *b* того же класса непосредственно, без явного указания имени объекта (и не используя оператор *"точка"*). Как упоминалось выше, функция-член всегда вызывается для определенного объекта (а коль вызов состоялся, объект, стало быть, известен). Таким образом, в теле функции-члена нет необходимости указывать объект вторично. Следовательно, ссылки на переменные *a* и *b* будут применяться к копиям этих переменных, относящимся к вызывающему объекту.

Теперь обратите внимание на то, что переменная *b* — открытый (*public*) член класса *myclass*. Это означает, что к *b* можно получить доступ из кода, определенного вне тела класса *myclass*. Следующая строка кода из функции *main()*, при выполнении которой переменной *b* присваивается число *20*, демонстрирует реализацию такого прямого доступа.

```
ob.b = 20; // К члену b можно получить прямой доступ.
// поскольку он является public-членом.
```

Поскольку эта инструкция не принадлежит телу класса *myclass*, то к переменной *b* возможен доступ только с использованием конкретного объекта (в данном случае объекта *ob*) и оператора *"точка"*.

Теперь обратите внимание на то, как вызывается функция-член *reset()* из функции main().

```
ob.reset();
```

Поскольку функция *reset()* является открытым членом класса, ее также можно вызвать из кода, определенного вне тела класса *myclass*, и посредством конкретного объекта (в данном случае объекта *ob*).

Наконец, рассмотрим код функции *reset()*. Тот факт, что она является функцией-членом, позволяет ей непосредственно обращаться к другим членам того же класса, не используя оператор *"точка"* или конкретный объект. В данном случае она вызывает функцию-член *setab()*. И снова-таки, поскольку объект уже известен (он используется для вызова функции *reset()*), нет никакой необходимости указывать его еще раз.

Здесь важно понять следующее: когда доступ к некоторому члену класса происходит извне этого класса, его необходимо квалифицировать (уточнить) с помощью имени конкретного объекта. Но код самой функции-члена может обращаться к другим членам того же класса напрямую.

На заметку. *Не стоит волноваться, если вы еще не почувствовали в себе уверенность в вопросах получения доступа к членам класса. Небольшое беспокойство при освоении этой темы — обычное явление для начинающих программистов. Смело продолжайте читать книгу, рассматривая как можно больше примеров, и тема доступа к членам класса вскоре станет такой же простой, как таблица умножения!*

Конструкторы и деструкторы

Конструктор — *это функция, которая вызывается при создании объекта.*

Как правило, некоторую часть объекта, прежде чем его можно будет использовать, необходимо инициализировать. Например, рассмотрим класс *queue* (он представлен выше в этой главе). Прежде чем класс *queue* можно будет использовать, переменным *rloc* и *sloc* нужно присвоить нулевые значения. В данном конкретном случае это требование

нужно присвоить нулевые значения. В данном конкретном случае это требование выполнялось с помощью функции *init()*. Но, поскольку требование инициализации членов класса весьма распространено, в C++ предусмотрена реализация этой возможности при создании объектов класса. Такая автоматическая инициализация выполняется благодаря использованию конструктора.

Конструктор — это специальная функция, которая является членом класса и имя которой совпадает с именем класса. Вот, например, как стал выглядеть класс *queue* после переделки, связанной с применением конструктора для инициализации его членов.

```
// Определение класса queue.
class queue {
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    void qput(int i);
    int qget();
};
```

Обратите внимание на то, что в объявлении конструктора *queue()* отсутствует тип возвращаемого значения. В C++ конструкторы не возвращают значений и, следовательно, нет смысла в указании их типа. (При этом нельзя указывать даже тип *void*.)

Теперь приведем код функции *queue()*.

```
// Определение конструктора.
queue::queue()
{
  sloc = rloc = 0;
  cout << "Очередь инициализирована.\n";
}
```

В данном случае при выполнении конструктора выводится сообщение *Очередь инициализирована*., которое служит исключительно иллюстративным целям. На практике же в большинстве случаев конструкторы не выводят никаких сообщений.

Конструктор объекта вызывается при создании объекта. Это означает, что он вызывается при выполнении инструкции объявления объекта. Конструкторы глобальных объектов вызываются в самом начале выполнения программы, еще до обращения к функции main(). Что касается локальных объектов, то их конструкторы вызываются каждый раз, когда встречается объявление такого объекта.

Деструктор — *это функция*, *которая вызывается при разрушении объекта*.

Дополнением к *конструктору* служит *деструктор*. Во многих случаях при разрушении объекту необходимо выполнить некоторое действие или даже некоторую последовательность действий. Локальные объекты создаются при входе в блок, в котором они определены, и разрушаются при выходе из него. Глобальные объекты разрушаются при завершении программы. Существует множество факторов, обуславливающих необходимость деструктора. Например, объект должен освободить ранее выделенную для него память. В C++ именно деструктору поручается обработка процесса дезактивизации объекта. Имя деструктора совпадает с именем конструктора, но предваряется символом *"~"* Подобно конструкторам деструкторы не возвращают значений, а следовательно, в их объявлениях отсутствует тип возвращаемого значения.

Рассмотрим уже знакомый нам класс *queue*, но теперь он содержит конструктор и

деструктор. (Справедливости ради отметим, что классу *queue* деструктор, по сути, не нужен, а его наличие здесь можно оправдать лишь иллюстративными целями.)

```
// Определение класса queue.
class queue {
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};

// Определение конструктора.
queue::queue()
{
  sloc = rloc = 0;
  cout << "Очередь инициализирована.\n";
}

// Определение деструктора.
queue::~queue()
{
  cout << "Очередь разрушена.\n";
}
```

Вот как выглядит новая версия программы реализации очереди, в которой демонстрируется использование конструктора и деструктора.

```
// Демонстрация использования конструктора и деструктора.
#include <iostream>
using namespace std;

// Определение класса queue.
class queue {
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};

// Определение конструктора.
queue::queue()
{
```

```
{
  sloc = rloc = 0;
  cout << "Очередь инициализирована.\n";
}

// Определение деструктора.
queue::~queue()
{
  cout << "Очередь разрушена.\n";
}

// Занесение в очередь целочисленного значения.
void queue::qput(int i)
{
  if(sloc==100) {
    cout << "Очередь заполнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}

// Извлечение из очереди целочисленного значения.
int queue::qget()
{
  if(rloc == sloc) {
    cout << "Очередь пуста.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}

int main()
{
  queue a, b; // Создание двух объектов класса queue.

  a.qput(10);
  b.qput(19);
  a.qput(20);
  b.qput(1);

  cout << a.qget() << " ";
  cout << a.qget() << "\n";
  cout << b.qget() << " ";
  cout << b.qget() << "\n";
```

```
  cout << b.qget() << "\n";

  return 0;
}
```

При выполнении этой программы получаются такие результаты.

```
Очередь инициализирована.
Очередь инициализирована.
10 20
19 1
Очередь разрушена.
Очередь разрушена.
```

Параметризованные конструкторы

Конструктор может иметь параметры. С их помощью при создании объекта членам данных (переменным класса) можно присвоить некоторые начальные значения, определяемые в программе. Это реализуется путем передачи аргументов конструктору объекта. В следующем примере мы усовершенствуем класс *queue* так, чтобы он принимал аргументы, которые будут служить идентификационными номерами (*ID*) очереди. Прежде всего необходимо внести изменения в определение класса *queue*. Теперь оно выглядит так.

```
// Определение класса queue.
class queue {
    int q[100];
    int sloc, rloc;
    int who; // содержит идентификационный номер очереди
  public:
    queue(int id); // параметризованный конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};
```

Переменная *who* используется для хранения идентификационного номера (*ID*) создаваемой программой очереди. Ее реальное значение определяется значением, передаваемым конструктору в качестве параметра *id*, при создании переменной типа *queue*. Конструктор *queue()* выглядит теперь следующим образом.

```
// Определение конструктора.
queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  cout << "Очередь " << who << " инициализирована.\n";
}
```

Чтобы передать аргумент конструктору, необходимо связать этот аргумент с объектом при объявлении объекта. C++ поддерживает два способа реализации такого связывания. Вот как выглядит первый способ.

```
queue a = queue (101);
```

В этом объявлении создается очередь с именем *a*, которой передается значение (идентификационный номер) *101*. Но эта форма (в таком контексте) используется редко, поскольку второй способ имеет более короткую запись и удобнее для использования. Во втором способе аргумент должен следовать за именем объекта и заключаться в круглые скобки. Например, следующая инструкция эквивалентна предыдущему объявлению,

```
queue a (101);
```

Это самый распространенный способ объявления параметризованных объектов. Опираясь на этот метод, приведем общий формат передачи аргументов конструкторам.

```
тип_класса имя_переменной(список_аргументов);
```

Здесь элемент *список_аргументов* представляет собой список разделенных запятыми аргументов, передаваемых конструктору.

На заметку. *Формально между двумя приведенными выше формами инициализации существует небольшое различие, которое вы поймете при дальнейшем чтении этой книги. Но это различие не влияет на результаты выполнения программ, представленных в этой главе.*

В следующей версии программы организации очереди демонстрируется использование параметризованного конструктора.

```
// Использование параметризованного конструктора.
#include <iostream>
using namespace std;

// Определение класса queue.
class queue {
    int q[100];
    int sloc, rloc;
    int who; // содержит идентификационный номер очереди
  public:
    queue(int id); // параметризованный конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};

// Определение конструктора.
queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  cout << "Очередь " << who << " инициализирована.\n";
}

// Определение деструктора.
queue::~queue()
{
  cout << "Очередь " << who << " разрушена.\n";
}
```

```
// Занесение в очередь целочисленного значения.
void queue::qput(int i)
{
  if(sloc==100) {
    cout << "Очередь заполнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}

// Извлечение из очереди целочисленного значения.
int queue::qget()
{
  if(rloc == sloc) {
    cout << "Очередь пуста.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}

int main()
{
  queue a(1), b(2); // Создание двух объектов класса queue.
  a.qput(10);
  b.qput(19);
  a.qput(20);
  return 0;
}
```

При выполнении эта версия программы генерирует такие результаты:

```
Очередь 1 инициализирована.
Очередь 2 инициализирована.
10 20 19 1
Очередь 2 разрушена.
Очередь 1 разрушена.
```

Как видно из кода функции main(), очереди, связанной с именем *a*, присваивается идентификационный номер *1*, а очереди, связанной с именем *b*, — идентификационный номер *2*.

Несмотря на то что в примере с использованием класса *queue* при создании объекта передается только один аргумент, в общем случае возможна передача двух аргументов и более. В следующем примере объектам типа *widget* передается два значения.

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

class widget {
   int i;
   int j;
  public:
   widget(int a, int b);
   void put_widget();
};

// Передаем 2 аргумента конструктору widget().
widget::widget(int a, int b)
{
  i = a; j = b;
}

void widget::put_widget()
{
  cout << i << " " << j << "\n";
}

int main()
{
  widget x(10, 20), y(0, 0);
  x.put_widget();
  y.put_widget();
  return 0;
}
```

Важно! *В отличие от конструкторов, деструкторы не могут иметь параметров. Причину понять нетрудно: не существует средств передачи аргументов объекту, который разрушается. Если же у вас возникнет та редкая ситуация, когда при вызове деструктора вашему объекту необходимо получить доступ к некоторым данным, определяемым только во время выполнения программы, создайте для этой цели специальную переменную. Затем непосредственно перед разрушением объекта установите эту переменную равной нужному значению.*

При выполнении эта программа отображает следующие результаты.

```
10 20
0 0
```

Альтернативный вариант инициализации объекта

Если конструктор принимает только один параметр, можно использовать альтернативный способ инициализации членов объекта. Рассмотрим следующую программу.

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

class myclass {
```

```
  int a;
 public:
  myclass(int x);
  int get_a();
};

myclass::myclass(int x)
{
 a = x;
}

int myclass::get_a()
{
 return a;
}

int main()
{
 myclass ob = 4; // вызов функции myclass(4)
 cout << ob.get_a();
 return 0;
}
```

Здесь конструктор для объектов класса *myclass* принимает только один параметр. Обратите внимание на то, как в функции main() объявляется объект *ob*. Для этого используется такой формат объявления:

```
myclass ob = 4;
```

В этой форме инициализации объекта число *4* автоматически передается параметру *x* при вызове конструктора *myclass()*. Другими словами, эта инструкция объявления обрабатывается компилятором так, как если бы она была записана следующим образом.

```
myclass ob = myclass(4);
```

В общем случае, если у вас есть конструктор, который принимает только один аргумент, для инициализации объекта вы можете использовать либо вариант *ob(x)*, либо вариант *ob=x*. Дело в том, что при создании конструктора с одним аргументом неявно создается преобразование из типа этого аргумента в тип этого класса.

Помните, что показанный здесь альтернативный способ инициализации объектов применяется только к конструкторам, которые имеют только один параметр.

Классы и структуры — родственные типы

Как упоминалось в предыдущей главе, в C++ структура также обладает объектно-ориентированными возможностями. В сущности, классы и структуры можно назвать близкими родственниками. За одним исключением, они взаимозаменяемы, поскольку структура также может включать данные и код, который манипулирует этими данными точно так же, как это может делать класс. Единственное различие между C++-структурой и C++-классом состоит в том, что по умолчанию члены класса являются закрытыми, а члены структуры — открытыми. В остальном же структуры и классы имеют одинаковое назначение. На самом деле в соответствии с формальным синтаксисом C++ объявление структуры создает тип класса.

Рассмотрим пример структуры со свойствами, подобными свойствам класса.

```
// Использование структуры для создания класса.
#include <iostream>
using namespace std;

struct cl {
   int get_i(); // Эти члены открыты (public)
   void put_i(int j); // по умолчанию.
 private:
   int i;
};

int cl::get_i()
{
  return i;
}

void cl::put_i(int j)
{
  i = j;
}

int main()
{
  cl s;
  s.put_i (10);
  cout << s.get_i();
  return 0;
}
```

В этой программе определяется тип структуры с именем *cl*, в которой функции-члены *get_i()* и *put_i()* являются открытыми (public), а член данных *i* — закрытым (private). Обратите внимание на то, что в структурах для объявления закрытых членов используется ключевое слово *private*.

Ключевое слово private используется для объявления закрытых членов класса.

В следующем примере показана эквивалентная программа, которая использует вместо типа *struct* тип *class*.

```
// Использование типа class вместо типа struct.
#include <iostream>
using namespace std;

class cl {
   int i; // закрытый член по умолчанию
 public:
   int get_i();
   void put_i(int j);
};
```

```
int cl::get_i()
{
  return i;
}

void cl::put_i(int j)
{
  i = j;
}

int main()
{
  cl s;
  s.put_i(10);
  cout << s.get_i();
  return 0;
}
```

Иногда C++-программисты к структурам, которые не содержат функции-члены, применяют термин POD-struct.

C++-программисты тип *class* используют главным образом для определения формы объекта, который содержит функции-члены, а тип *struct* — для создания объектов, которые содержат только члены данных. Иногда для описания структуры, которая не содержит функции-члены, используется аббревиатура *POD* (Plain Old Data).

Сравнение структур с классами

Тот факт, что и структуры, и классы обладают практически идентичными возможностями, создает впечатление избыточности. Многие новички в программировании на C++ недоумевают, почему в нем существует такое очевидное дублирование. Нередко приходится слышать предложения отказаться от ненужного ключевого слова (*class* или *struct*) и оставить только одно из них.

Ответ на эту цепь рассуждений лежит в происхождении языка C++ от C и намерении сохранить C++ совместимым снизу вверх с C. В соответствии с современным определением C++ стандартная C-структура одновременно является совершенно законной C++-структурой. В языке C, который не содержит ключевых слов *public* или *private*, все члены структуры являются открытыми. Вот почему и члены C++-структур по умолчанию являются открытыми (а не закрытыми). Поскольку конструкция типа *class* специально разработана для поддержки инкапсуляции, есть определенный смысл в том, чтобы по умолчанию ее члены были закрытыми. Следовательно, чтобы избежать несовместимости с языком C в этом вопросе, аспекты доступа, действующие по умолчанию, менять было нельзя, поэтому и решено было добавить новое ключевое слово. Но в перспективе можно говорить о более веской причине для отделения структур от классов. Поскольку тип *class* синтаксически отделен от типа *struct*, определение класса вполне открыто для эволюционных изменений, которые синтаксически могут оказаться несовместимыми с C-подобными структурами. Поскольку мы имеем дело с двумя отдельными типами, будущее направление развития языка C++ не обременяется *"моральными обязательствами"*, связанными с совместимостью с C-структурами.

Под "занавес" этой темы отметим следующее. Структура определяет тип класса. Следовательно, структура является классом. На этом настаивал создатель языка C++, Бьерн Страуструп. Он полагал, что если структура и классы будут более или менее эквивалентны, то переход от C к C++ станет проще. И история доказала его правоту!

Объединения и классы — родственные типы

Тот факт, что *структуры* и *классы* — родственны, обычно никого не удивляет; однако вы можете удивиться, узнав, что объединения также связаны с классами "близкими отношениями". Согласно определению C++ объединение — это, по сути, тот же класс, в котором все члены данных хранятся в одной и той же области. (Таким образом, объединение также определяет тип класса.) Объединение может содержать конструктор и деструктор, а также функции-члены. Конечно же, члены объединения по умолчанию открыты (public), а не закрыты (private).

Рассмотрим программу, в которой объединение используется для отображения символов, составляющих содержимое старшего и младшего байтов короткого целочисленного значения (предполагается, что короткие целочисленные значения занимают в памяти компьютера два байта).

```
// Создание класса на основе объединения.
#include <iostream>
using namespace std;

union u_type {
  u_type(short int a); // открытые члены по умолчанию
  void showchars();
  short int i;
  char ch[2];
};

// конструктор
u_type::u_type(short int a)
{
  i = a;
}

// Отображение символов, составляющих  значение типа short int.
void u_type::showchars()
{
  cout << ch[0] << " ";
  cout << ch[1] << "\n";
}

int main()
{
  u_type u(1000);
  u.showchars();
  return 0;
}
```

Подобно структуре, C++-объединение также произошло от своего C-предшественника. Но в C++ оно имеет более широкие "классовые" возможности. Однако лишь то, что C++ наделяет "свои" объединения более могучими средствами и большей степенью гибкости, не означает, что вы непременно должны их использовать. Если вас вполне устраивает объединение с традиционным стилем поведения, вы вольны именно таким его и использовать. Но в случаях, когда можно инкапсулировать данные объединения вместе с

функциями, которые их обрабатывают, все же стоит воспользоваться C++-возможностями, что придаст вашей программе дополнительные преимущества.

Встраиваемые функции

Прежде чем мы продолжим освоение класса, сделаем небольшое, но важное отступление. Оно не относится конкретно к объектно-ориентированному программированию, но является очень полезным средством C++, которое довольно часто используется в определениях классов. Речь идет о *встраиваемой*, или *подставляемой*, функции (*inline function*). Встраиваемой называется функция, код которой подставляется в то место строки программы, из которого она вызывается, т.е. вызов такой функции заменяется ее кодом. Существует два способа создания встраиваемой функции. Первый состоит в использовании модификатора *inline*. Например, чтобы создать встраиваемую функцию *f()*, которая возвращает *int*-значение и не принимает ни одного параметра, достаточно объявить ее таким образом.

```
inline int f( )
{
  // ...
}
```

Модификатор *inline* должен предварять все остальные аспекты объявления функции.

Встраиваемая функция — *это небольшая (по объему кода) функция, код которой подставляется вместо ее вызова*.

Причиной существования встраиваемых функций является эффективность. Ведь при каждом вызове обычной функции должна быть выполнена целая последовательность инструкций, связанных с обработкой самого вызова, включающего помещение ее аргументов в стек, и с возвратом из функции. В некоторых случаях значительное количество циклов центрального процессора используется именно для выполнения этих действий. Но если функция встраивается в строку программы, подобные системные затраты попросту отсутствуют, и общая скорость выполнения программы возрастает. Если же встраиваемая функция оказывается не такой уж маленькой, общий размер программы может существенно увеличиться. Поэтому лучше всего в качестве встраиваемых использовать только очень маленькие функции, а те, что побольше, — оформлять в виде обычных.

Продемонстрируем использование встраиваемой функции на примере следующей программы.

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

class cl {
    int i; // закрытый член по умолчанию
  public:
    int get_i( );
    void put_i(int j);
};

inline int cl::get_i( )
{
  return i;
}

inline void cl::put_i(int j)
```

```
{
  i = j;
}

int main()
{
  cl s;
  s.put_i(10);
  cout << s.get_i();
  return 0;
}
```

Здесь вместо вызова функций *get_i()* и *put_i()* подставляется их код. Так, в функции main() строка

```
s.put_i(10);
```

функционально эквивалентна следующей инструкции присваивания:

```
s.i = 10;
```

Поскольку переменная *i* по умолчанию закрыта в рамках класса *cl*, эта строка не может реально существовать в коде функции *main()*, но за счет встраивания функции *put_i()* мы достигли того же результата, одновременно избавившись от затрат системных ресурсов, связанных с вызовом функции.

Важно понимать, что в действительности использование модификатора *inline* является *запросом*, а не *командой*, по которой компилятор сгенерирует встраиваемый (*inline*-) код. Существуют различные ситуации, которые могут не позволить компилятору удовлетворить наш запрос. Вот несколько примеров.

■ Некоторые компиляторы не генерируют встраиваемый код, если соответствующая функция содержит цикл, конструкцию *switch* или инструкцию *goto*.

■ Чаще всего встраиваемыми не могут быть рекурсивные функции.

■ Как правило, встраивание "не проходит" для функций, которые содержат статические (static) переменные.

Узелок на память. *Ограничения на использование встраиваемых функций зависят от конкретной реализации системы, поэтому, чтобы узнать, какие ограничения имеют место в вашем случае, обратитесь к документации, прилагаемой к вашему компилятору.*

Использование встраиваемых функций в определении класса

Существует еще один способ создания встраиваемой функции. Он состоит в определении кода для функции-члена класса в самом объявлении класса. Любая функция, которая определяется в объявлении класса, автоматически становится встраиваемой. В этом случае необязательно предварять ее объявление ключевым словом *inline*. Например, предыдущую программу можно переписать в таком виде.

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

class cl {
   int i; // закрытый член по умолчанию
  public:
   // автоматически встраиваемые функции
   int get_i() { return i; }
```

```
    int get_i() { return i; }
    void put_i(int j) { i = j; }
};

int main()
{
  s.put_i(10);
  cout << s.get_i();
  return 0;
}
```

Здесь функции *get_i()* и *put_i()* определены в теле объявления класса *cl* и автоматически являются встраиваемыми.

Обратите внимание на то, как выглядит код функций, определенных *"внутри"* класса *cl*. Для очень небольших по объему функций такое представление кода отражает обычный стиль языка C++. Однако можно сформатировать эти функции и таким образом.

```
class cl {
    int i; // закрытый член по умолчанию
  public:
    // встраиваемые функции
    int get_i()
    {
      return i;
    }
    void put_i(int j)
    {
      i = j;
    }
};
```

В общем случае небольшие функции (как представленные в этом примере) определяются в объявлении класса. Это соглашение применяется и к остальным примерам данной книги.

Важно! *Определение небольших функций-членов в объявлении класса — обычная практика в C++-программировании. И дело даже не в средстве автоматического встраивания, а просто в удобстве. Вряд ли вы встретите в профессиональных программах, чтобы короткие функции-члены определялись вне их класса.*

Массивы объектов

Массивы объектов можно создавать точно так же, как создаются массивы значений других типов. Например, в следующей программе создается класс *display*, который содержит значения разрешения для различных режимов видеомонитора. В функции main() создается массив для хранения трех объектов типа *display*, а доступ к объектам, которые являются элементами этого массива, осуществляется с помощью обычной процедуры индексирования массива.

```
// Пример использования массива объектов.
#include <iostream>
using namespace std;

enum resolution {low, medium, high}
```

```
class display {
    int width;
    int height;
    resolution res;
  public:
    void set_dim(int w, int h) {width=w; height=h;}
    void get_dim(int &w, int &h) {w=width; h=height;}
    void set_res(resolution r) {res = r;}
    resolution get_res() {return res;}
};

char names[3][8] = {
  "низкий",
  "средний",
  "высокий",
};

int main()
{
  display display_mode[3];
  int i, w, h;

  display_mode[0].set_res(low);
  display_mode[0].set_dim(640, 480);

  display_mode[1].set_res(medium);
  display_mode[1].set_dim(800, 600);

  display_mode[2].set_res(high);
  display_mode[2].set_dim(1600, 1200);

  cout << "Возможные режимы отображения данных:\n\n";

  for(i=0; i<3; i++) {
    cout << names[display_mode[i].get_res()] << ":";
    display_mode[i].get_dim(w, h);
    cout << w << " x " << h << "\n";
  }
  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Возможные режимы отображения данных:
низкий: 640 x 480
средний: 800 x 600
высокий: 1600 x 1200
```

Обратите внимание на использование двумерного символьного массива *names* для

преобразования перечислимого значения в эквивалентную символьную строку. Во всех перечислениях, которые не содержат явно заданной инициализации, первая константа имеет значение *0*, вторая — значение *1* и т.д. Следовательно, значение, возвращаемое функцией *get_res()*, можно использовать для индексации массива *names*, что позволяет вывести на экран соответствующее название режима отображения.

Многомерные массивы объектов индексируются точно так же, как многомерные массивы значений других типов.

Инициализация массивов объектов

Если класс включает параметризованный конструктор, то массив объектов такого класса можно инициализировать. Например, в следующей программе используется параметризованный класс *samp* и инициализируемый массив *sampArray* объектов этого класса.

```
// Инициализация массива объектов.
#include <iostream>
using namespace std;

class samp {
   int a;
  public:
   samp(int n) { a = n; }
   int get_a() { return a; }
};

int main()
{
  samp sampArray[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i<4; i++) cout << sampArray[i].get_a() << ' ';
  cout << "\n";
  return 0;
}
```

Результаты выполнения этой программы

```
-1 -2 -3 -4
```

подтверждают, что конструктору *samp* действительно были переданы значения от *-1* до *-4*.

В действительности синтаксис инициализации массива, выраженный строкой

```
samp sampArray[4] = { -1, -2, -3, -4 };
```

представляет собой сокращенный вариант следующего (более длинного) формата:

```
samp sampArray[4] = { samp(-1), samp(-2), samp(-3), samp(-4) };
```

Формат инициализации, представленный в программе, используется программистами чаще, чем его более длинный эквивалент, однако следует помнить, что он работает для массивов таких объектов, конструкторы которых принимают только один аргумент. При инициализации массива объектов, конструкторы которых принимают несколько аргументов, необходимо использовать более длинный формат инициализации. Рассмотрим пример.

```
#include <iostream>
```

```
using namespace std;

class samp {
   int a, b;
  public:
   samp(int n, int m) { a = n; b = m; }
   int get_a() { return a; }
   int get_b() { return b; }
};

int main()
{
  samp sampArray[4][2] = {
    samp(1, 2),
    samp(3, 4),
    samp(5, 6),
    samp(7, 8),
    samp(9, 10),
    samp(11, 12),
    samp(13, 14),
    samp(15, 16)
  };
  int i;
  for(i=0; i<4; i++) {
    cout << sampArray[i][0].get_a() << ' ';
    cout << sampArray[i][0].get_b() << "\n";
    cout << sampArray[i][1].get_a() << ' ';
    cout << sampArray[i][1].get_b() << "\n";
  }
  cout << "\n";
  return 0;
}
```

В этом примере конструктор класса *samp* принимает два аргумента. В функции main() объявляется и инициализируется массив *sampArray* путем непосредственных вызовов конструктора *samp()*. Инициализируя массивы, можно всегда использовать длинный формат инициализации, даже если объект принимает только один аргумент (короткая форма просто более удобна для применения). Нетрудно проверить, что при выполнении эта программа отображает такие результаты.

```
1 2
3 4
5 6
7 8
9 10
11 12
```

```
11 12
13 14
15 16
```

Указатели на объекты

Как было показано в предыдущей главе, доступ к структуре можно получить напрямую или через указатель на эту структуру. Аналогично можно обращаться и к объекту: непосредственно (как во всех предыдущих примерах) или с помощью указателя на объект. Чтобы получить доступ к отдельному члену объекта исключительно *"силами"* самого объекта, используется оператор *"точка"*. А если для этого служит указатель на этот объект, необходимо использовать оператор *"стрелка"*. (Применение операторов *"точка"* и *"стрелка"* для объектов соответствует их применению для структур и объединений.)

Чтобы объявить указатель на объект, используется тот же синтаксис, как и в случае объявления указателей на значения других типов. В следующей программе создается простой класс *P_example*, определяется объект этого класса *ob* и объявляется указатель на объект типа *P_example* с именем *p*. В этом примере показано, как можно напрямую получить доступ к объекту *ob* и как использовать для этого указатель (в этом случае мы имеем дело с косвенным доступом).

```
// Простой пример использования указателя на объект.
#include <iostream>
using namespace std;

class P_example {
    int num;
  public:
    void set_num(int val) {num = val;}
    void show_num();
};

void P_example::show_num()
{
  cout << num << "\n";
}

int main()
{
  P_example ob, *p; // Объявляем объект и  указатель на него.
  ob.set_num(1); // Получаем прямой доступ к объекту ob.
  ob.show_num();
  p = &ob; // Присваиваем указателю p адрес объекта ob.
  p->show_num(); // Получаем доступ к объекту ob с помощью указателя.
  return 0;
}
```

Обратите внимание на то, что адрес объекта *ob* получается путем использования оператора что соответствует получению адреса для переменных любого другого типа.

Как вы знаете, при инкрементации или декрементации указателя он инкрементируется или декрементируется так, чтобы всегда указывать на следующий или предыдущий элемент базового типа. То же самое происходит и при инкрементации или декрементации указателя

базового типа. То же самое происходит и при инкрементации или декрементации указателя на объект: он будет указывать на следующий или предыдущий объект. Чтобы проиллюстрировать этот механизм, модифицируем предыдущую программу. Теперь вместо одного объекта *ob* объявим двухэлементный массив *ob* типа *P_example*. Обратите внимание на то, как инкрементируется и декрементируется указатель *p* для доступа к двум элементам этого массива.

```
// Инкрементация и декрементация указателя на объект.
#include <iostream>
using namespace std;

class P_example {
    int num;
  public:
    void set_num(int val) {num = val;}
    void show_num();
};

void P_example::show_num()
{
  cout << num << "\n";
}

int main()
{
  P_example ob[2], *p;

  ob[0].set_num(10); // прямой доступ к объектам
  ob[1].set_num(20);

  p = &ob[0]; // Получаем указатель на первый элемент.
   p->show_num(); // Отображаем значение элемента ob[0] с помощью указателя.

  p++; // Переходим к следующему объекту.
   p->show_num(); // Отображаем значение элемента ob[1] с помощью указателя.

  p--; // Возвращаемся к предыдущему объекту.
  p->show_num(); // Снова отображаем значение элемента ob[0].
  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
10
20
10
```

Как будет показано ниже в этой книге, указатели на объекты играют главную роль в реализации одного из важнейших принципов C++: полиморфизма.

Ссылки на объекты

На объекты можно ссылаться таким же образом, как и на значения любого другого типа. Для

На объекты можно ссылаться таким же образом, как и на значения любого другого типа. Для этого не существует никаких специальных инструкций или ограничений. Но, как будет показано в следующей главе, использование ссылок на объекты позволяет справляться с некоторыми специфическими проблемами, которые могут встретиться при использовании классов.

Глава 12: О классах подробнее

В этой главе мы продолжим рассмотрение классов, начатое в главе 11. Здесь вы познакомитесь с "дружественными" функциями, перегрузкой конструкторов, а также с возможностью передачи и возвращения объектов функциями. Кроме того, вы узнаете о существовании специального типа конструктора, именуемого конструктором копии, который используется в случае, когда возникает необходимость в создании копии объекта. Завершает главу описание ключевого слова *this*.

Функции-"друзья"

В C++ существует возможность разрешить доступ к закрытым членам класса функциям, которые не являются членами этого класса. Для этого достаточно объявить эти функции *"дружественными"* (или *"друзьями"*) по отношению к рассматриваемому классу. Чтобы сделать функцию *"другом"* класса, включите ее прототип в *public*-раздел объявления класса и предварите его ключевым словом *friend*. Например, в этом фрагменте кода функция *frnd()* объявляется *"другом"* класса *cl*.

```
class cl {
   // . . .
  public:
   friend void frnd(cl ob);
   // . . .
};
```

Ключевое слово friend предоставляет функции, которая не является членом класса, доступ к его закрытым членам.

Как видите, ключевое слово *friend* предваряет остальную часть прототипа функции. Функция может быть *"другом"* нескольких классов.

Рассмотрим короткий пример, в котором функция-*"друг"* используется для доступа к закрытым членам класса *myclass*.

```
// Демонстрация использования функции-"друга".
#include <iostream>
using namespace std;

class myclass {
   int a, b;
  public:
   myclass(int i, int j) { a=i; b=j; }
   friend int sum(myclass x); // Функция sum() - "друг" класса myclass.
};

// Обратите внимание на то, что функция sum() не является членом ни одного класса
int sum(myclass x)
{
  /* Поскольку функция sum() — "друг" класса myclass, она имеет право на
```

```
  // Поскольку функция sum() - друг класса myclass, она имеет право на
прямой доступ к его членам данных a и b. */
  return x.a + x.b;
}

int main ()
{
  myclass n (3, 4);
  cout << sum(n);
  return 0;
}
```

В этом примере функция *sum()* не является членом класса *myclass*. Тем не менее она имеет полный доступ к *private*-членам класса *myclass*. В частности, она может непосредственно использовать значения *x.a* и *x.b*. Обратите также внимание на то, что функция *sum()* вызывается обычным образом, т.е. без привязки к объекту (и без использования оператора *"точка"*). Поскольку она не функция-член, то при вызове ее не нужно квалифицировать с указанием имени объекта. (Точнее, при ее вызове нельзя задавать имя объекта.) Обычно функции-*"другу"* в качестве параметра передается один или несколько объектов класса, для которого она является *"другом"*, как в случае функции *sum()*.

Несмотря на то что в данном примере мы не извлекаем никакой пользы из объявления функции *sum() "другом"*, а не членом класса *myclass*, существуют определенные обстоятельства, при которых статус функции-*"друга"* имеет большое значение. Во-первых, функции-*"друзья"* могут быть полезны для перегрузки операторов определенных типов. Во-вторых, функции-*"друзья"* упрощают создание некоторых функций ввода-вывода. Об этом речь впереди.

Третья причина использования функций-*"друзей"* состоит в том, что в некоторых случаях два (или больше) класса могут содержать члены, которые находятся во взаимной связи с другими частями программы. Например, у нас есть два различных класса, которые при возникновении определенных событий отображают на экране "всплывающие" сообщения. Другие части программы, которые предназначены для вывода данных на экран, должны знать, является ли "всплывающее" сообщение активным, чтобы случайно не перезаписать его. В каждом классе можно создать функцию-член, возвращающую значение, по которому можно судить о том, активно сообщение или нет; однако проверка этого условия потребует дополнительных затрат (т.е. двух вызовов функций вместо одного). Если статус "всплывающего" сообщения необходимо проверять часто, эти дополнительные затраты могут оказаться попросту неприемлемыми. Однако с помощью функции, "дружественной" для обоих классов, можно напрямую проверять статус каждого объекта, вызывая только одну функцию, которая будет иметь доступ к обоим классам. В подобных ситуациях функция-*"друг"* позволяет написать более эффективный код. Эта идея иллюстрируется на примере следующей программы.

```
// Использование функции-"друга".
#include <iostream>
using namespace std;

const int IDLE=0;
const int INUSE=1;
class C2; // опережающее объявление
class C1 {
    int status; // IDLE если сообщение неактивно, INUSE если сообщение
выведено на экран.
```

```
  // ...
 public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
   int status; // IDLE если сообщение неактивно, INUSE если сообщение выведено на экран.
  // ...
 public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

void C1::set_status(int state)
{
 status = state;
}

void C2::set_status(int state)
{
 status = state;
}

// Функция idle() - "друг" для классов C1 и C2.
int idle(C1 a, C2 b)
{
 if(a.status || b.status) return 0;
 else return 1;
}

int main()
{
 C1 x;
 C2 y;

 x.set_status(IDLE);
 y.set_status(IDLE);

 if(idle(x, y)) cout << "Экран свободен.\n";
 else cout << "Отображается сообщение.\n";

 x.set_status(INUSE);

 if(idle(x, y)) cout << "Экран свободен.\n";
 else cout << "Отображается сообщение.\n";
```

```
  return 0;
}
```

При выполнении программа генерирует такие результаты.

```
Экран свободен.
Отображается сообщение.
```

Поскольку функция *idle()* является *"другом"* как для класса *C1*, так и для класса *C2*, она имеет доступ к закрытому члену *status*, определенному в обоих классах. Таким образом, состояние объекта каждого класса одновременно можно проверить всего одним обращением к функции *idle()*.

Опережающее объявление предназначено для объявления имени классового типа до определения самого класса.

Обратите внимание на то, что в этой программе используется *опережающее объявление* (также именуемое *опережающей ссылкой*) для класса *C2*. Его необходимость обусловлена тем, что объявление функции *idle()* в классе *C1* использует ссылку на класс *C2* до его объявления. Чтобы создать опережающее объявление для класса, достаточно использовать формат, представленный в этой программе.

"Друг" одного класса может быть членом другого класса. Перепишем предыдущую программу так, чтобы функция *idle()* стала членом класса *C1*. Обратите внимание на использование оператора разрешения области видимости (или оператора разрешения контекста) при объявлении функции *idle()* в качестве *"друга"* класса *C2*.

```
/* Функция может быть членом одного класса и одновременно "другом"
другого.
*/
#include <iostream>
using namespace std;

const int IDLE=0;
const int INUSE=1;

class C2; // опережающее объявление

class C1 {
    int status; // IDLE, если сообщение неактивно, INUSE, если сообщение
выведено на экран.
   // ...
  public:
   void set_status(int state);
   int idle(C2 b); // теперь это член класса C1
};

class C2 {
    int status; // IDLE, если сообщение неактивно, INUSE, если сообщение
выведено на экран.
   // . . .
  public:
   void set_status(int state);
   friend int C1::idle(C2 b); // функция-"друг"
};
```

```
void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

// Функция idle( ) -- член класса C1 и "друг" класса C2.
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main( )
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Экран свободен.\n";
  else cout << "Отображается сообщение.\n";

  x.set_status(INUSE);

  if(x.idle(y)) cout << "Экран свободен.\n";
  else cout << "Отображается сообщение.\n";

  return 0;
}
```

Поскольку функция *idle()* является членом класса *C1*, она имеет прямой доступ к переменной *status* объектов типа *C1*. Следовательно, в качестве параметра необходимо передавать функции *idle()* только объекты типа *C2*.

Перегрузка конструкторов

Несмотря на выполнение конструкторами уникальных действий, они не сильно отличаются от функций других типов и также могут подвергаться перегрузке. Чтобы перегрузить конструктор класса, достаточно объявить его во всех нужных форматах и определить каждое действие, связанное с соответствующим форматом. Например, в следующей программе объявляется класс *timer*, который действует как вычитающий таймер. При создании объекта типа *timer* таймеру присваивается некоторое начальное значение времени. При вызове функции *run()* таймер выполняет счет в обратном порядке до нуля, а затем подает звуковой сигнал. В этом примере конструктор перегружается трижды, предоставляя тем самым возможность задавать время как в секундах (причем либо числом,

либо строкой), так и в минутах и секундах (с помощью двух целочисленных значений). В этой программе используется стандартная библиотечная функция *clock()*, которая возвращает количество сигналов, принятых от системных часов с момента начала выполнения программы. Вот как выглядит прототип этой функции:

```
clock_t clock( );
```

Тип *clock_t* представляет собой разновидность длинного целочисленного типа. Операция деления значения, возвращаемого функцией *clock()*, на значение *CLOCKS_PER_SEC* позволяет преобразовать результат в секунды. Как прототип для функции *clock()*, так и определение константы *CLOCKS_PER_SEC* принадлежат заголовку *<ctime>*.

```
// Использование перегруженных конструкторов.
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

class timer{
    int seconds;
  public:
    // секунды, задаваемые в виде строки
    timer(char *t) { seconds = atoi (t); }

    // секунды, задаваемые в виде целого числа
    timer(int t) { seconds = t; }

    // время, задаваемое в минутах и секундах
    timer(int min, int sec) { seconds = min*60 + sec; }

    void run( );
};

void timer::run( )
{
  clock_t t1;
  t1 = clock( );
  while( (clock( )/CLOCKS_PER_SEC - t1/CLOCKS_PER_SEC)<seconds);
  cout << "\a"; // звуковой сигнал
}

int  main( )
{
  timer a (10), b("20"), c(1, 10);

  a.run( ); // отсчет 10 секунд
  b.run( ); // отсчет 20 секунд
  c.run( ); // отсчет 1 минуты и 10 секунд

  return 0;
}
```

При создании в функции main() объектов *a*, *b* и *c* класса *timer* им присваиваются начальные значения тремя различными способами, поддерживаемыми перегруженными функциями конструкторов. В каждом случае вызывается конструктор, который соответствует заданному списку параметров и потому надлежащим образом инициализирует *"свой"* объект.

На примере предыдущей программы вы, возможно, не оценили значимость перегрузки функций конструктора, поскольку здесь можно было обойтись единым способом задания временного интервала. Но если бы вы создавали библиотеку классов на заказ, то вам стоило бы предусмотреть набор конструкторов, охватывающий самый широкий спектр различных форматов инициализации, тем самым обеспечив других программистов наиболее подходящими для их программ форматами. Кроме того, как будет показано ниже, в C++ существует атрибут, который делает перегруженные конструкторы особенно ценным средством инициализации объектов.

Динамическая инициализация

В C++ как локальные, так и глобальные переменные можно инициализировать во время выполнения программы. Этот процесс иногда называют *динамической инициализацией*. До сих пор в большинстве инструкций инициализации, представленных в этой книге, использовались константы. Однако переменную можно также инициализировать во время выполнения программы, используя любое C++-выражение, действительное на момент объявления этой переменной. Это означает, что переменную можно инициализировать с помощью других переменных и/или вызовов функций при условии, что в момент выполнения инструкции объявления общее выражение инициализации имеет действительное значение. Например, следующие варианты инициализации переменных абсолютно допустимы в C++,

```
int n = strlen(str);
double arc = sin(theta);
float d = 1.02 * count / deltax;
```

Применение динамической инициализации к конструкторам

Подобно простым переменным, объекты можно инициализировать динамически при их создании. Это средство позволяет создавать объект нужного типа с использованием информации, которая становится известной только во время выполнения программы. Чтобы показать, как работает механизм динамической инициализации, модифицируем программу реализации таймера, приведенную в предыдущем разделе.

Вспомните, что в первом примере программы таймера мы не получили большого преимущества от перегрузки конструктора *timer()*, поскольку все объекты этого типа инициализировались с помощью констант, известных во время компиляции программы. Но в случаях, когда объект необходимо инициализировать во время выполнения программы, можно получить существенный выигрыш от наличия множества разных форматов инициализации. Это позволяет программисту выбрать из существующих конструкторов тот, который наиболее точно соответствуют текущему формату данных.

Например, в следующей версии программы таймера для создания двух объектов *b* и *c* используется динамическая инициализация.

```
// Демонстрация динамической инициализации.
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

class timer{
    int seconds;
```

```
  public:
    // секунды, задаваемые в виде строки
    timer(char *t) { seconds = atoi(t); }

    // секунды, задаваемые в виде целого числа
    timer(int t) { seconds = t; }

    // время, задаваемое в минутах и секундах
    timer(int min, int sec) { seconds = min*60 + sec; }

    void run();
};

void timer::run()
{
  clock_t t1;
  t1 = clock();
  while((clock()/CLOCKS_PER_SEC - t1/CLOCKS_PER_SEC)<seconds);
  cout << "\a"; // звуковой сигнал
}

int main()
{
  timer a(10);
  a.run();

  cout << "Введите количество секунд: ";
  char str[80];
  cin >> str;
  timer b(str); // инициализация в динамике
  b.run();

  cout << "Введите минуты и секунды: ";
  int min, sec;
  cin >> min >> sec;
  timer c(min, sec); // инициализация в динамике
  c.run();

  return 0;
}
```

Как видите, объект *a* создается с использованием целочисленной константы. Однако основой для создания объектов *b* и *c* служит информация, вводимая пользователем. Поскольку для объекта *b* пользователь вводит строку, имеет смысл перегрузить конструктор *timer()* для приема строк. Объект *c* также создается во время выполнения программы с использованием данных, вводимых пользователем. Поскольку в этом случае время вводится в виде минут и секунд, для построения объекта *c* логично использовать формат конструктора, принимающего два аргумента. Трудно не согласиться с тем, что наличие множества форматов инициализации избавляет программиста от выполнения дополнительных преобразований при инициализации объектов.

Механизм перегрузки конструкторов способствует понижению уровня сложности программ, позволяя создавать объекты наиболее естественным для их применения образом. Поскольку существует три наиболее распространенных способа передачи объекту значений временных интервалов, имеет смысл позаботиться о том, чтобы конструктор *timer()* был перегружен для реализации каждого из этих способов. При этом перегрузка конструктора *timer()* для приема значения, выраженного в днях или наносекундах, вряд ли себя оправдает. Загромождение кода конструкторами для обработки редко возникающих ситуаций оказывает, как правило, дестабилизирующее влияние на программу.

Узелок на память. *Разрабатывая перегруженные конструкторы, необходимо определиться в том, какие ситуации важно предусмотреть, а какие можно и не учитывать*.

Присваивание объектов

Если два объекта имеют одинаковый тип (т.е. оба они — объекты одного класса), то один объект можно присвоить другому. Для присваивания недостаточно, чтобы два класса были физически подобны; имена классов, объекты которых участвуют в операции присваивания, должны совпадать. Если один объект присваивается другому, то по умолчанию данные первого объекта поразрядно копируются во второй. Присваивание объектов демонстрируется в следующей программе.

```
// Демонстрация присваивания объектов.
#include <iostream>
using namespace std;

class myclass {
   int a, b;
  public:
   void setab(int i, int j) { a = i, b = j; }
   void showab();
};

void myclass::showab()
{
  cout << "a равно " << a << '\n';
  cout << "b равно " << b << '\n';
}

int main()
{
  myclass ob1, ob2;

  ob1.setab(10, 20);
  ob2.setab(0, 0);

  cout << "Объект ob1 до присваивания: \n";
  ob1.showab();
  cout << "Объект ob2 до присваивания: \n";
  ob2.showab();

  cout << ' \n';
```

```
  ob2 = ob1; // Присваиваем объект ob1 объекту ob2.

  cout << "Объект ob1 после присваивания: \n";
  ob1.showab();
  cout << "Объект ob2 после присваивания: \n";
  ob2.showab();

  return 0;
}
```

При выполнении программа генерирует такие результаты.

```
Объект ob1 до присваивания:
a равно 10
b равно 20
Объект ob2 до присваивания:
a равно 0
b равно 0
Объект ob1 после присваивания:
a равно 10
b равно 20
Объект ob2 после присваивания:
a равно 10
b равно 20
```

По умолчанию все данные из одного объекта присваиваются другому путем создания поразрядной копии. (Другими словами, создается точный дубликат объекта.) Но, как будет показано ниже, оператор присваивания можно перегрузить, определив собственные операции присваивания.

Узелок на память. *Присваивание одного объекта другому просто делает их данные идентичными, но эти два объекта остаются совершенно независимыми. Следовательно, последующая модификация данных одного объекта не оказывает никакого влияния на данные другого.*

Передача объектов функциям

Объект можно передать функции точно так же, как значение любого другого типа данных. Объекты передаются функциям путем использования обычного C++-соглашения о передаче параметров по значению. Таким образом, функции передается не сам объект, а его копия. Следовательно, изменения, внесенные в объект при выполнении функции, не оказывают никакого влияния на объект, используемый в качестве аргумента для функции. Этот механизм демонстрируется в следующей программе.

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

class OBJ {
   int i;
  public:
   void set_i(int x) { i = x; }
   void out_i() { cout << i << " "; }
```

```
};

void f(OBJ x)
{
  x.out_i(); // Выводит число.
  x.set_i(100); // Устанавливает только локальную копию.
  x.out_i(); // Выводит число 100.
}

int main()
{
  OBJ o;
  o.set_i(10);
  f(o);
  o.out_i(); // По-прежнему выводит число 10, значение переменной i не изменилось.

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
10 100 10
```

Как подтверждают эти результаты, модификация объекта *x* в функции *f()* не влияет на объект *o* в функции *main()*.

Конструкторы, деструкторы и передача объектов

Несмотря на то что передача функциям несложных объектов в качестве аргументов — довольно простая процедура, при этом могут происходить непредвиденные события, имеющие отношение к конструкторам и деструкторам. Чтобы разобраться в этом, рассмотрим следующую программу.

```
// Конструкторы, деструкторы и передача объектов.
#include <iostream>
using namespace std;

class myclass {
   int val;
  public:
   myclass(int i) { val = i; cout << "Создание\n"; }
   ~myclass() { cout << "Разрушение\n"; }
   int getval() { return val; }
};

void display(myclass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
```

```
{
  myclass a(10);
  display(a);
  return 0;
}
```

При выполнении эта программа выводит следующие неожиданные результаты.

```
Создание
10
Разрушение
Разрушение
```

Как видите, здесь выполняется одно обращение к функции конструктора (при создании объекта *a*), но почему-то два обращения к функции деструктора. Давайте разбираться, в чем тут дело.

При передаче объекта функции создается его копия (и эта копия становится параметром в функции). Создание копии означает "рождение" нового объекта. Когда выполнение функции завершается, копия аргумента (т.е. параметр) разрушается. Здесь возникает сразу два вопроса. Во-первых, вызывается ли конструктор объекта при создании копии? Во-вторых, вызывается ли деструктор объекта при разрушении копии? Ответы могут удивить вас.

Когда при вызове функции создается копия аргумента, обычный конструктор не вызывается. Вместо этого вызывается *конструктор копии* объекта. Конструктор копии определяет, как должна быть создана копия объекта. (Как создать конструктор копии, будет показано ниже в этой главе.) Но если в классе явно не определен конструктор копии, C++ предоставляет его по умолчанию. Конструктор копии по умолчанию создает побитовую (т.е. идентичную) копию объекта. Поскольку обычный конструктор используется для инициализации некоторых аспектов объекта, он не должен вызываться для создания копии уже существующего объекта. Такой вызов изменил бы его содержимое. При передаче объекта функции имеет смысл использовать текущее состояние объекта, а не его начальное состояние.

Но когда функция завершается и разрушается копия объекта, используемая в качестве аргумента, вызывается деструктор этого объекта. Необходимость вызова деструктора связана с выходом объекта из области видимости. Именно поэтому предыдущая программа имела два обращения к деструктору. Первое произошло при выходе из области видимости параметра функции *display()*, а второе— при разрушении объекта *a* в функции main() по завершении программы.

Итак, когда объект передается функции в качестве аргумента, обычный конструктор не вызывается. Вместо него вызывается конструктор копии, который по умолчанию создает побитовую (идентичную) копию этого объекта. Но когда эта копия разрушается (обычно при выходе за пределы области видимости по завершении функции), обязательно вызывается деструктор.

Потенциальные проблемы при передаче параметров

Несмотря на то что объекты передаются функциям *"по значению"*, т.е. посредством обычного C++-механизма передачи параметров, который теоретически защищает аргумент и изолирует его от принимаемого параметра, здесь все-таки возможен побочный эффект или даже угроза для "жизни" объекта, используемого в качестве аргумента. Например, если объект, используемый как аргумент, требует динамического выделения памяти и освобождает эту память при разрушении, его локальная копия при вызове деструктора освободит ту же самую область памяти, которая была выделена оригинальному объекту. И этот факт становится уже целой проблемой, поскольку оригинальный объект все еще использует эту (уже освобожденную) область памяти. Описанная ситуация делает исходный объект "ущербным" и, по сути, непригодным для использования. Рассмотрим следующую

простую программу.

```
// Демонстрация проблемы, возможной при передаче объектов функциям.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int *p;
  public:
   myclass(int i);
   ~myclass();
   int getval() { return *p; }
};

myclass::myclass(int i)
{
  cout << "Выделение памяти, адресуемой указателем p.\n";
  p = new int;
  *p = i;
}

myclass::~myclass()
{
  cout <<"Освобождение памяти, адресуемой указателем p.\n";
  delete p;
}

// При выполнении этой функции и возникает проблема.
void display(myclass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  myclass a(10);
  display(a);
  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Выделение памяти, адресуемой указателем p.
10
Освобождение памяти, адресуемой указателем p.
Освобождение памяти, адресуемой указателем p.
```

Эта программа содержит принципиальную ошибку. И вот почему: при создании в функции

Эта программа содержит принципиальную ошибку. Вот ее суть: при создании в функции *main()* объекта *a* выделяется область памяти, адрес которой присваивается указателю *a.p* . При передаче функции *display()* объект *a* копируется в параметр *ob*. Это означает, что оба объекта (*a* и *ob*) будут иметь одинаковое значение для указателя *p*.

Другими словами, в обоих объектах (в оригинале и его копии) член данных *p* будет указывать на одну и ту же динамически выделенную область памяти. По завершении функции *display()* объект *ob* разрушается, и его разрушение сопровождается вызовом деструктора. Деструктор освобождает область памяти, адресуемую указателем *ob.p*. Но ведь эта (уже освобожденная) область памяти — та же самая область, на которую все еще указывает член данных (исходного объекта) *a.p*! Налицо серьезная ошибка.

В действительности дела обстоят еще хуже. По завершении программы разрушается объект *a*, и динамически выделенная (еще при его создании) память освобождается вторично. Дело в том, что освобождение одной и той же области динамически выделенной памяти во второй раз считается неопределенной операцией, которая, как правило (в зависимости от того, как реализована система динамического распределения памяти), вызывает неисправимую ошибку.

Возможно, читатель уже догадался, что один из путей решения проблемы, связанной с разрушением (еще нужных) данных деструктором объекта, являющегося параметром функции, состоит не в передаче самого объекта, а в передаче указателя на него или ссылки. В этом случае копия объекта не создается; следовательно, по завершении функции деструктор не вызывается. Вот как выглядит, например, один из способов исправления предыдущей программы.

```
// Одно из решений проблемы передачи объектов.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int *p;
  public:
   myclass(int i);
   ~myclass();
   int getval() { return *p; }
};

myclass::myclass(int i)
{
  cout << "Выделение памяти, адресуемой указателем p.\n";
  p = new int;
  *p = i;
}

myclass::~myclass()
{
  cout <<"Освобождение памяти, адресуемой указателем p.\n";
  delete p;
}

/* Эта функция НЕ создает проблем. Поскольку объект ob теперь передается
```

```
/* Эта функция не создает проблем. Поскольку объект ob теперь передается
по ссылке, копия аргумента не создается, а следовательно, объект не
выходит из области видимости по завершении функции display().
*/
void display(myclass &ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  myclass a(10);
  display(a);
  return 0;
}
```

Результаты выполнения этой версии программы выглядят гораздо лучше предыдущих.

```
Выделение памяти, адресуемой указателем p.
10
Освобождение памяти, адресуемой указателем p.
```

Как видите, здесь деструктор вызывается только один раз, поскольку при передаче по ссылке аргумента функции *display()* копия объекта не создается.

Передача объекта по ссылке — прекрасное решение описанной проблемы, но только в случаях, когда ситуация позволяет принять его, что бывает далеко не всегда. К счастью, есть более общее решение: можно создать собственную версию конструктора копии. Это позволит точно определить, как именно следует создавать копию объекта и тем самым избежать описанных выше проблем. Но прежде чем мы займемся конструктором копии, имеет смысл рассмотреть еще одну ситуацию, в обработке которой мы также можем выиграть от создания *конструктора копии*.

Возвращение объектов функциями

Если объекты можно передавать функциям, то *"с таким же успехом"* функции могут возвращать объекты. Чтобы функция могла вернуть объект, во-первых, необходимо объявить в качестве типа возвращаемого ею значения тип соответствующего класса. Во-вторых, нужно обеспечить возврат объекта этого типа с помощью обычной инструкции *return*. Рассмотрим пример функции, которая возвращает объект.

```
// Использование функции, которая возвращает объект.
#include <iostream>
#include <cstring>
using namespace std;

class sample {
   char s[80];
  public:
   void show() { cout << s << "\n"; }
   void set(char *str) { strcpy(s, str); }
};

// Эта функция возвращает объект типа sample.
```

```
sample input()
{
  char instr[80];
  sample str;

  cout << "Введите строку: ";
    cin >> instr;

  str.set(instr);
  return str;
}

int main()
{
  sample ob;
  // Присваиваем объект, возвращаемый
  // функцией input(), объекту ob.
  ob = input();
  ob.show();
  return 0;
}
```

В этом примере функция *input()* создает локальный объект *str* класса *sample*, а затем считывает строку с клавиатуры. Эта строка копируется в строку *str.s*, после чего объект *str* возвращается функцией *input()* и присваивается объекту *ob* в функции *main()*.

Потенциальная проблема при возвращении объектов функциями

Относительно возвращения объектов функциями важно понимать следующее. Если функция возвращает объект класса, она автоматически создает временный объект, который хранит возвращаемое значение. Именно этот объект реально и возвращается функцией. После возврата значения объект разрушается. Разрушение временного объекта в некоторых ситуациях может вызвать непредвиденные побочные эффекты. Например, если объект, возвращаемый функцией, имеет деструктор, который освобождает динамически выделяемую память, эта память будет освобождена даже в том случае, если объект, получающий возвращаемое функцией значение, все еще ее использует. Рассмотрим следующую некорректную версию предыдущей программы.

```
// Ошибка, генерируемая при возвращении объекта функцией.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class sample {
   char *s;
  public:
    sample() { s = 0; }
    ~sample() {
      if(s) delete [] s;
```

```
    if(s) delete [] s;
    cout << "Освобождение s-памяти.\n";
  }
  void show() { cout << s << "\n"; }
  void set(char *str);
};

// Загрузка строки.
void sample::set(char *str)
{
  s = new char[strlen(str)+1];
  strcpy(s, str);
}

// Эта функция возвращает объект типа sample.
sample input()
{
  char instr[80];
  sample str;

  cout << "Введите строку: ";
  cin >> instr;

  str.set(instr);
  return str;
}

int main()
{
  sample ob;

  // Присваиваем объект, возвращаемый
  // функцией input(), объекту ob.
  ob = input(); // Эта инструкция генерирует ошибку!!!!
  ob.show(); // Отображение "мусора".

  return 0;
}
```

Результаты выполнения этой программы выглядят таким образом.

```
Введите строку: Привет
Освобождение s-памяти.
Освобождение s-памяти.
Здесь мусор
Освобождение s-памяти.
```

Обратите внимание на то, что деструктор класса *sample* вызывается три раза! В первый раз он вызывается при выходе локального объекта *str* из области видимости в момент возвращения из функции *input()*. Второй вызов деструктора *~sample()* происходит тогда,

когда разрушается временный объект, возвращаемый функцией *input()*. Когда функция возвращает объект, автоматически генерируется невидимый (для вас) временный объект, который хранит возвращаемое значение. В данном случае этот объект просто представляет собой побитовую копию объекта *str*, который является значением, возвращаемым из функции. Следовательно, после возвращения из функции выполняется деструктор временного объекта. Поскольку область памяти, выделенная для хранения строки, вводимой пользователем, уже была освобождена (причем дважды!), при вызове функции *show()* на экран выведется *"мусор"*. (Вы можете не увидеть вывод на экран *"мусора"*. Это зависит от того, как ваш компилятор реализует динамическое выделение памяти. Однако ошибка все равно здесь присутствует.) Наконец, по завершении программы вызывается деструктор объекта *ob* (в функции *main()*). Ситуация здесь осложняется тем, что при первом вызове деструктора освобождается память, выделенная для хранения строки, получаемой функцией *input()*. Таким образом, само по себе плохо не только то, что остальные два обращения к деструктору класса *sample* попытаются освободить уже освобожденную область динамически выделяемой памяти, но они также могут разрушить систему динамического распределения памяти.

Здесь важно понимать, что при возврате объекта из функции для временного объекта, хранящего возвращаемое значение, будет вызван его деструктор. Поэтому следует избегать возврата объектов в ситуациях, когда это может иметь пагубные последствия. Для решения этой проблемы вместо возврата объекта из функции используется возврат указателя или ссылки на объект. Но это не всегда осуществимо. Еще один способ решения этой проблемы включает использование конструктора копии, которому посвящен следующий раздел.

Создание и использование конструктора копии

Одним из самых важных форматов перегруженного конструктора является конструктор копии. Как было показано в предыдущих примерах, при передаче объекта функции или возврате объекта из функции могут возникать проблемы. В этом разделе вы узнаете, что один из способов избежать этих проблем состоит в определении *конструктора копии*, который представляет собой специальный тип перегруженного конструктора.

Для начала еще раз сформулируем проблемы, для решения которых мы хотим определить конструктор копии. При передаче объекта функции создается побитовая (т.е. точная) копия этого объекта, которая передается параметру этой функции. Однако возможны ситуации, когда такая идентичная копия нежелательна. Например, если оригинальный объект содержит указатель на выделяемую динамически память, то и указатель, принадлежащий копии, также будет ссылаться на ту же область памяти. Следовательно, если копия внесет изменения в содержимое этой области памяти, эти изменения коснутся также оригинального объекта! Более того, при завершении функции копия будет разрушена (с вызовом деструктора). Это может нежелательным образом сказаться на исходном объекте.

Аналогичная ситуация возникает при возврате объекта из функции. Компилятор генерирует временный объект, который будет хранить копию значения, возвращаемого функцией. (Это делается автоматически, и без нашего на то согласия.) Этот временный объект выходит за пределы области видимости сразу же, как только инициатору вызова этой функции будет возвращено "обещанное" значение, после чего незамедлительно вызывается деструктор временного объекта. Но если этот деструктор разрушит что-либо нужное для выполняемого далее кода, последствия будут печальны.

Конструктор копии позволяет управлять действиями, составляющими процесс создания копии объекта.

В сердцевине рассматриваемых проблем лежит создание побитовой копии объекта. Чтобы предотвратить их возникновение, необходимо точно определить, что должно происходить, когда создается копия объекта, и тем самым избежать нежелательных побочных эффектов. Этого можно добиться путем создания конструктора копии.

Прежде чем подробнее знакомиться с использованием конструктора копии, важно понимать, что в C++ определено два отдельных вида ситуаций, в которых значение одного объекта передается другому. Первой такой ситуацией является присваивание, а второй — инициализация. Инициализация может выполняться тремя способами, т.е. в случаях, когда:

■ один объект явно инициализирует другой объект, как, например, в объявлении;

■ копия объекта передается параметру функции;

■ генерируется временный объект (чаще всего в качестве значения, возвращаемого функцией).

Конструктор копии применяется только к инициализациям. Он не применяется к присваиваниям.

Узелок на память. *Конструкторы копии не оказывают никакого влияния на операции*

присваивания.

Конструктор копии вызывается в случае, когда один объект инициализирует другой.

Вот как выглядит самый распространенный формат конструктора копии.

```
имя_класса (const имя_класса &obj) {
  // тело конструктора
}
```

Здесь элемент *obj* означает ссылку на объект, которая используется для инициализации другого объекта. Например, предположим, у нас есть класс *myclass* и объект *y* типа *myclass*, тогда при выполнении следующих инструкций будет вызван конструктор копии класса *myclass*.

```
myclass x = y; // Объект y явно инициализирует объект x
x.func1(y); // Объект y передается в качестве аргумента.
y = func2(); // Объект y принимает объект, возвращаемый функцией.
```

В первых двух случаях конструктору копии будет передана ссылка на объект *y*, а в третьем — ссылка на объект, возвращаемый функцией *func2()*.

Чтобы глубже понять назначение конструкторов копии, рассмотрим подробнее их роль в каждой из этих трех ситуаций.

Конструкторы копии и параметры функции

При передаче объекта функции в качестве аргумента создается копия этого объекта. Если в классе определен конструктор копии, то именно он и вызывается для создания копии. Рассмотрим программу, в которой используется конструктор копии для надлежащей обработки объектов типа *myclass* при их передаче функции в качестве аргументов. (Ниже приводится корректная версия некорректной программы, представленной выше в этой главе.)

```
// Использование конструктора копии для
// определения параметра.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int *p;
  public:
   myclass(int i); // обычный конструктор
   myclass(const myclass &ob); // конструктор копии
   ~myclass();
   int getval() { return *p; }
};

// Конструктор копии.
myclass::myclass(const myclass &obj)
{
  p = new int;
  *p = *obj.p; // значение копии
  cout << "Вызван конструктор копии.\n";
```

```
}

// Обычный конструктор.
myclass::myclass(int i)
{
  cout << "Выделение памяти, адресуемой указателем p.\n";
  p = new int;
  *p = i;
}

myclass::~myclass()
{
  cout <<"Освобождение памяти, адресуемой указателем p.\n";
  delete p;
}

// Эта функция принимает один объект-параметр.
void display(myclass ob)
{
  cout << ob.getval() << '\n';
}

int main()
{
  myclass a(10);
  display(a);
  return 0;
}
```

Эта программа генерирует такие результаты.

```
Выделение памяти, адресуемой указателем p.
Вызван конструктор копии.
10
Освобождение памяти, адресуемой указателем p.
Освобождение памяти, адресуемой указателем p.
```

При выполнении этой программы здесь происходит следующее: когда в функции *main()* создается объект *a*, *"стараниями"* обычного конструктора выделяется память, и адрес этой области памяти присваивается указателю *a.p*. Затем объект *a* передается функции *display()*, а именно— ее параметру *ob*. В этом случае вызывается конструктор копии, который создает копию объекта *a*. Конструктор копии выделяет память для этой копии, а значение указателя на выделенную область памяти присваивает члену *p* объекта-копии. Затем значение, адресуемое указателем *p* исходного объекта, записывается в область памяти, адрес которой хранится в указателе *p* объекта-копии. Таким образом, области памяти, адресуемые указателями *a.p* и *ob.p*, раздельны и независимы одна от другой, но хранимые в них значения (на которые указывают *a.p* и *ob.p*) одинаковы. Если бы конструктор копии не был определен, то в результате создания по умолчанию побитовой копии члены *a.p* и *ob.p* указывали бы на одну и ту же область памяти.

По завершении функции *display()* объект *ob* выходит из области видимости. Этот выход сопровождается вызовом его деструктора, который освобождает область памяти, адресуемую указателем *ob.p*. Наконец, по завершении функции *main()* выходит из области видимости объект *a*, что также сопровождается вызовом его деструктора и соответствующим освобождением области памяти, адресуемой указателем *a.p*. Как видите, использование конструктора копии устраняет деструктивные побочные эффекты, связанные с передачей объекта функции.

Использование конструкторов копии при инициализации объектов

Конструктор копии также вызывается в случае, когда один объект используется для инициализации другого.

Рассмотрим следующую простую программу.

```
// Вызов конструктора копии для инициализации объекта.
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
   int *p;
  public:
   myclass(int i); // обычный конструктор
   myclass(const myclass &ob); // конструктор копии
   ~myclass();
   int getval() { return *p; }
};

// Конструктор копии.
myclass::myclass(const myclass &ob)
{
  p = new int;
  *p = *ob.p; // значение копии
  cout << "Выделение р-памяти конструктором копии.\n";
}

// Обычный конструктор.
myclass::myclass(int i)
{
  cout << "Выделение р-памяти обычным конструктором.\n";
  p = new int;
  *p = i;
}

myclass::~myclass()
{
  cout << "Освобождение р-памяти.\n";
  delete p;
}
```

```
}

int main()
{
  myclass a(10); // Вызывается обычный конструктор.
  myclass b = a; // Вызывается конструктор копии.
  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Выделение p-памяти обычным конструктором.
Выделение p-памяти конструктором копии.
Освобождение p-памяти.
Освобождение p-памяти.
```

Как подтверждают результаты выполнения этой программы, при создании объекта *a* вызывается обычный конструктор. Но когда объект *a* используется для инициализации объекта *b*, вызывается конструктор копии. Использование конструктора копии гарантирует, что объект *b* выделит для своих членов данных собственную область памяти. Без конструктора копии объект *b* попросту представлял бы собой точную копию объекта *a*, а член *a.p* указывал бы на ту же самую область памяти, что и член *b.p*.

Следует иметь в виду, что конструктор копии вызывается только в случае выполнения инициализации. Например, следующая последовательность инструкций не вызовет конструктор копии, определенный в предыдущей программе:

```
myclass a(2), b(3);
// ...
b = a;
```

Здесь инструкция *b = a* выполняет операцию присваивания, а не операцию копирования.

Использование конструктора копии при возвращении функцией объекта

Конструктор копии также вызывается при создании временного объекта, который является результатом возвращения функцией объекта. Рассмотрим следующую короткую программу.

```
/* Конструктор копии вызывается в результате создания временного объекта
в качестве значения, возвращаемого функцией.
*/
#include <iostream>
using namespace std;

class myclass {
  public:
    myclass() { cout << "Обычный конструктор.\n"; }
    myclass(const myclass &obj) {cout << "Конструктор копии.\n"; }
};

myclass f()
{
  myclass ob; // Вызывается обычный конструктор.
  return ob; // Неявно вызывается конструктор копии.
}
```

```
}
int main()
{
  myclass a; // Вызывается обычный конструктор.
  a = f(); // Вызывается конструктор копии.
  return 0;
}
```

Эта программа генерирует такие результаты.

```
Обычный конструктор.
Обычный конструктор.
Конструктор копии.
```

Здесь обычный конструктор вызывается дважды: первый раз при создании объекта *a* в функции *main()*, второй — при создании объекта *ob* в функции *f()*. Конструктор копии вызывается в момент, когда генерируется временный объект в качестве значения, возвращаемого из функции *f()*.

Хотя "скрытые" вызовы конструкторов копии могут отдавать мистикой, нельзя отрицать тот факт, что практически каждый класс в профессионально написанных программах содержит явно определенный конструктор копии, без которого не избежать побочных эффектов, возникающих в результате создания по умолчанию побитовых копий объекта.

Конструкторы копии — а нельзя ли найти что-то попроще?

Как уже неоднократно упоминалось в этой книге, C++ — очень мощный язык. Он имеет множество средств, которые наделяют его широкими возможностями, но при этом его можно назвать сложным языком. Конструкторы копии представляют собой механизм, на который ссылаются многие программисты как на основной пример сложности языка, поскольку это средство не воспринимается на интуитивном уровне. Начинающие программисты часто не понимают, почему так важен конструктор копии. Для многих не сразу становится очевидным ответ на вопрос: когда нужен конструктор копии, а когда — нет. Эта ситуация часто выражается в такой форме: *"А не существует ли более простого способа?"*. Ответ также непрост: *и да, и нет!*

Такие языки, как Java и C#, не имеют конструкторов копии, поскольку ни в одном из них не создаются побитовые копии объектов. Дело в том, что как Java, так и C# динамически выделяют память для всех объектов, а программист оперирует этими объектами исключительно через ссылки. Поэтому при передаче объектов в качестве параметров функции или при возврате их из функций в копиях объектов нет никакой необходимости.

Тот факт, что ни Java, ни C# не нуждаются в конструкторах копии, делает эти языки проще, но за простоту тоже нужно платить. Работа с объектами исключительно посредством ссылок (а не напрямую, как в C++) налагает ограничения на тип операций, которые может выполнять программист. Более того, такое использование объектных ссылок в Java и C# не позволяет точно определить, когда объект будет разрушен. В C++ же объект всегда разрушается при выходе из области видимости.

Язык C++ предоставляет программисту полный контроль над ситуациями, складывающимися в программе, поэтому он несколько сложнее, чем Java и C#. Это — цена, которую мы платим за мощность программирования.

Ключевое слово this

Ключевое слово this — *это указатель на объект, который вызывает функцию-член*.

При каждом вызове функции-члена ей автоматически передается указатель, именуемый ключевым словом *this*, на объект, для которого вызывается эта функция. Указатель *this* — это *неявный* параметр, принимаемый всеми функциями-членами. Следовательно, в любой

функции-члене указатель this можно использовать для ссылки на вызывающий объект.

Как вы знаете, функция-член может иметь прямой доступ к закрытым (private) членам данных своего класса.

Например, у нас определен такой класс.

```
class cl {
  int i;
  void f( ) { ... };
  // . . .
};
```

В функции *f()* можно использовать следующую инструкцию для присваивания члену *i* значения *10*.

```
i = 10;
```

В действительности предыдущая инструкция представляет собой сокращенную форму следующей.

```
this->i = 10;
```

Чтобы понять, как работает указатель *this*, рассмотрим следующую короткую программу.

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

class cl {
    int i;
  public:
    void load_i(int val) { this->i = val; } // то же самое, что i = val
    int get_i( ) { return this->i; } // то же самое, что return i
};

int main( )
{
  cl o;
  o.load_i (100);
  cout << o.get_i( );
  return 0;
}
```

При выполнений эта программа отображает число *100*.

Безусловно, предыдущий пример тривиален, но в нем показано, как можно использовать указатель *this*. Скоро вы поймете, почему указатель *this* так важен для программирования на C++.

Важно! *Функции-"друзья" не имеют указателя this, поскольку они не являются членами класса. Только функции-члены имеют указатель this.*

Глава 13: Перегрузка операторов

В C++ операторы можно перегружать для "классовых" типов, определяемых программистом. Принципиальный выигрыш от перегрузки операторов состоит в том, что она позволяет органично интегрировать новые типы данных в среду программирования.

Перегружая оператор, можно определить его значение для конкретного класса. Например

Перегружая оператор, можно определить его значение для конкретного класса. Например, класс, который определяет связный список, может использовать оператор *"+"* для добавления объекта к списку. Класс, которые реализует стек, может использовать оператор *"+"* для записи объекта в стек. В каком-нибудь другом классе тот же оператор *"+"* мог бы служить для совершенно иной цели. При перегрузке оператора ни одно из оригинальных его значений не теряется. Перегруженный оператор (в своем новом качестве) работает как совершенно новый оператор. Поэтому перегрузка оператора *"+"* для обработки, например, связного списка не приведет к изменению его функции (т.е. операции сложения) по отношению к целочисленным значениям.

Перегрузка операторов тесно связана с перегрузкой функций. Чтобы перегрузить оператор, необходимо определить значение новой операции для класса, к которому она будет применяться. Для этого создается функция *operator* (операторная функция), которая определяет действие этого оператора. Общий формат функции *operator* таков.

```
тип имя_класса::operator#(список_аргументов)
{
  операция_над_классом
}
```

Операторы перегружаются с помощью функции operator.

Здесь перегружаемый оператор обозначается символом *"#"*, а элемент *тип* представляет собой тип значения, возвращаемого заданной операцией. И хотя он в принципе может быть любым, тип значения, возвращаемого функцией *operator*, часто совпадает с именем класса, для которого перегружается данный оператор. Такая корреляция облегчает использование перегруженного оператора в составных выражениях. Как будет показано ниже, конкретное значение элемента *список_аргументов* определяется несколькими факторами.

Операторная функция может быть членом класса или не быть им. Операторные функции, не являющиеся членами класса, часто определяются как его "друзья". Операторные функции-члены и функции-не члены класса различаются по форме перегрузке. Каждый из вариантов мы рассмотрим в отдельности.

Перегрузка операторов с использованием функций-членов

Начнем с простого примера. В следующей программе создается класс *three_d*, который поддерживает координаты объекта в трехмерном пространстве. Для класса *three_d* перегружаются операторы *"+"* и *"="*. Итак, рассмотрим внимательно код этой программы.

```
// Перегрузка операторов с помощью функций-членов.
#include <iostream>
using namespace std;

class three_d {
   int x, y, z; // 3-мерные координаты
  public:
   three_d() { x = y = z = 0; }
   three_d(int i, int j, int k) {x = i; y = j; z = k; }

   three_d operator+(three_d op2); // Операнд op1 передается неявно.
   three_d operator=(three_d op2); // Операнд op1 передается неявно.

   void show();
};

// Перегрузка оператора "+".
three_d three_d::operator+(three_d op2)
```

```
three_d three_d::operator+(three_d op2)
{
  three_d temp;
  temp.x = x + op2.x; // Операции сложения целочисленных
  temp.y = y + op2.y; // значений сохраняют оригинальный
  temp.z = z + op2.z; // смысл.

  return temp;
}

// Перегрузка оператора присваивания.
three_d three_d::operator=(three_d op2)
{
  x = op2.x; // Операции присваивания целочисленных
  y = op2.y; // значений сохраняют оригинальный
  z = op2.z; // смысл.

  return *this;
}

// Отображение координат X, Y, Z.
void three_d::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  three_d a(1, 2, 3), b(10, 10, 10), c;
  a.show();
  b.show();

  c=a+b; // сложение объектов a и b
  c.show();

  c=a+b+c; // сложение объектов a, b и c
  c.show();

  c=b=a; // демонстрация множественного присваивания
  c.show();
  b.show();

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
1, 2, 3
```

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

Исследуя код этой программы, вы, вероятно, удавились, увидев, что обе операторные функции имеют только по одному параметру, несмотря на то, что они перегружают бинарные операции. Это, на первый взгляд, "вопиющее" противоречие можно легко объяснить. Дело в том, что при перегрузке бинарного оператора с использованием функции-члена ей передается явным образом только один аргумент. Второй же неявно передается через указатель *this*. Таким образом, в строке

```
temp.x = x + op2.x;
```

под членом *x* подразумевается член *this->x*, т.е. член *x* связывается с объектом, который вызывает данную операторную функцию. Во всех случаях неявно передается объект, указываемый слева от символа операции, который стал причиной вызова операторной функции. Объект, располагаемый с правой стороны от символа операции, передается этой функции в качестве аргумента. В общем случае при использовании функции-члена для перегрузки унарного оператора параметры не используются вообще, а для перегрузки бинарного — только один параметр. (Тернарный оператор *"?"* перегружать нельзя.) В любом случае объект, который вызывает операторную функцию, неявно передается через указатель *this*.

Чтобы понять, как работает механизм перегрузки операторов, рассмотрим внимательно предыдущую программу, начиная с перегруженного оператора *"+"*. При обработке двух объектов типа *three_d* оператором *"+"* выполняется сложение значений соответствующих координат, как показано в функции *operator+()*. Но заметьте, что эта функция не модифицирует значение ни одного операнда. В качестве результата операции эта функция возвращает объект типа *three_d*, который содержит результаты попарного сложения координат двух объектов. Чтобы понять, почему операция *"+"* не изменяет содержимое ни одного из объектов-участников, рассмотрим стандартную арифметическую операцию сложения, примененную, например, к числам *10* и *12*. Результат операции *10+12* равен *22*, но при его получении ни *10*, ни *12* не были изменены. Хотя не существует правила, которое бы не позволяло перегруженному оператору изменять значение одного из его операндов, все же лучше, чтобы он не противоречил общепринятым нормам и оставался в согласии со своим оригинальным назначением.

Обратите внимание на то, что функция *operator+()* возвращает объект типа *three_d*. Несмотря на то что она могла бы возвращать значение любого допустимого в C++ типа, тот факт, что она возвращает объект типа *three_d*, позволяет использовать оператор *"+"* в таких составных выражениях, как *a+b+c*. Часть этого выражения, *a+b*, генерирует результат типа *three_d*, который затем суммируется с объектом *c*. И если бы эта часть выражения генерировала значение иного типа (а не типа *three_d*), такое составное выражение попросту не работало бы.

В отличие от оператора *"+"*, оператор присваивания приводит к модификации одного из своих аргументов. (Прежде всего, это составляет саму суть присваивания.) Поскольку функция *operator=()* вызывается объектом, который расположен слева от символа присваивания (=), именно этот объект и модифицируется в результате операции присваивания. После выполнения этой операции значение, возвращаемое перегруженным оператором, содержит объект, указанный слева от символа присваивания. (Такое положение вещей вполне согласуется с традиционным действием оператора *"="*.) Например, чтобы можно было выполнять инструкции, подобные следующей

```
a = b = c = d;
```

неооходимо, чтооы операторная функция *operator=()* возвращала ооъект, адресуемыи указателем *this*, и чтобы этот объект располагался слева от оператора *"="*. Это позволит выполнить любую цепочку присваиваний. Операция присваивания — это одно из самых важных применений указателя *this*.

Узелок на память. *Если для перегрузки бинарного оператора используется функция-член, объект, стоящий слева от оператора, вызывает операторную функцию и передается ей неявно через указатель this. Объект, расположенный справа от оператора, передается операторной функции как параметр.*

Использование функций-членов для перегрузки унарных операторов

Можно также перегружать такие унарные операторы, как *"++"*, *"--"*, или унарные *"-"* и *"+"*. Как упоминалось выше, при перегрузке унарного оператора с помощью функции-члена операторной функции ни один объект не передается явным образом. Операция же выполняется над объектом, который генерирует вызов этой функции через неявно переданный указатель *this*. Например, рассмотрим расширенную версию предыдущего примера программы. В этом варианте для объектов типа *three_d* определяется операция инкремента.

```
// Перегрузка унарного оператора.
#include <iostream>
using namespace std;

class three_d {
    int x, y, z; // 3-мерные координаты
  public:
    three_d() { x = y = z = 0; }
    three_d(int i, int j, int k) {x = i; y = j; z = k; }

    three_d operator+(three_d op2); // Операнд op1 передается неявно.
    three_d operator=(three_d op2); // Операнд op1 передается неявно.
    three_d operator++(); // префиксная версия оператора ++

    void show();
};

// Перегрузка оператора " + ".
three_d three_d::operator+(three_d op2)
{
  three_d temp;
  temp.x = x + op2.x; // Операции сложения целочисленных
  temp.y = y + op2.y; // значений сохраняют оригинальный
  temp.z = z + op2.z; // смысл.
  return temp;
}

// Перегрузка оператора присваивания.
three_d three_d::operator=(three_d op2)
{
  x = op2.x; // Операции присваивания целочисленных
  y = op2.y; // значений сохраняют оригинальный
```

```
  z = op2.z; // смысл.
  return *this;
}

// Перегруженная префиксная версия оператора "++".
three_d three_d::operator++()
{
  x++; // инкремент координат x, y и z
  y++;
  z++;
  return *this;
}

// Отображение координат X, Y, Z.
void three_d::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  three_d a(1, 2, 3), b(10, 10, 10), c;
  a.show();
  b.show();

  c = a + b; // сложение объектов a и b
  c.show();

  c=a+b+c; // сложение объектов a, b и c
  c.show();

  c = b = a; // множественное присваивание
  c.show();
  b.show();

  ++c; // инкремент c
  c.show();

  return 0;
}
```

Эта версия программы генерирует такие результаты.

```
1, 2, 3
10, 10, 10
11, 12, 13
```

```
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
2, 3, 4
```

Как видно по последней строке результатов программы, операторная функция *operator++()* инкрементирует каждую координату объекта и возвращает модифицированный объект, что вполне согласуется с традиционным действием оператора *"++"*.

Как вы знаете, операторы *"++"* и *"--"* имеют префиксную и постфиксную формы. Например, оператор инкремента можно использовать в форме

```
++0;
```

и в форме

```
0++;.
```

Как отмечено в комментариях к предыдущей программе, функция *operator++()* определяет префиксную форму оператора *"++"* для класса *three_d*. Но нам ничего не мешает перегрузить и постфиксную форму. Прототип постфиксной формы оператора *"++"* для класса *three_d* имеет следующий вид.

```
three_d three_d::operator++(int notused);
```

Операторы инкремента и декремента имеют как префиксную, так и постфиксную формы.

Параметр *notused* не используется самой функцией. Он служит индикатором для компилятора, позволяющим отличить префиксную форму оператора инкремента от постфиксной. (Этот параметр также используется в качестве признака постфиксной формы и для оператора декремента.) Ниже приводится один из возможных способов реализации постфиксной версии оператора *"++"* для класса *three_d*.

```
// Перегрузка постфиксной версии оператора "++".
three_d three_d::operator++(int notused)
{
  three_d temp = *this; // сохранение исходного значения

  x++; // инкремент координат x, y и z
  y++;
  z++;
  return temp; // возврат исходного значения
}
```

Обратите внимание на то, что эта функция сохраняет текущее значение операнда путем выполнения такой инструкции.

```
three_d temp = *this;
```

Сохраненное значение операнда (в объекте *temp*) возвращается с помощью инструкции *return*. Следует иметь в виду, что традиционный постфиксный оператор инкремента сначала получает значение операнда, а затем его инкрементирует. Следовательно, прежде чем инкрементировать текущее значение операнда, его нужно сохранить, а затем и возвратить (не забывайте, что постфиксный оператор инкремента не должен возвращать модифицированное значение своего операнда).

В следующей версии исходной программы реализованы обе формы оператора *"++"*.

```
// Демонстрация перегрузки оператора "++" с
// использованием его префиксной и постфиксной форм.
#include <iostream>
```

```
using namespace std;

class three_d {
    int x, y, z; // 3-мерные координаты
  public:
    three_d() { x = y = z = 0; }
    three_d(int i, int j, int k) {x = i; y = j; z = k; }

    three_d operator+(three_d op2); // Операнд op1 передается неявно.
    three_d operator=(three_d op2); // Операнд op1 передается неявно.
    three_d operator++(); // префиксная версия
    three_d operator++(int notused); // постфиксная версия

    void show();
};

// Перегрузка оператора " + ".
three_d three_d::operator+(three_d op2)
{
  three_d temp;
  temp.x = x + op2.x; // Операции сложения целочисленных
  temp.y = y + op2.y; // значений сохраняют оригинальный
  temp.z = z + op2.z; // смысл.
  return temp;
}

// Перегрузка оператора присваивания.
three_d three_d::operator=(three_d op2)
{
  x = op2.x; // Операции присваивания целочисленных
  y = op2.y; // значений сохраняют оригинальный
  z = op2.z; // смысл.
  return *this;
}

// Перегрузка префиксной версии оператора "++".
three_d three_d::operator++()
{
  x++; // инкремент координат x, y и z
  y++;
  z++;
  return *this;
}

// Перегрузка постфиксной версии оператора "++".
three_d three_d::operator++(int notused)
```

```
three_d three_d::operator++ (int notused)
{
  three_d temp = *this; // сохранение исходного значения
  x++; // инкремент координат x, y и z
  y++;
  z++;
  return temp; // возврат исходного значения
}

// Отображение координат X, Y, Z.
void three_d::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  three_d a(1, 2, 3), b(10, 10, 10), c;
  a.show();
  b.show();

  c = a + b; // сложение объектов a и b
  c.show();

  c=a+b+c; // сложение объектов a, b и c
  c.show();

  c = b = a; // множественное присваивание
  c.show();
  b.show();

  ++c; // префиксная форма инкремента
  c.show();
  c++; // постфиксная форма инкремента
  c.show();
   a = ++c; // Объект a получает значение объекта c после его инкрементирования.
  a.show(); // Теперь объекты a и c
  c.show(); // имеют одинаковые значения.
   a = c++; // Объект a получает значение объекта c до его инкрементирования.
  a.show(); // Теперь объекты a и c
  c.show(); // имеют различные значения.
  return 0;
```

```
}
```

Вот как выглядят результаты выполнения этой версии программы.

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
2, 3, 4
3, 4, 5
4, 5, 6
4, 5, 6
4, 5, 6
5, 6, 7
```

Как подтверждают последние четыре строки результатов программы, при префиксном инкрементировании значение объекта *c* увеличивается до выполнения присваивания объекту *a*, при постфиксном инкрементировании — после присваивания.

Помните, что если символ *"++"* стоит перед операндом, вызывается операторная функция *operator++()*, а если после операнда — операторная функция *operator++(int notused)*.Тот же подход используется и для перегрузки префиксной и постфиксной форм оператора декремента для любого класса. В качестве упражнения определите оператор декремента для класса *three_d*.

Важно! *Ранние версии языка C++ не содержали различий между префиксной и постфиксной формами операторов инкремента и декремента. Тогда в обоих случаях вызывалась префиксная форма операторной функции. Это следует иметь в виду, если вам придется работать со старыми C++-программами.*

Советы по реализации перегрузки операторов

Действие перегруженного оператора применительно к классу, для которого он определяется, не обязательно должно иметь отношение к стандартному действию этого оператора применительно к встроенным C++-типам. Например, операторы *"<<"* и *">>"*, применяемые к объектам *cout* и *cin*, имеют мало общего с аналогичными операторами, применяемыми к значениям целочисленного типа. Но для улучшения структурированности и читабельности программного кода создаваемый перегруженный оператор должен по возможности отражать исходное назначение того или иного оператора. Например, оператор *"+"*, перегруженный для класса *three_d*, концептуально подобен оператору *"+"*, определенному для целочисленных типов. Ведь вряд ли есть логика в определении для класса, например, оператора *"+"*, который по своему действию больше напоминает оператор деления (*/*). Таким образом, основная идея создания перегруженного оператора — наделить его новыми (нужными для вас) возможностями, которые, тем не менее, связаны с его первоначальным назначением.

На перегрузку операторов налагается ряд ограничений. Во-первых, нельзя изменять приоритет оператора. Во-вторых, нельзя изменять количество операндов, принимаемых оператором, хотя операторная функция могла бы игнорировать любой операнд. Наконец, за исключением оператора вызова функции (о нем речь впереди), операторные функции не могут иметь аргументов по умолчанию. Некоторые операторы вообще нельзя перегружать. Ниже перечислены операторы, перегрузка которых запрещена.

```
.     ::     .*     ?
```

Оператор ".*" — это оператор специального назначения (он рассматривается ниже в этой

Оператор "." — это оператор специального назначения (он рассматривается ниже в этой книге).

О значении порядка операндов

Перегружая бинарные операторы, помните, что во многих случаях порядок следования операндов имеет значение. Например, выражение *A+B* коммутативно, а выражение *A-B* — нет. (Другими словами, *A - B* не то же самое, что *B - A*!) Следовательно, реализуя перегруженные версии некоммутативных операторов, необходимо помнить, какой операнд стоит слева от символа операции, а какой — справа от него. Например, в следующем фрагменте кода демонстрируется перегрузка оператора вычитания для класса *three_d*.

```
// Перегрузка оператора вычитания.
three_d three_d::operator-(three_d op2)
{
  three_d temp;
  temp.x = x - op2.x;
  temp.y = y - op2.y;
  temp.z = z - op2.z;
  return temp;
}
```

Помните, что именно левый операнд вызывает операторную функцию. Правый операнд передается в явном виде. Вот почему для корректного выполнения операции вычитания используется именно такой порядок следования операндов:

`x - op2.x.`

Перегрузка операторов с использованием функций-не членов класса

Бинарные операторные функции, которые не являются членами класса, имеют два параметра, а унарные (тоже не члены) — один.

Перегрузку оператора для класса можно реализовать и с использованием функции, не являющейся членом этого класса. Такие функции часто определяются *"друзьями"* класса. Как упоминалось выше, функции-не члены (в том числе и функции-*"друзья"*) не имеют указателя *this*. Следовательно, если для перегрузки бинарного оператора используется функция-*"друг"*, явным образом передаются оба операнда. Если же с помощью функции-*"друга"* перегружается унарный оператор, операторной функции передается один оператор. С использованием функций-не членов класса нельзя перегружать такие операторы:

`=, (), [] и ->.`

Например, в следующей программе для перегрузки оператора *"+"* вместо функции-члена используется функция-*"друг"*.

```
// Перегрузка оператора "+" с помощью функции-"друга".
#include <iostream>
using namespace std;

class three_d {
   int x, y, z; // 3-мерные координаты
  public:
   three_d() { x = y = z = 0; }
   three_d(int i, int j, int k) { x = i; y = j; z = k; }

   friend three_d operator+(three_d op1, three_d op2);
```

```
    three_d operator= (three_d op2); // Операнд op1 передается неявно.
    void show();
};

// Теперь это функция-"друг".
three_d operator+(three_d op1, three_d op2)
{
  three_d temp;
  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
  temp.z = op1.z + op2.z;
  return temp;
}

// Перегрузка присваивания.
three_d three_d::operator=(three_d op2)
{
  x = op2.x;
  y = op2.y;
  z = op2.z;
  return *this;
}

// Отображение координат X, Y, Z.
void three_d::show()
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  three_d a(1, 2, 3), b(10, 10, 10), c;
  a.show();
  b.show();

  c = a + b; // сложение объектов a и b
  c.show();

  c=a+b+c; // сложение объектов a, b и c
  c.show();

  c = b = a; // демонстрация множественного присваивания
  c.show();
  b.show();
```

```
  return 0;
}
```

Как видите, операторной функции *operator+()* теперь передаются два операнда. Левый операнд передается параметру *op1*, а правый — параметру *op2*.

Во многих случаях при перегрузке операторов с помощью функций-*"друзей"* нет никакого преимущества по сравнению с использованием функций-членов класса. Однако возможна ситуация (когда нужно, чтобы слева от бинарного оператора стоял объект встроенного типа), в которой функция-*"друг"* оказывается чрезвычайно полезной. Чтобы понять это, рассмотрим следующее. Как вы знаете, указатель на объект, который вызывает операторную функцию-член, передается с помощью ключевого слова *this*. При использовании бинарного оператора функцию вызывает объект, расположенный слева от него. И это замечательно при условии, что левый объект определяет заданную операцию. Например, предположим, что у нас есть некоторый объект *ob*, для которого определена операция сложения с целочисленным значением, тогда следующая запись представляет собой вполне допустимое выражение.

```
ob + 10; // будет работать
```

Поскольку объект *ob* стоит слева от оператора *"+"*, он вызывает перегруженную операторную функцию, которая (предположительно) способна выполнить операцию сложения целочисленного значения с некоторым элементом объекта *ob*. Но эта инструкция работать не будет.

```
10 + ob; // не будет работать
```

Дело в том, что в этой инструкции объект, расположенный слева от оператора *"+"*, представляет собой целое число, т.е. значение встроенного типа, для которого не определена ни одна операция, включающая целое число и объект классового типа.

Решение описанной проблемы состоит в перегрузке оператора *"+"* с использованием двух функций-*"друзей"* В этом случае операторной функции явным образом передаются оба аргумента, и она вызывается подобно любой другой перегруженной функции, т.е. на основе типов ее аргументов. Одна версия операторной функции *operator+()* будет обрабатывать аргументы *объект + int-значение*, а другая — аргументы *int-значение + объект*. Перегрузка оператора *"+"* (или любого другого бинарного оператора) с использованием функций-*"друзей"* позволяет ставить значение встроенного типа как справа, так и слева от оператора. Реализация этого решения показана в следующей программе.

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

class CL {
  public:
    int count;
    CL operator=(CL obj);
    friend CL operator+(CL ob, int i);
    friend CL operator+(int i, CL ob);
};

CL CL::operator=(CL obj)
{
  count = obj.count;
  return *this;
```

```
}

// Эта версия обрабатывает аргументы
// объект + int-значение.
CL operator+(CL ob, int i)
{
  CL temp;
  temp.count = ob.count + i;
  return temp;
}

// Эта версия обрабатывает аргументы
// int-значение + объект.
CL operator+(int i, CL ob)
{
  CL temp;
  temp.count = ob.count + i;
  return temp;
}

int main( )
{
  CL o;
  o.count = 10;
  cout << o.count << " "; // выводит число 10

  o=10+o; // сложение числа с объектом
  cout << o.count << " "; // выводит число 20

  o=o+12; // сложение объекта с числом
  cout << 0.count; // выводит число 32

  return 0;
}
```

Как видите, операторная функция *operator+()* перегружается дважды, позволяя тем самым предусмотреть два возможных способа участия целого числа и объекта типа *CL* в операции сложения.

Использование функций-"друзей" для перегрузки унарных операторов

С помощью функций-*"друзей"* можно перегружать и унарные операторы. Но это потребует от программиста дополнительных усилий. Для начала мысленно вернемся к исходной версии перегруженного оператора *"++"*, определенной для класса *three_d* и реализованной в виде функции-члена. Для удобства приведем код этой операторной функции здесь.

```
// Перегрузка префиксной формы оператора "++".
three_d three_d::operator++( )
{
  x++;
  y++;
```

```
  z++;
  return *this;
}
```

Как вы знаете, каждая функция-член получает (в качестве неявно переданного) аргумент *this*, который является указателем на объект, вызвавший эту функцию. При перегрузке унарного оператора с помощью функции-члена аргументы явным образом не передаются вообще. Единственным аргументом, необходимым в этой ситуации, является неявный указатель на вызывающий объект. Любые изменения, вносимые в данные объекта, повлияют на объект, для которого была вызвана эта операторная функция. Следовательно, при выполнении инструкции *x++* (в предыдущей функции) будет инкрементирован член *x* вызывающего объекта.

В отличие от *функций-членов*, *функции-не члены* (в том числе и "друзья" класса) не получают указатель *this* и, следовательно, не имеют доступа к объекту, для которого они были вызваны. Но мы знаем, что "дружественной" операторной функции операнд передается явным образом. Поэтому попытка создать операторную функцию-*"друга" operator++()* в таком виде успехом не увенчается.

```
// ЭТОТ ВАРИАНТ РАБОТАТЬ НЕ БУДЕТ
three_d operator++(three_d op1)
{
  op1.x++;
  op1.y++;
  op1.z++;
  return op1;
}
```

Эта функция неработоспособна, поскольку только копия объекта, активизировавшего вызов функции *operator++()*, передается функции через параметр *op1*. Таким образом, изменения в теле функции *operator++()* не повлияют на вызывающий объект, они изменяют только локальный параметр.

Если вы хотите для перегрузки операторов инкремента или декремента использовать функцию-*"друга"*, необходимо передать ей объект по ссылке. Поскольку ссылочный параметр представляет собой неявный указатель на аргумент, то изменения, внесенные в параметр, повлияют и на аргумент. Применение ссылочного параметра позволяет функции успешно инкрементировать или декрементировать объект, используемый в качестве операнда.

Если для перегрузки операторов инкремента или декремента используется функция-*"друг"*, ее префиксная форма принимает один параметр (который и является операндом), а постфиксная форма — два параметра (вторым является целочисленное значение, которое не используется).

Ниже приведен полный код программы обработки трехмерных координат, в которой используется операторная функция-*"друг" operator++()*. Обратите внимание на то, что перегруженными являются как префиксная, так и постфиксная формы операторов инкремента.

```
// В этой программе используются перегруженные
// операторные функции-"друзья" operator++() .
#include <iostream>
using namespace std;

class three_d {
```

```
class three_d {
    int x, y, z; // 3-мерные координаты
  public:
    three_d() { x = y = z = 0; }
    three_d(int i, int j, int k) {x = i; y = j; z = k; }

    friend three_d operator+(three_d op1, three_d op2);
    three_d operator=(three_d op2);

    // Эти функции для перегрузки
    // оператора "++" используют ссылочные параметры.
    friend three_d operator++(three_d &op1);
    friend three_d operator++(three_d &op1, int notused);

    void show();
};

// Теперь это функция-"друг".
three_d operator+(three_d op1, three_d op2)
{
  three_d temp;
  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
  temp.z = op1.z + op2.z;
  return temp;
}

// Перегрузка оператора "=".
three_d three_d::operator=(three_d op2)
{
  x = op2.x;
  y = op2.y;
  z = op2.z;
  return *this;
}

/* Перегрузка префиксной версии оператора "++" с использованием функции-
"друга". Для этого необходимо использование ссылочного параметра.
*/
 three_d operator++(three_d &op1)
{
  op1.x++;
  op1.y++;
  op1.z++;
  return op1;
}
```

```
/* Перегрузка постфиксной версии оператора "++" с использованием функции-"друга". Для этого необходимо использование ссылочного параметра.
*/
 three_d operator++(three_d &op1, int notused)
{
  three_d temp = op1;
  op1.x++;
  op1.y++;
  op1.z++;
  return temp;
}

// Отображение координат X, Y, Z.
void three_d:: show( )
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main( )
{
  three_d a(1, 2, 3), b(10, 10, 10), c;
  a.show( );
  b.show( );

  c = a + b; // сложение объектов a и b
  c.show( );

  c=a+b+c; // сложение объектов a, b и c
  c.show( );

  c = b = a; // демонстрация множественного присваивания
  c.show( );
  b.show( );

  ++c; // префиксная версия инкремента
  c.show( );

  c++; // постфиксная версия инкремента
  c. show( );

   a = ++c; // Объект a получает значение объекта c после инкрементирования.
  a.show( ); // В этом случае объекты a и c
  c.show( ); // имеют одинаковые значения координат.
```

```
  a = C++; // Объект a получает значение объекта c до инкрементирования.
  a.show(); // В этом случае объекты a и c
  c.show(); // имеют различные значения координат.
  return 0;
}
```

Узелок на память. *Для реализации перегрузки операторов следует использовать функции-члены. Функции-"друзья" используются в C++ в основном для обработки специальных ситуаций.*

Перегрузка операторов отношения и логических операторов

Операторы отношений (например, *"=="* или *"<"*) и логические операторы (например, *"&&"* или *"||"*) также можно перегружать, причем делать это совсем нетрудно. Как правило, перегруженная операторная функция отношения возвращает объект класса, для которого она перегружается. А перегруженный оператор отношения или логический оператор возвращает одно из двух возможных значений: *true* или *false*. Это соответствует обычному применению этих операторов и позволяет использовать их в условных выражениях.

Рассмотрим пример перегрузки оператора *"=="* для уже знакомого нам класса *three_d*.

```
// Перегрузка оператора "=="
bool three_d::operator==(three_d op2)
{
  if((x == op2.x) && (y == op2.y) && (z == op2.z)) return true;
  else return false;
}
```

Если считать, что операторная функция *operator==()* уже реализована, следующий фрагмент кода совершенно корректен.

```
three_d a, b;
  // ...
if(a == b) cout << "a равно b\n";
else cout << "a не равно b\n";
```

Поскольку операторная функция *operator==()* возвращает результат типа *bool*, ее можно использовать для управления инструкцией *if*. В качестве упражнения попробуйте реализовать и другие операторы отношений и логические операторы для класса *three_d*.

Подробнее об операторе присваивания

В предыдущей главе рассматривалась потенциальная проблема, связанная с передачей объектов функциям и возвратом объектов из функций. В обоих случаях проблема была вызвана использованием конструктора по умолчанию, который создает побитовую копию объекта. Вспомните, что решение этой проблемы лежит в создании собственного конструктора копии, который точно определяет, как должна быть создана копия объекта.

Подобная проблема может возникать и при присваивании одного объекта другому. По умолчанию объект, находящийся с левой стороны от оператора присваивания, получает побитовую копию объекта, находящегося справа. К печальным последствиям это может привести в случаях, когда при создании объект выделяет некоторый ресурс (например, память), а затем изменяет его или освобождает. Если после выполнения операции присваивания объект изменяет или освобождает этот ресурс, второй объект также изменяется, поскольку он все еще использует его. Решение этой проблемы состоит в перегрузке оператора присваивания.

Чтобы до конца понять суть описанной проблемы, рассмотрим следующую (некорректную) программу.

```
// Ошибка, генерируемая при возврате объекта из функции.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class sample {
   char *s;
  public:
   sample() { s = 0; }
   sample(const sample &ob); // конструктор копии

   ~sample() {
     if(s) delete [] s;
     cout << "Освобождение s-памяти.\n";
   }

   void show() { cout << s << "\n"; }
   void set(char *str);
};

// Конструктор копии.
sample::sample(const sample &ob)
{
  s = new char[strlen(ob.s) +1];
  strcpy(s, ob.s);
}

// Загрузка строки.
void sample::set(char *str)
{
  s = new char[strlen(str) +1];
  strcpy(s, str);
}

// Эта функция возвращает объект типа sample.
sample input()
{
  char instr[80];
  sample str;
  cout << "Введите строку: ";
    cin >> instr;
  str.set(instr);
```

```
  return str;
}

int main()
{
  sample ob;
  // Присваиваем объект, возвращаемый
  // функцией input(), объекту ob.
  ob = input(); // Эта инструкция генерирует ошибку!!!!
  ob.show();
  return 0;
}
```

Возможные результаты выполнения этой программы выглядят так.

```
Введите строку: Привет
Освобождение s-памяти.
Освобождение s-памяти.
Здесь "мусор"
Освобождение s-памяти.
```

В зависимости от используемого компилятора, вы можете увидеть *"мусор"* или нет. Программа может также сгенерировать ошибку во время выполнения. В любом случае ошибки не миновать. И вот почему.

В этой программе конструктор копии корректно обрабатывает возвращение объекта функцией *input()*. Вспомните, что в случае, когда функция возвращает объект, для хранения возвращаемого ею значения создается временный объект. Поскольку при создании объекта-копии конструктор копии выделяет новую область памяти, член *s* исходного объекта и член *s* объекта-копии будут указывать на различные области памяти, которые, следовательно, не станут портить друг друга.

Однако ошибки не миновать, если объект, возвращаемый функцией, присваивается объекту *ob*, поскольку при выполнении присваивания по умолчанию создается побитовая копия. В данном случае временный объект, возвращаемый функцией *input()*, копируется в объект *ob*. В результате член *ob.s* указывает на ту же самую область памяти, что и член *s* временного объекта. Но после присваивания в процессе разрушения временного объекта эта память освобождается. Следовательно, член *ob.s* теперь будет указывать на уже освобожденную память! Более того, память, адресуемая членом *ob.s*, должна быть освобождена и по завершении программы, т.е. во второй раз. Чтобы предотвратить возникновение этой проблемы, необходимо перегрузить оператор присваивания так, чтобы объект, располагаемый слева от оператора присваивания, выделял собственную область памяти.

Реализация этого решения показана в следующей откорректированной программе.

```
// Эта программа работает корректно.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class sample {
   char *s;
```

```
  public:
    sample(); // обычный конструктор
    sample(const sample &ob); // конструктор копии
    ~sample() {
      if(s) delete [] s;
      cout << "Освобождение s-памяти.\n";
    }
    void show() { cout << s << "\n"; }
    void set(char *str);
    sample operator=(sample &ob); // перегруженный оператор присваивания
};

// Обычный конструктор.
sample::sample()
{
  s = new char('\0'); // Член s указывает на null-строку.
}

// Конструктор копии.
sample::sample(const sample &ob)
{
  s = new char[strlen(ob.s)+1];
  strcpy(s, ob.s);
}

// Загрузка строки.
void sample::set(char *str)
{
  s = new char[strlen(str)+1];
  strcpy(s, str);
}

// Перегрузка оператора присваивания.
sample sample::operator=(sample &ob)
{
   /* Если выделенная область памяти имеет недостаточный размер,
выделяется новая область памяти. */
  if(strlen (ob.s) > strlen(s)) {
    delete [] s;
    s = new char[strlen(ob.s)+1];
  }
  strcpy(s, ob.s);
  return *this;
```

```
}

// Эта функция возвращает объект типа sample.
sample input()
{
  char instr[80];
  sample str;
  cout << "Введите строку: ";
    cin >> instr;
  str.set(instr);
  return str;
}

int main()
{
  sample ob;

  // Присваиваем объект, возвращаемый
  // функцией input(), объекту ob.
  ob = input(); // Теперь здесь все в порядке!
  ob.show();

  return 0;
}
```

Эта программа теперь отображает такие результаты (в предположении, что на приглашение *"Введите строку: "* вы введете *"Привет"*).

```
Введите строку: Привет
Освобождение s-памяти.
Освобождение s-памяти.
Освобождение s-памяти.
Привет
Освобождение s-памяти.
```

Как видите, эта программа теперь работает корректно. Вы должны понимать, почему выводится каждое из сообщений *"Освобождение s-памяти. "*. (Подсказка: одно из них вызвано инструкцией *delete* в теле операторной функции *operator=()*.)

Перегрузка оператора индексации массивов ([])

В дополнение к традиционным операторам C++ позволяет перегружать и более "экзотические", например, оператор индексации массивов (*[]*). В C++ (с точки зрения механизма перегрузки) оператор *"[]"* считается бинарным. Его можно перегружать только для класса и только с использованием функции-члена. Вот как выглядит общий формат операторной функции-члена *operator[]()*.

```
тип имя_класса::operator[](int индекс)
{
  // ...
}
```

}

Оператор "[]" перегружается как бинарный оператор.

Формально параметр *индекс* необязательно должен иметь тип *int*, но операторные функции *operator[]()* обычно используются для обеспечения индексации массивов, поэтому в общем случае в качестве аргумента этой функции передается целочисленное значение.

Предположим, у нас определен объект *ob*, тогда выражение

```
ob[3]
```

преобразуется в следующий вызов операторной функции *operator[]()*:

```
ob.operator[](3)
```

Другими словами, значение выражения, заданного в операторе индексации, передается операторной функции *operator[]()* в качестве явно заданного аргумента. При этом указатель *this* будет указывать на объект *ob*, т.е. объект, который генерирует вызов этой функции.

В следующей программе в классе *atype* объявляется массив для хранения трех int-значений. Его конструктор инициализирует каждый член этого массива. Перегруженная операторная функция *operator[]()* возвращает значение элемента, заданного его параметром.

```
// Перегрузка оператора индексации массивов
#include <iostream>
using namespace std;

const int SIZE = 3;

class atype {
   int a[SIZE];
  public:
   atype() {
     register int i;
     for(i=0; i<SIZE; i++) a[i] = i;
   }

   int operator[](int i) {return a[i];}
};

int main()
{
  atype ob;
  cout << ob[2]; // отображает число 2
  return 0;
}
```

Здесь функция *operator[]()* возвращает значение *i*-го элемента массива *a*. Таким образом, выражение *ob[2]* возвращает число *2*, которое отображается инструкцией *cout*. Инициализация массива *a* с помощью конструктора (в этой и следующей программах) выполняется лишь в иллюстративных целях.

Можно разработать операторную функцию *operator[]()* так, чтобы оператор *"[]"* можно было использовать как слева, так и справа от оператора присваивания. Для этого достаточно указать, что значение, возвращаемое операторной функцией *operator[]()*, является ссылкой. Эта возможность демонстрируется в следующей программе.

```
// Возврат ссылки из операторной функции operator()[].
#include <iostream>
```

```
using namespace std;

const int SIZE = 3;

class atype {
   int a[SIZE];
  public:
   atype() {
     register int i;
     for(i=0; i<SIZE; i++) a[i] = i;
   }

   int &operator[](int i) {return a[i];}
};

int main()
{
  atype ob;

  cout << ob[2]; // Отображается число 2.
  cout <<" ";

  ob[2] = 25; // Оператор "[]" стоит слева от оператора "=".
  cout << ob[2]; // Теперь отображается число 25.

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
2 25
```

Поскольку функция *operator[]()* теперь возвращает ссылку на элемент массива, индексируемый параметром *i*, оператор *"[]"* можно использовать слева от оператора присваивания, что позволит модифицировать любой элемент массива. (Конечно же, его по-прежнему можно использовать и справа от оператора присваивания.)

Одно из достоинств перегрузки оператора *"[]"* состоит в том, что с его помощью мы можем обеспечить средство реализации безопасной индексации массивов. Как вы знаете, в C++ возможен выход за границы массива во время выполнения программы без Соответствующего уведомления (т.е. без генерирования сообщения о динамической ошибке). Но если создать класс, который содержит массив, и разрешить доступ к этому массиву только через перегруженный оператор индексации *"[]"*, то возможен перехват индекса, значение которого вышло за дозволенные пределы. Например, следующая программа (в основу которой положен код предыдущей) оснащена средством контроля попадания в допустимый интервал.

```
// Пример организации безопасного массива.

#include <iostream>

#include <cstdlib>

using namespace std;

const int SIZE = 3;
```

```
class atype {
   int a[SIZE];
  public:
   atype( ) {
     register int i;
     for(i=0; i<SIZE; i++) a[i] = i;
   }
  int &operator[] (int i);
};

// Обеспечение контроля попадания в допустимый интервал для класса atype.
int &atype:: operator [](int i)
{
  if(i<0 || i>SIZE-1) {
    cout << "\n Значение индекса ";
    cout << i << " выходит за границы массива. \n";
    exit(1);
  }
  return a[i];
}

int main( )
{
  atype ob;

  cout << ob[2]; // Отображается число 2.
  cout << " ";

  ob[2] =25; // Оператор "[]" стоит в левой части.
  cout << ob[2]; // Отображается число 25.

  ob[3] = 44; // Генерируется ошибка времени выполнения.
  // поскольку значение 3 выходит за границы массива.
  return 0;
}
```

При выполнении эта программа выводит такие результаты.

```
2 25
Значение индекса 3 выходит за границы массива.
```

При выполнении инструкции

```
ob[3] = 44;
```

операторной функцией *operator[]()* перехватывается ошибка нарушения границ массива, после чего программа тут же завершается, чтобы не допустить никаких потенциально возможных разрушений.

Перегрузка оператора "()"

Возможно, самым интригующим оператором, который можно перегружать, является оператор *"()"* (оператор вызова функций). При его перегрузке создается не новый способ вызова функций, а операторная функция, которой можно передать произвольное число параметров. Начнем с примера. Предположим, что некоторый класс содержит следующее объявление перегруженной операторной функции.

```
int operator()(float f, char *p);
```

И если в программе создается объект *ob* этого класса, то инструкция

```
ob (99.57, "перегрузка");
```

преобразуется в следующий вызов операторной функции *operator()*:

```
operator() (99.57, "перегрузка");
```

В общем случае при перегрузке оператора *"()"* определяются параметры, которые необходимо передать функции *operator()*. При использовании оператора *"()"* в программе задаваемые при этом аргументы копируются в эти параметры. Как всегда, объект, который генерирует вызов операторной функции (*ob* в данном примере), адресуется указателем *this*.

Рассмотрим пример перегрузки оператора *"()"* для класса *three_d*. Здесь создается новый объект класса *three_d*, координаты которого представляют собой результаты суммирования соответствующих значений координат вызывающего объекта и значений, передаваемых в качестве аргументов.

```
// Демонстрация перегрузки оператора "()".
#include <iostream>
using namespace std;

class three_d {
    int x, y, z; // 3-мерные координаты
  public:
    three_d() { x = y = z = 0; }
    three_d(int i, int j, int k) {x = i; y = j; z = k; }

    three_d operator()(int a, int b, int c);

    void show();
};

// Перегрузка оператора "()".
three_d three_d::operator()(int a, int b, int c)
{
  three_d temp;
  temp.x = x + a;
  temp.y = y + b;
  temp.z = z + c;
  return temp;
}

// Отображение координат x, y, z.
void three_d::show()
{
```

```
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main()
{
  three_d ob1(1, 2, 3), ob2;

  ob2 = ob1(10, 11, 12); // вызов функции operator()

  cout << "ob1: ";
  ob1.show();

  cout << "ob2: ";
  ob2.show();

  return 0;
}
```

Эта программа генерирует такие результаты.

```
ob1: 1, 2, 3
ob2: 11, 13, 15
```

Не забывайте, что при перегрузке оператора *"()"* можно использовать параметры любого типа, да и сама операторная функция *operator()* может возвращать значение любого типа. Выбор типа должен диктоваться потребностями конкретных программ.

Перегрузка других операторов

За исключением таких операторов, как *new*, *delete*, *->*, *->** и *"запятая"*, остальные C++-операторы можно перегружать таким же способом, который был показан в предыдущих примерах. Перегрузка операторов *new* и *delete* требует применения специальных методов, полное описание которых приводится в главе 17 (она посвящена обработке исключительных ситуаций). Операторы *->*, *->** и *"запятая"* — это специальные операторы, подробное рассмотрение которых выходит за рамки этой книги. Читатели, которых интересуют другие примеры перегрузки операторов, могут обратиться к моей книге *Полный справочник по C++*.

Еще один пример перегрузки операторов

Завершая тему перегрузки операторов, рассмотрим пример, который часто называют квинтэссенцией примеров, посвященных перегрузке операторов, а именно класс строк. Несмотря на то что C++-подход к строкам (которые реализуются в виде символьных массивов с завершающим нулем, а не в качестве отдельного типа) весьма эффективен и гибок, начинающие C++-программисты часто испытывают недостаток в понятийной ясности реализации строк, которая присутствует в таких языках, как BASIC. Конечно же, эту ситуацию нетрудно изменить, поскольку в C++ существует возможность определить класс строк, который будет обеспечивать реализацию строк подобно тому, как это сделано в других языках программирования. По правде говоря, "на заре" развития C++ реализация класса строк была забавой для программистов. И хотя стандарт C++ теперь определяет строковый класс, который описан ниже в этой книге, вам будет полезно реализовать простой вариант такого класса самим. Это упражнение наглядно иллюстрирует мощь механизма перегрузки операторов.

Сначала определим "классовый" тип *str_type*.

```
#include <iostream>
```

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

class str_type {
    char string[80];
  public:
    str_type(char *str = "") { strcpy(string, str); }

    str_type operator+(str_type str); // конкатенация строк
    str_type operator=(str_type str); // присваивание строк

    // Вывод строки
    void show_str() { cout << string; }
};
```

Как видите, в классе *str_type* объявляется закрытый символьный массив *string*, предназначенный для хранения строки. В данном примере условимся, что размер строк не будет превышать *79 байт*. В реальном же классе строк память для их хранения должна выделяться динамически, и это ограничение действовать не будет. Кроме того, чтобы не загромождать логику этого примера, мы решили освободить этот класс (и его функции-члены) от контроля выхода за границы массива. Безусловно, в любой настоящей реализации подобного класса должен быть обеспечен полный контроль за ошибками.

Этот класс имеет один конструктор, который можно использовать для инициализации массива *string* с использованием заданного значения или для присваивания ему пустой строки в случае отсутствия инициализатора. В этом классе также объявляются два перегруженных оператора, которые выполняют конкатенацию и присваивание. Наконец, класс *str_type* содержит функцию *show_str()*, которая выводит строку на экран. Вот как выглядит код операторных функций *operator+()* и *operator=()*.

```
// Конкатенация двух строк.
str_type str_type::operator+(str_type str) {
  str_type temp;
  strcpy(temp.string, string);
  strcat(temp.string, str.string);
  return temp;
}

// Присваивание одной строки другой.
str_type str_type::operator=(str_type str) {
  strcpy(string, str.string);
  return *this;
}
```

Имея определения этих функций, покажем, как их можно использовать на примере следующей функции main().

```
int main()
{
  str_type a("Всем "), b("привет"), c;
  c = a + b;
  c.show_str();
```

```
  return 0;
}
```

При выполнении эта программа выводит на экран строку *Всем привет*. Сначала она конкатенирует строки (объекты класса *str_type*) *a* и *b*, а затем присваивает результат конкатенации строке *c*.

Следует иметь в виду, что операторы *"="* и *"+"* определены только для объектов типа *str_type*. Например, следующая инструкция неработоспособна, поскольку она представляет собой попытку присвоить объекту *a* строку с завершающим нулем.

```
a = "Этого пока делать нельзя.";
```

Но класс *str_type*, как будет показано ниже, можно усовершенствовать и разрешить выполнение таких инструкций.

Для расширения круга операций, поддерживаемых классом *str_type* (например, чтобы можно было объектам типа *str_type* присваивать строки с завершающим нулем или конкатенировать строку с завершающим нулем с объектом типа *str_type*), необходимо перегрузить операторы *"="* и *"+"* еще раз. Вначале изменим объявление класса.

```
class str_type {
   char string{80];
  public:
   str_type(char *str = "") { strcpy(string, str); }

    str_type operator+(str_type str); /* конкатенация объектов типа str_type*/

    str_type operator+(char *str); /* конкатенация объекта класса str_type со строкой с завершающим нулем */

   str_type operator=(str_type str); /* присваивание одного объекта типа str_type другому */

    char *operator=(char *str); /* присваивание строки с завершающим нулём объекту типа str_type */

   void show_str() { cout << string; }
};
```

Затем реализуем перегрузку операторных функций *operator+()* и *operator=()*.

```
// Присваивание строки с завершающим нулем объекту типа str_type.
str_type str_type::operator=(char *str)
{
  str_type temp;
  strcpy(string, str);
  strcpy(temp.string, string);
  return temp;
}

// Конкатенация строки с завершающим нулем с объектом типа str_type.
str_type str_type::operator+(char *str)
{
  str_type temp;
  strcpy(temp.string, string);
```

```
  strcpy(temp.string, string);
  strcat(temp.string, str);
  return temp;
}
```

Внимательно рассмотрите код этих функций. Обратите внимание на то, что правый аргумент является не объектом типа *str_type*, а указателем на символьный массив с завершающим нулем, т.е. обычной C++-строкой. Но обе эти функции возвращают объект типа *str_type*. И хотя теоретически они могли бы возвращать объект любого другого типа, весь смысл их существования и состоит в том, чтобы возвращать объект типа *str_type*, поскольку результаты этих операций принимаются также объектами типа *str_type*. Достоинство определения строковой операции, в которой в качестве правого операнда участвует строка с завершающим нулем, заключается в том, что оно позволяет писать некоторые инструкции в естественной форме. Например, следующие инструкции вполне законны.

```
str_type a, b, c;
a = "Привет всем"; /* присваивание строки с завершающим нулем объекту */
c = a + " Георгий"; /* конкатенация объекта со строкой с завершающим нулем */
```

Следующая программа включает дополнительные определения операторов *"="* и *"+"*.

```
// Усовершенствование строкового класса.
#include <iostream>
#include <cstring>
using namespace std;

class str_type {
    char string[80];
  public:
    str_type(char *str = "") { strcpy(string, str); }

    str_type operator+(str_type str);
    str_type operator+(char *str);
    str_type operator=(str_type str);
    str_type operator=(char *str);

    void show_str() { cout << string; }
};

str_type str_type::operator+(str_type str)
{
  str_type temp;
  strcpy(temp.string, string);
  strcat(temp.string, str.string);
  return temp;
}

str_type str_type::operator=(str_type str)
{
  strcpy(string, str.string);
```

```
  strcpy(string, str.string);
  return *this;
}

str_type str_type::operator=(char *str)
{
  str_type temp;
  strcpy(string, str);
  strcpy(temp.string, string);
  return temp;
}

str_type str_type::operator+(char *str)
{
  str_type temp;
  strcpy(temp.string, string);
  strcat(temp.string, str);
  return temp;
}

int main()
{
  str_type a("Привет "), b("всем"), c;

  c = a + b;
  c.show_str();
  cout << "\n";

  a = "для программирования, потому что";
  a.show_str();
  cout << "\n";

  b = c = "C++ это супер";
  c = c + " " + a + " " +b;
  c.show_str();
  return 0;
}
```

При выполнении эта программа отображает на экране следующее.

```
Привет всем
для программирования, потому что
C++ это супер для программирования, потому что C++ это супер
```

Прежде чем переходить к следующей главе, убедитесь в том, что до конца понимаете, как получены эти результаты. Теперь вы можете сами определять другие операции над строками. Попытайтесь, например, определить операцию удаления подстроки на основе оператора *"-"*. Так, если строка объекта *A* содержит фразу *"Это трудный-трудный тест"*, а строка объекта *B* — фразу *"трудный"*, то вычисление выражения *A-B* даст в результате *"Это - тест"*. В данном случае из исходной строки были удалены все вхождения подстроки *"трудный"*. Определите также *"дружественную"* функцию, которая бы позволяла строке с

"трубный". Определите также *"дружественную"* функцию, которая бы позволяла строке с завершающим нулем находиться слева от оператора *"+"*. Наконец, добавьте в программу код, обеспечивающий контроль за ошибками.

Важно! *Для создаваемых вами классов всегда имеет смысл экспериментировать с перегрузкой операторов. Как показывают примеры этой главы, механизм перегрузки операторов можно использовать для добавления новых типов данных в среду программирования. Это одно из самых мощных средств C++.*

Глава 14: Наследование

Наследование — один из трех фундаментальных принципов объектно-ориентированного программирования, поскольку именно благодаря ему возможно создание иерархических классификаций. Используя наследование, можно создать общий класс, который определяет характеристики, присущие множеству связанных элементов. Этот класс затем может быть унаследован другими, узкоспециализированными классами с добавлением в каждый из них своих, уникальных особенностей.

В стандартной терминологии языка C++ класс, который наследуется, называется базовым. Класс, который наследует базовый класс, называется производным. Производный класс можно использовать в качестве базового для другого производного класса. Таким путем и строится многоуровневая иерархия классов.

Понятие о наследовании

Базовый класс наследуется производным классом.

Язык C++ поддерживает механизм наследования, позволяя в объявление класса встраивать другой класс. Для этого базовый класс задается при объявлении производного. Лучше всего начать с примера. Рассмотрим класс *road_vehicle*, который в самых общих чертах определяет дорожное транспортное средство. Его члены данных позволяют хранить количество колес и число пассажиров, которое может перевозить транспортное средство.

```
class road_vehicle {
   int wheels;
   int passengers;
 public:
   void set_wheels(int num) { wheels = num; }
   int get_wheels( ) { return wheels; }
   void set_pass(int num) { passengers = num; }
   int get_pass( ) { return passengers; }
};
```

Это общее определение дорожного транспортного средства можно использовать для определения конкретных типов транспортных средств. Например, в следующем фрагменте путем наследования класса *road_vehicle* создается класс *truck* (грузовых автомобилей).

```
class truck : public road_vehicle {
   int cargo;
 public:
   void set_cargo(int size) { cargo = size; }
   int get_cargo( ) { return cargo; }
   void show( );
};
```

Тот факт, что класс *truck* наследует класс *road_vehicle*, означает, что класс *truck* наследует все содержимое класса *road_vehicle*. К содержимому класса *road_vehicle* класс *truck* добавляет

член данных *cargo*, а также функции-члены, необходимые для поддержки члена *cargo*.

Обратите внимание на то, как наследуется класс *road_vehicle*. Общий формат для обеспечения наследования имеет следующий вид.

```
class имя_производного_класса : доступ имя_базового_класса
{
  тело нового класса
}
```

Здесь элемент *доступ* необязателен. При необходимости он может быть выражен одним из спецификаторов доступа: *public*, *private* или *protected*. Подробнее об этих спецификаторах доступа вы узнаете ниже в этой главе. А пока в определениях всех наследуемых классов мы будем использовать спецификатор *public*. Это означает, что все *public*-члены базового класса также будут *public*-членами производного класса. Следовательно, в предыдущем примере члены класса *truck* имеют доступ к открытым функциям-членам класса *road_vehicle*, как будто они (эти функции) были объявлены в теле класса *truck*. Однако класс *truck* не имеет доступа к *private*-членам класса *road_vehicle*. Например, для класса *truck* закрыт доступ к члену данных *wheels*.

Рассмотрим программу, которая использует механизм наследования для создания двух подклассов класса *road_vehicle*: *truck* и *automobile*.

```
// Демонстрация наследования.
#include <iostream>
using namespace std;

// Определяем базовый класс транспортных средств.
class road_vehicle {
   int wheels;
   int passengers;
  public:
   void set_wheels(int num) { wheels = num; }
   int get_wheels() { return wheels; }
   void set_pass(int num) { passengers = num; }
   int get_pass() { return passengers; }
};

// Определяем класс грузовиков.
class truck : public road_vehicle {
   int cargo;
  public:
   void set_cargo(int size) { cargo = size; }
   int get_cargo() { return cargo; }
   void show();
};

enum type {car, van, wagon};

// Определяем класс автомобилей.
class automobile : public road_vehicle {
```

```
class automobile : public road_vehicle {
    enum type car_type;
  public:
    void set_type(type t) { car_type = t; }
    enum type get_type() { return car_type; }
    void show();
};

void truck::show()
{
  cout << "колес: " << get_wheels() << "\n";
  cout << "пассажиров: " << get_pass() << "\n";
  cout << "грузовместимость в кубических футах: " << cargo << "\n";
}

void automobile::show()
{
  cout << "колес: " << get_wheels() << "\n";
  cout << "пассажиров: " << get_pass() << "\n";
  cout << "тип: ";

  switch(get_type()) {
    case van: cout << "автофургон\n";
      break;
    case car: cout << "легковой\n";
      break;
    case wagon: cout << "фура\n";
  }
}

int main()
{
  truck t1, t2;
  automobile c;

  t1.set_wheels(18);
  t1.set_pass(2);
  t1.set_cargo(3200);

  t2.set_wheels(6);
  t2.set_pass(3);
  t2.set_cargo(1200);

  t1.show();
  cout << "\n";
  t2.show();
```

```
  cout << "\n";
  c.set_wheels(4);
  c.set_pass(6);
  c.set_type(van);
  c.show();
  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
колес: 18
пассажиров: 2
грузовместимость в кубических футах: 3200

колес: 6
пассажиров: 3
грузовместимость в кубических футах: 1200

колес: 4
пассажиров: 6
тип: автофургон
```

Как видно по результатам выполнения этой программы, основное достоинство наследования состоит в том, что оно позволяет создать базовый класс, который затем можно включить в состав более специализированных классов. Таким образом, каждый производный класс может служить определенной цели и при этом оставаться частью общей классификации.

И еще. Обратите внимание на то, что оба класса *truck* и *automobile* включают функцию-член *show()*, которая отображает информацию об объекте. Эта функция демонстрирует еще один аспект объектно-ориентированного программирования — полиморфизм. Поскольку каждая функция *show()* связана с собственным классом, компилятор может легко "понять", какую именно функцию нужно вызвать для данного объекта.

После ознакомления с общей процедурой наследования одним классом другого можно перейти и к деталям этой темы.

Управление доступом к членам базового класса

Если один класс наследует другой, члены базового класса становятся членами производного. Статус доступа членов базового класса в производном классе определяется спецификатором доступа, используемым для наследования базового класса. Спецификатор доступа базового класса выражается одним из ключевых слов: *public*, *private* или *protected*. Если спецификатор доступа не указан, то по умолчанию используется спецификатор *private*, если речь идет о наследовании типа *class*. Если же наследуется тип *struct*, то при отсутствии явно заданного спецификатора доступа по умолчанию используется спецификатор *public*. Рассмотрим рамификацию (разветвление) использования спецификаторов *public* или *private*. (Спецификатор *protected* описан в следующем разделе.)

Если базовый класс наследуется как public-класс, его public-члены становятся public-членами производного класса.

Если базовый класс наследуется как *public*-класс, все его *public*-члены становятся *public*-членами производного класса. Во всех случаях *private*-члены базового класса остаются закрытыми в рамках этого класса и не доступны для членов производного. Например, в следующей программе *public*-члены класса *base* становятся *public*-членами класса *derived*.

Следовательно, они будут доступны и для других частей программы.

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

class base {
   int i, j;
  public:
   void set (int a, int b) { i = a; j = b; }
   void show( ) { cout << i << " " << j << "\n"; }
};

class derived : public base {
   int k;
  public:
   derived(int x) { k = x; }
   void showk( ) { cout << k << "\n"; }
};

int main( )
{
  derived ob(3);
  ob.set(1, 2); // доступ к членам класса base
  ob.show( ); // доступ к членам класса base
  ob.showk( ); // доступ к члену класса derived
  return 0;
}
```

Поскольку функции *set()* и *show()* (члены класса base) унаследованы классом *derived* как *public*-члены, их можно вызывать для объекта типа *derived* в функции main(). Поскольку члены данных *i* и *j* определены как *private*-члены, они остаются закрытыми в рамках своего класса *base*.

Если базовый класс наследуется как private-класс, его public-члены становятся private-членами производного класса.

Противоположностью открытому (*public*) наследованию является закрытое (*private*). Если базовый класс наследуется как *private*-класс, все его *public*-члены становятся *private*-членами производного класса. Например, следующая программа не скомпилируется, поскольку обе функции *set()* и *show()* теперь стали *private*-членами класса *derived*, и поэтому их нельзя вызывать из функции main().

```
// Эта программа не скомпилируется.
#include <iostream>
using namespace std;

class base {
   int i, j;
  public:
   void set (int a, int b) { i = a; j = b; }
   void show( ) { cout << i << " " << j << "\n"; }
```

```
};

// Открытые члены класса base теперь становятся
// закрытыми членами класса derived.
class derived : private base {
   int k;
  public:
   derived(int x) { k = x; }
   void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob (3);
  ob.set(1, 2); // Ошибка, доступа к функции set() нет.
  ob.show(); // Ошибка, доступа к функции show() нет.
  return 0;
}
```

Важно запомнить, что в случае, если базовый класс наследуется как *private*-класс, его открытые члены становятся закрытыми (*private*) членами производного класса. Это означает, что они доступны для членов производного класса, но не доступны для других частей программы.

Использование защищенных членов

Член класса может быть объявлен не только открытым (*public*) или закрытым (*private*), но и защищенным (*protected*). Кроме того, базовый класс в целом может быть унаследован с использованием спецификатора *protected*. Ключевое слово *protected* добавлено в C++ для предоставления механизму наследования большей гибкости.

Если член класса объявлен с использованием спецификатора *protected*, он не будет доступен для других элементов программы, которые не являются членами данного класса. С одним важным исключением доступ к защищенному члену идентичен доступу к закрытому члену, т.е. к нему могут обращаться только другие члены того же класса. Единственное исключение из этого правила проявляется при наследовании защищенного члена. В этом случае защищенный член существенно отличается от закрытого.

Спецификатор доступа protected объявляет защищенные члены или обеспечивает наследование защищенного класса.

Как вы знаете, закрытый член базового класса не доступен никаким другим частям программы, включая и производные классы. Однако с защищенными членами все обстоит иначе. Если базовый класс наследуется как *public*-класс, защищенные члены базового класса становятся защищенными членами производного класса, т.е. доступными для производного класса. Следовательно, используя спецификатор *protected*, можно создать члены класса, которые закрыты в рамках своего класса, но которые может унаследовать производный класс, причем с получением доступа к ним.

Рассмотрим следующий пример программы.

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

class base {
```

```
  protected:
    int i, j; // Эти члены закрыты в классе base, но доступны для класса
derived.
  public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
    int k;
  public:
    // Класс derived имеет доступ к членам класса base i и j.
    void setk() { k = i*j; }
    void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob;
  ob.set(2, 3); // OK, классу derived это позволено.
  ob.show(); // OK, классу derived это позволено.
  ob.setk();
  ob.showk();
  return 0;
}
```

Поскольку класс *base* унаследован классом *derived* открытым способом (т.е. как public-класс), и поскольку члены *i* и *j* объявлены защищенными в классе *base*, функция *setk()* (член класса derived) может получать к ним доступ. Если бы члены *i* и *j* были объявлены в классе *base* закрытыми, то класс *derived* не мог бы обращаться к ним, и эта программа не скомпилировалась бы.

Узелок на память. *Спецификатор protected позволяет создать член класса, который будет доступен в рамках данной иерархии классов, но закрыт для остальных элементов программы.*

Если некоторый производный класс используется в качестве базового для другого производного класса, то любой защищенный член исходного базового класса, который наследуется (открытым способом) первым производным классом, может быть унаследован еще раз (в качестве защищенного члена) вторым производным классом. Например, в следующей (вполне корректной) программе класс *derived2* имеет законный доступ к членам *i* и *j*.

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

class base {
  protected:
    int i, j;
  public:
    void set(int a, int b) { i = a; j = b; }
```

```
    void show( ) { cout << i << " " << j << "\n"; }
};

// Члены i и j наследуются как protected-члены.
class derived1: public base {
    int k;
  public:
    void setk( ) { к = i*j; } // правомерный доступ
    void showk( ) { cout << к << "\n"; }
};

// Члены i и j наследуются косвенно через класс derived1.
class derived2 : public derived1 {
    int m;
  public :
    void setm( ) { m = i-j; } // правомерный доступ
    void showm( ) { cout << m << "\n"; }
};

int main( )
{
  derived1 ob1;
  derived2 ob2;

  ob1.set (2, 3);
  ob1.show( );
  ob1.setk( );
  ob1.showk( );

  ob2.set (3, 4);
  ob2.show( );
  ob2.setk( );
  ob2.setm( );
  ob2.showk( );
  ob2.showm( );

  return 0;
}
```

Если базовый класс наследуется закрытым способом (т.е. с использованием спецификатора *private*), защищенные (*derived*) члены этого базового класса становятся закрытыми (*private*) членами производного класса. Следовательно, если бы в предыдущем примере класс *base* наследовался закрытым способом, то все его члены стали бы *private*-членами класса *derived1*, и в этом случае они не были бы доступны для класса *derived2*. (Однако члены *i* и *j* по-прежнему остаются доступными для класса *derived1*.) Эта ситуация иллюстрируется в следующей программе, которая поэтому некорректна (и не скомпилируется). Все ошибки отмечены в комментариях.

отмечены в комментариях.

```
// Эта программа не скомпилируется.
#include <iostream>
using namespace std;

class base {
  protected:
    int i, j;
  public:
    void set (int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};

// Теперь все элементы класса base будут закрыты
// в рамках класса derived1.
class derived1 : private base {
    int k;
  public:
    // Вызовы этих функций вполне законны, поскольку
    // переменные i и j являются одновременно
    // private-членами класса derived1.
    void setk() { k = i*j; } // OK
    void showk() { cout << k << "\n"; }
};

// Доступ к членам i, j, set() и show() не наследуется.
class derived2 : public derived1 {
    int m;
  public :
    // Неверно, поскольку члены i и j закрыты в рамках
    // класса derived1.
    void setm() { m = i-j; } // ошибка
    void showm() { cout << m << "\n"; }
};

int main()
{
  derived1 ob1;
  derived2 ob2;

  ob1.set(1, 2); // Ошибка: нельзя вызывать функцию set().
  ob1.show(); // Ошибка: нельзя вызывать функцию show().

  ob2.set(3, 4); // Ошибка: нельзя вызывать функцию set().
  ob2.show(); // Ошибка: нельзя вызывать функцию show().

  return 0;
}
```

```
  return 0;
}
```

Несмотря на то что класс *base* наследуется классом *derived1* закрытым способом, класс *derived1*, тем не менее, имеет доступ к *public*- и *protected*-членам класса *base*. Однако он не может эту привилегию передать дальше, т.е. вниз по иерархии классов. Ключевое слово *protected*— это часть языка C++. Оно обеспечивает механизм защиты определенных элементов класса от модификации функциями, которые не являются членами этого класса, но позволяет передавать их *"по наследству"*.

Спецификатор *protected* можно также использовать в отношении структур. Но его нельзя применять к объединениям, поскольку объединение не наследуется другим классом. (Некоторые компиляторы допускают использование спецификатора *protected* в объявлении объединения, но, поскольку объединения не могут участвовать в наследовании, в этом контексте ключевое слово *protected* будет равносильным ключевому слову *private*.)

Спецификатор защищенного доступа может стоять в любом месте объявления класса, но, как правило, *protected*-члены объявляются после (объявляемых по умолчанию) *private*-членов и перед *public*-членами. Таким образом, самый общий формат объявления класса обычно выглядит так.

```
class имя_класса {
    private-члены
  protected:
    protected-члены
  public:
    public-члены
};
```

Напомню, что раздел защищенных членов необязателен.

Использование спецификатора protected для наследования базового класса

Спецификатор *protected* можно использовать не только для придания членам класса статуса "защищенности", но и для наследования базового класса. Если базовый класс наследуется как защищенный, все его открытые и закрытые члены становятся защищенными членами производного класса. Рассмотрим пример.

```
// Демонстрация наследования защищенного базового класса.
#include <iostream>
using namespace std;

class base {
    int i;
  protected:
    int j;
  public:
    int k;
    void seti(int a) { i = a; }
    int geti() { return i; }
};

// Наследуем класс base как protected-класс.
class derived : protected base {
```

```
  public:
    void setj(int a) { j = a; } // j — здесь protected-член
    void setk(int a) { k = a; } // k — здесь protected-член
    int getj() { return j; }
    int getk() { return k; }
};

int main()
{
  derived ob;
  /* Следующая строка неправомочна, поскольку функция seti() является protected-членом класса derived, что делает ее недоступной за его пределами. */
  // ob.seti (10);
  // cout << ob.geti(); // Неверно, поскольку функция geti() — protected-член.
  //ob.k=10; // Неверно, поскольку переменная k — protected-член.

  // Следующие инструкции правомочны.
    ob.setk(10);
    cout << ob.getk() << ' ';
    ob.setj(12);
    cout << ob.getj() << ' ';
    return 0;
}
```

Как отмечено в комментариях к этой программе, члены (класса base) *k*, *j*, *seti()* и *geti()* становятся *protected*-членами класса *derived*. Это означает, что к ним нельзя получить доступ из кода, "прописанного" вне класса *derived*. Поэтому ссылки на эти члены в функции main() (через объект *ob*) неправомочны.

Об использовании спецификаторов public, protected и private

Поскольку права доступа, определяемые спецификаторами *public*, *protected* и *private*, принципиальны для программирования на C++, имеет смысл обобщить все, что мы уже знаем об этих ключевых словах.

При объявлении члена класса открытым (с использованием ключевого слова *public*) к нему можно получить доступ из любой другой части программы. Если член класса объявляется закрытым (с помощью спецификатора *private*), к нему могут получать доступ только члены того же класса. Более того, к закрытым членам базового класса не имеют доступа даже производные классы. Если же член класса объявляется защищенным (*protected*-членом), к нему могут получать доступ только члены того же или производных классов. Таким образом, спецификатор *protected* позволяет наследовать члены, но оставляет их закрытыми в рамках иерархии классов.

Если базовый класс наследуется с использованием ключевого слова *public*, его *public*-члены становятся *public*-членами производного класса, а его *protected*-члены — *protected*-членами производного класса.

Если базовый класс наследуется с использованием спецификатора *protected*, его *public*- и *protected*-члены становятся *protected*-членами производного класса.

Если базовый класс наследуется с использованием ключевого слова *private*, его *public*- и

protected-члены становятся *private*-членами производного класса.

Во всех случаях *private*-члены базового класса остаются закрытыми в рамках этого класса и не наследуются.

По мере увеличения вашего опыта в программировании на C++ применение спецификаторов *public*, *protected* и *private* не будет доставлять вам хлопот. А пока, если вы еще не уверены в правильности использования того или иного спецификатора доступа, напишите простую экспериментальную программу и проанализируйте полученные результаты.

Наследование нескольких базовых классов

Производный класс может наследовать два или больше базовых классов. Например, в этой короткой программе класс *derived* наследует оба класса *base1* и *base2*.

```
// Пример использования нескольких базовых классов.
#include <iostream>
using namespace std;

class base1 {
  protected:
    int x;
  public:
    void showx() { cout << x << "\n"; }
};

class base2 {
  protected:
    int y;
  public:
    void showy() { cout << y << "\n"; }
};

// Наследование двух базовых классов.
class derived: public base1, public base2 {
  public:
    void set(int i, int j) { x = i; y = j; }
};

int main()
{
  derived ob;
  ob.set (10, 20); // член класса derived
  ob.showx(); // функция из класса base1
  ob.showy(); // функция из класса base2
  return 0;
}
```

Как видно из этого примера, чтобы обеспечить наследование нескольких базовых классов, необходимо через запятую перечислить их имена в виде списка. При этом нужно указать

необходимо через запятую перечислить их имена в виде списка. При этом нужно указать спецификатор доступа для каждого наследуемого базового класса.

Конструкторы, деструкторы и наследование

При использовании механизма наследования обычно возникает два важных вопроса, связанных с конструкторами и деструкторами. Первый: когда вызываются конструкторы и деструкторы базового и производного классов? Второй: как можно передать параметры конструктору базового класса? Ответы на эти вопросы изложены в следующем разделе.

Когда выполняются конструкторы и деструкторы

Базовый и/или производный класс может содержать конструктор и/или деструктор. Важно понимать порядок, в котором выполняются эти функции при создании объекта производного класса и его (объекта) разрушении.

Рассмотрим короткую программу.

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

class base {
  public:
    base( ) { cout <<"Создание base-объекта.\n"; }
    ~base( ) { cout <<"Разрушение base-объекта.\n"; }
};

class derived: public base {
  public:
    derived( ) { cout <<"Создание derived-объекта.\n"; }
    ~derived( ) { cout <<"Разрушение derived-объекта.\n"; }
};

int main( )
{
  derived ob;
  // Никаких действий, кроме создания и разрушения объекта ob.
  return 0;
}
```

Как отмечено в комментариях для функции main(), эта программа лишь создает и тут же разрушает объект *ob*, который имеет тип *derived*. При выполнении программа отображает такие результаты.

```
Создание base-объекта.
Создание derived-объекта.
Разрушение derived-объекта.
Разрушение base-объекта.
```

Судя по результатам, сначала выполняется конструктор класса *base*, а за ним — конструктор класса *derived*. Затем (по причине немедленного разрушения объекта *ob* в этой программе) вызывается деструктор класса *derived*, а за ним — деструктор класса *base*.

Конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке.

Результаты вышеописанного эксперимента можно обобщить следующим образом. При создании объекта производного класса сначала вызывается конструктор базового класса, а

за ним — конструктор производного класса. При разрушении ооъекта производного класса сначала вызывается его "родной" конструктор, а за ним — конструктор базового класса. Другими словами, конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке.

Вполне логично, что функции конструкторов выполняются в порядке происхождения их классов. Поскольку базовый класс "ничего не знает" ни о каком производном классе, операции по инициализации, которые ему нужно выполнить, не зависят от операций инициализации, выполняемых производным классом, но, возможно, создают предварительные условия для последующей работы. Поэтому конструктор базового класса должен выполняться первым.

Аналогичная логика присутствует и в том, что деструкторы выполняются в порядке, обратном порядку происхождения классов. Поскольку базовый класс лежит в основе производного класса, разрушение базового класса подразумевает разрушение производного. Следовательно, деструктор производного класса имеет смысл вызвать до того, как объект будет полностью разрушен.

При расширенной иерархии классов (т.е. в ситуации, когда производный класс становится базовым классом для еще одного производного) применяется следующее общее правило: конструкторы вызываются в порядке происхождения классов, а деструкторы — в обратном порядке. Например, при выполнении этой программы

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

class base {
  public:
    base( ) { cout <<"Создание base-объекта.\n"; }
    ~base( ){ cout <<"Разрушение base-объекта.\n"; }
};

class derived1 : public base {
  public:
    derived1( ) { cout <<"Создание derived1-объекта.\n"; }
    ~derived1( ){ cout <<"Разрушение derived1-объекта.\n"; }
};

class derived2: public derived1 {
  public:
    derived2( ) { cout <<"Создание derived2-объекта.\n"; }
    ~derived2( ){ cout <<"Разрушение derived2-объекта.\n"; }
};

int main( )
{
  derived2 ob;
  // Создание и разрушение объекта ob.
  return 0;
}
```

отображаются такие результаты:

```
Создание base-объекта.
Создание derived1-объекта.
Создание derived2-объекта.
Разрушение derived2-объекта.
Разрушение derived1-объекта.
Разрушение base-объекта.
```

То же общее правило применяется и в ситуациях, когда производный класс наследует несколько базовых классов. Например, при выполнении этой программы

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

class base1 {
  public:
    base1( ) { cout <<"Создание base1-объекта.\n"; }
    ~base1( ){ cout <<"Разрушение base1-объекта.\n"; }
};

class base2 {
  public:
    base2( ) { cout <<"Создание base2-объекта.\n"; }
    ~base2( ){ cout <<"Разрушение base2-объекта.\n"; }
};

class derived: public base1, public base2 {
  public:
    derived( ) { cout <<"Создание derived-объекта.\n"; }
    ~derived( ){ cout <<"Разрушение derived-объекта.\n"; }
};

int main( )
{
  derived ob;
  // Создание и разрушение объекта ob.
  return 0;
}
```

генерируются такие результаты:

```
Создание base1-объекта.
Создание base2-объекта.
Создание derived-объекта.
Разрушение derived-объекта.
Разрушение base2-объекта.
Разрушение base1-объекта.
```

Как видите, конструкторы вызываются в порядке происхождения их классов, слева направо, в порядке их задания в списке наследования для класса *derived*. Деструкторы вызываются в обратном порядке, справа налево. Это означает, что если бы класс *base2* стоял перед

классом *base1* в списке класса *derived*, т.е. в соответствии со следующей инструкцией:

```
class derived: public base2, public base1 {};
```

то результаты выполнения предыдущей программы были бы такими:

```
Создание base2-объекта.
Создание base1-объекта.
Создание derived-объекта.
Разрушение derived-объекта.
Разрушение base1-объекта.
Разрушение base2-объекта.
```

Передача параметров конструкторам базового класса

До сих пор ни один из предыдущих примеров не включал конструкторы, для которых требовалось бы передавать аргументы. В случаях, когда конструктор лишь производного класса требует передачи одного или нескольких аргументов, достаточно использовать стандартный синтаксис параметризованного конструктора. Но как передать аргументы конструктору базового класса? В этом случае необходимо использовать расширенную форму объявления конструктора производного класса, в которой предусмотрена возможность передачи аргументов одному или нескольким конструкторам базового класса. Вот как выглядит общий формат такого расширенного объявления.

```
конструктор_производного_класса (список_аргументов) :
  base1 (список_аргументов),
  base2 (список_аргументов),
  .
  .
  .
  baseN {список_аргументов);
{
  тело конструктора производного класса
}
```

Здесь элементы *base1-baseN* означают имена базовых классов, наследуемых производным классом. Обратите внимание на то, что объявление конструктора производного класса отделяется от списка базовых классов двоеточием, а имена базовых классов разделяются запятыми (в случае наследования нескольких базовых классов).

Рассмотрим следующую простую программу.

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

class base {
  protected:
    int i;
  public:
    base (int x) {
      i = x;
      cout << "Создание base-объекта.\n";
    }
```

```
    ~base( ) {cout << "Разрушение base1-объекта.\n";}
};

class derived: public base {
    int j;
  public:
     // Класс derived использует параметр x, а параметр y передается конструктору класса base.
    derived(int x, int y): base(y){
      j = x;
      cout << "Создание derived-объекта.\n";
    }

    ~derived( ) { cout << "Разрушение derived-объекта.\n"; }

    void show( ) { cout << i << " " << j << "\n"; }
};

int main( )
{
  derived ob(3, 4);
  ob.show( ); // отображает числа 4 3
  return 0;
}
```

Здесь конструктор класса *derived* объявляется с двумя параметрами, *x* и *y*. Однако конструктор *derived()* использует только параметр *x*, а параметр *y* передается конструктору *base()*. В общем случае конструктор производного класса должен объявлять параметры, которые принимает его класс, а также те, которые требуются базовому классу. Как показано в предыдущем примере, любые параметры, требуемые базовым классом, передаются ему в списке аргументов базового класса, указываемого после двоеточия.

Рассмотрим пример программы, в которой демонстрируется наследование нескольких базовых классов.

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

class base1 {
  protected:
    int i;
  public:
    base1(int x) {
      i = x;
      cout << "Создание base1-объекта.\n";
    }

    ~base1( ) { cout << "Разрушение base1-объекта.\n"; }
};
```

```
class base2 {
  protected:
    int k;
  public:
    base2(int x) {
      k = x;
      cout << "Создание base2-объекта.\n";
    }
    ~base2() { cout << "Разрушение base2-объекта.\n"; }
};
class derived: public base1, public base2 {
    int j;
  public:
    derived(int x, int y, int z): base1(y), base2(z){
      j = x;
      cout << "Создание derived-объекта.\n";
    }
    ~derived() { cout << "Разрушение derived-объекта.\n"; }
    void show() { cout << i << " "<< j << " "<< k << " \ n"; }
};
int main()
{
  derived ob(3, 4, 5);
  ob.show(); // отображает числа 4 3 5
  return 0;
}
```

Важно понимать, что аргументы для конструктора базового класса передаются через аргументы, принимаемые конструктором производного класса. Поэтому, даже если конструктор производного класса не использует никаких аргументов, он, тем не менее, должен объявить один или несколько аргументов, если базовый класс принимает один или несколько аргументов. В этой ситуации аргументы, передаваемые производному классу, "транзитом" передаются базовому. Например, в следующей программе конструкторы *base1()* и *base2()*, в отличие от конструктора класса *derived*, принимают аргументы.

```
#include <iostream>
using namespace std;
class base1 {
  protected:
    int i;
  public:
    base1(int x) {
```

```
  base1(int x) {
    i=x;
    cout << "Создание base1-объекта.\n";
  }

  ~base1() { cout << "Разрушение base1-объекта.\n"; }
};

class base2 {
  protected:
    int k;
  public:
    base2(int x) {
      k = x;
      cout << "Создание base2-объекта.\n";
    }

    ~base2() { cout << "Разрушение base2-объекта.\n"; }
};

class derived: public base1, public base2 {
  public:
    /* Конструктор класса derived не использует параметров, но должен
объявить их, чтобы передать конструкторам базовых классов.
    */
    derived(int x, int y): base1(x), base2(y){
      cout << "Создание derived-объекта.\n";
    }

    ~derived() { cout << "Разрушение derived-объекта.\n"; }

    void show() { cout << i << " " << k << "\n"; }
};

int main()
{
  derived ob(3, 4);
  ob.show(); // отображает числа 3 4
  return 0;
}
```

Конструктор производного класса может использовать любые (или все) параметры, которые им объявлены для приема, независимо от того, передаются ли они (один или несколько) базовому классу. Другими словами, тот факт, что некоторый аргумент передается базовому классу, не мешает его использованию и самим производным классом. Например, этот фрагмент кода совершенно допустим.

```
class derived: public base {
    int j;
  public:
```

```
  public:
    // Класс derived использует оба параметра x и y а также передает их
классу base.
    derived(int x, int y): base(x, y){
      j = x*y;
      cout << "Создание derived-объекта.\n";
    }
// . . .
};
```

При передаче аргументов конструкторам базового класса следует иметь в виду, что передаваемый аргумент может содержать любое (действительное на момент передачи) выражение, включающее вызовы функций и переменные. Это возможно благодаря тому, что C++ позволяет выполнять динамическую инициализацию данных.

Предоставление доступа

Когда базовый класс наследуется закрытым способом (как private-класс), все его члены (открытые, защищенные и закрытые) становятся private-членами производного класса. Но при определенных обстоятельствах один или несколько унаследованных членов необходимо вернуть к их исходной спецификации доступа. Например, несмотря на то, что базовый класс наследуется как *private*-класс, определенным его *public*-членам нужно предоставить *public*-статус в производном классе. Это можно сделать двумя способами. Во-первых, в производном классе можно использовать объявление *using* (этот способ рекомендован стандартом C++ для использования в новом коде). Но мы отложили рассмотрение директивы *using* до темы пространств имен. (Основное назначение директивы *using*— обеспечить поддержку пространств имен.) Во-вторых, можно настроить доступ к унаследованному члену с помощью объявлений доступа. Объявления доступа все еще поддерживаются стандартом C++, но в последнее время активизировались возражения против их применения, а это значит, что их не следует использовать в новом коде. Поскольку они все еще используются в C++-коде, мы уделим внимание этой теме.

Объявление доступа имеет такой формат:

```
имя_базового_класса::член;
```

Объявление доступа восстанавливает уровень доступа унаследованного члена, в результате чего он получает тот уровень доступа, который был у него в базовом классе.

Объявление доступа помещается в производном классе под соответствующим спецификатором доступа. Обратите внимание на то, что объявления типа в этом случае указывать не требуется.

Чтобы понять, как работает объявление доступа, рассмотрим сначала этот короткий фрагмент кода.

```
class base {
  public:
    int j; // public-доступ в классе base
};

// Класс base наследуется как private-класс.
class derived: private base {
  public:
    // Вот использование объявления доступа:
    base::j; // Теперь член j снова стал открытым.
```

```
// . . .
};
```

Поскольку класс *base* наследуется классом *derived* закрытым способом, его *public*-переменная *j* становится *private*-переменной класса *derived*. Однако включение этого объявления доступа

```
base::j;
```

в классе *derived* под спецификатором *public* восстанавливает *public*-статус члена *j*.

Объявление доступа можно использовать для восстановления прав доступа *public*- и *protected*-членов. Однако для изменения (повышения или понижения) статуса доступа его использовать нельзя. Например, член, объявленный закрытым в базовом классе, нельзя сделать открытым в производном. (Разрешение подобных вещей разрушило бы инкапсуляцию!)

Использование объявления доступа иллюстрируется в следующей программе.

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

class base {
   int i; // private-член в классе base
  public:
   int j, k;
   void seti (int x) { i = x; }
   int geti() { return i; }
};

// Класс base наследуется как private-класс.
class derived: private base {
  public:
    /* Следующие три инструкции переопределяют private-наследование класса base и восстанавливают public-статус доступа для членов j, seti() и geti(). */
     base::j; // Переменная j становится снова public-членом, а переменная k остается закрытым членом.
   base::seti; // Функция seti() становится public-членом.
   base::geti; // Функция geti() становится public-членом.
   // base::i; // Неверно: нельзя повышать уровень доступа.
   int a; // public-член
};

int main()
{
  derived ob;
  //ob.i = 10; // Неверно, поскольку член i  закрыт в классе derived.
  ob.j = 20; // Допустимо, поскольку член j стал  открытым в классе derived.
```

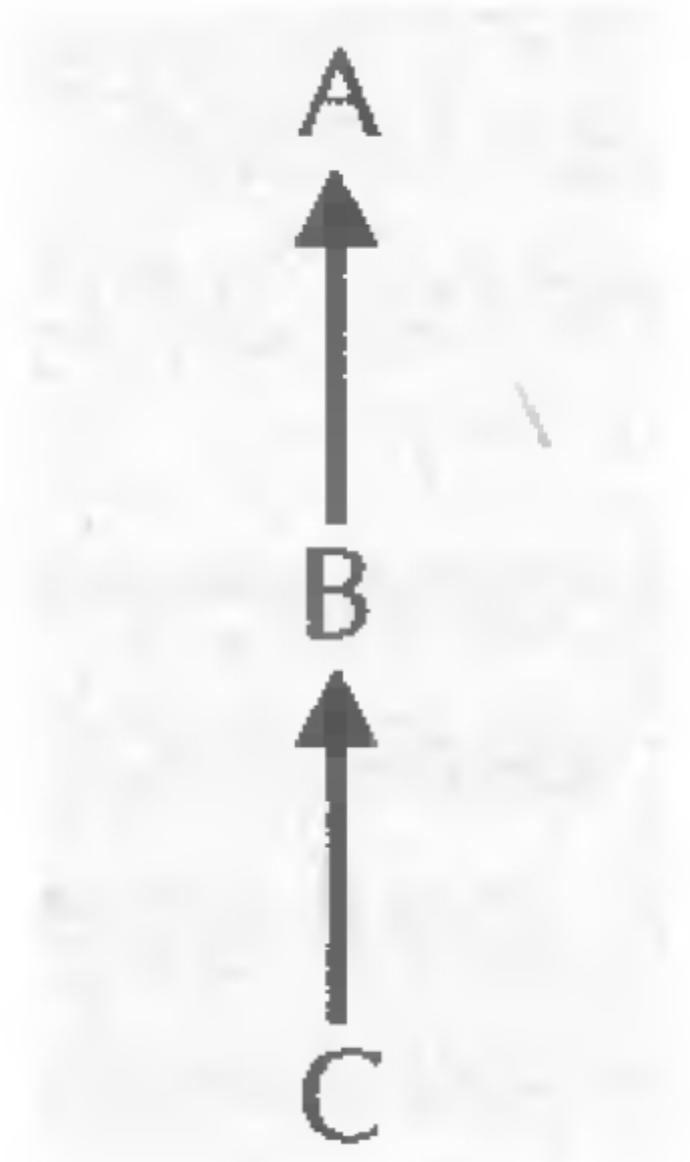
```
  //ob.k =30; // Неверно, поскольку член k  закрыт в классе derived.
  ob.a = 40; // Допустимо, поскольку член a  открыт в классе derived.
  ob.seti(10);
  cout << ob.geti() << " " << ob.j << " " << ob.a;
  return 0;
}
```

Обратите внимание на то, как в этой программе используются объявления доступа для восстановления статуса *public* у членов *j*, *seti()* и *geti()*. В комментариях отмечены и другие ограничения, связанные со статусом доступа.

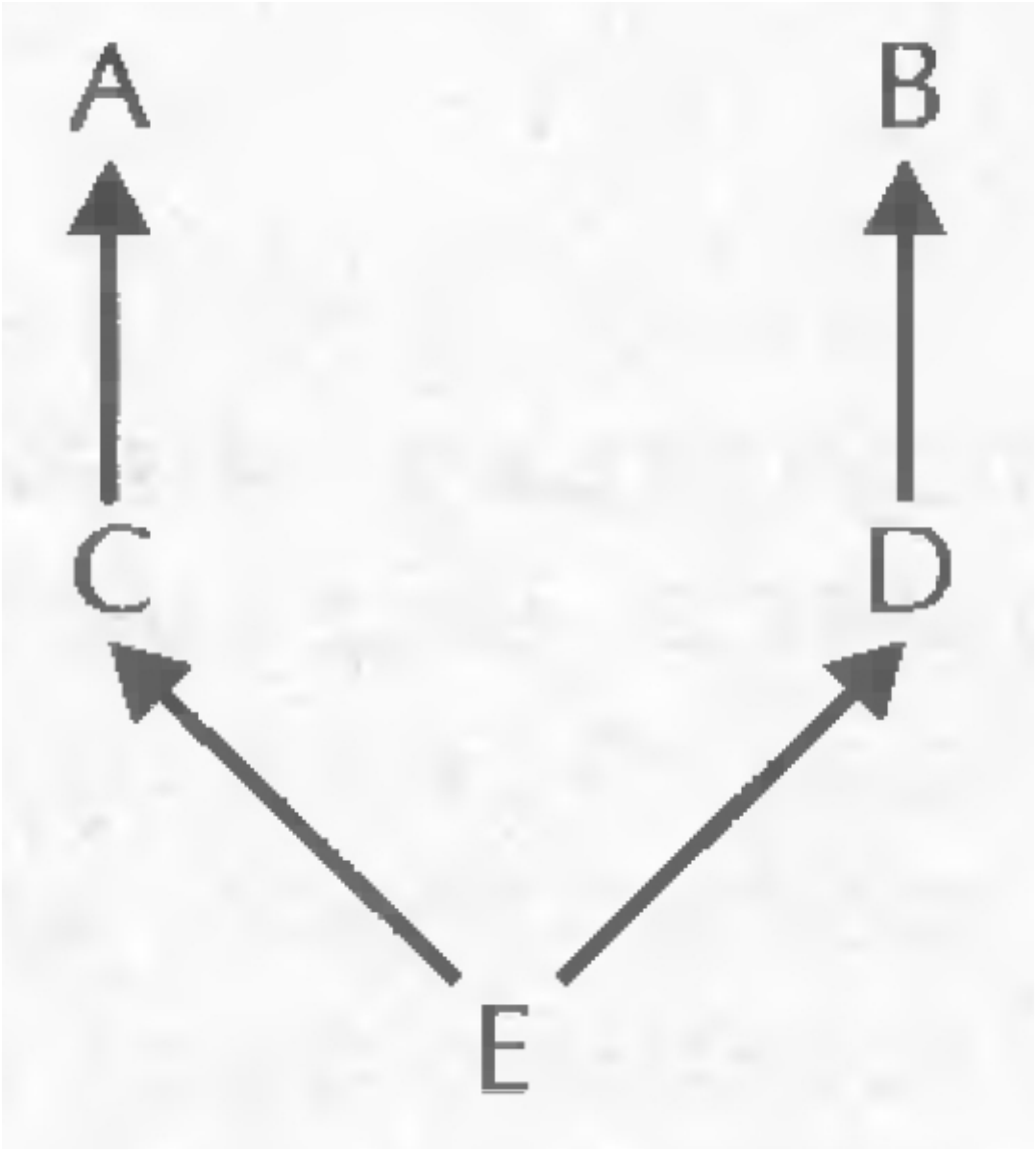
C++ обеспечивает возможность восстановления уровня доступа для унаследованных членов, чтобы программист мог успешно программировать такие специальные ситуации, когда большая часть наследуемого класса должна стать закрытой, а прежний *public*-или *protected*-статус нужно вернуть лишь нескольким членам. И все же к этому средству лучше прибегать только в крайних случаях.

Чтение C++-графов наследования

Иногда иерархии C++-классов изображаются графически, что облегчает их понимание. Но порой различные способы изображения графов наследования классов вводят новичков в заблуждение. Рассмотрим, например, ситуацию, в которой класс *A* наследуется классом *B*, который в свою очередь наследуется классом *C*. Используя стандартную C++-систему обозначений, эту ситуацию можно отобразить так:



Как видите, стрелки на этом рисунке направлены вверх, а не вниз. Многие поначалу считают такое направление стрелок алогичным, но именно этот стиль принят большинством C++-программистов. Согласно стилевой графике C++ стрелка должна указывать на базовый класс. Следовательно, стрелка означает *"выведен из"*, а не *"порождает"*. Рассмотрим другой пример. Можете ли вы описать словами значение следующего изображения?



Из этого графа следует, что класс *E* выведен из обоих классов *C* и *D*. (Другими словами, класс *E* имеет два базовых класса *C* и *D*.) При этом класс *C* выведен из класса *A*, а класс *D* — из класса *B*. Несмотря на то что направление стрелок может вас обескураживать на первых порах, все же лучше познакомиться с этим стилем графических обозначений, поскольку он широко используется в книгах, журналах и документации на компиляторы.

Виртуальные базовые классы

При наследовании нескольких базовых классов в C++-программу может быть внесен элемент неопределенности. Рассмотрим эту некорректную программу.

```
/* Эта программа содержит ошибку и не скомпилируется.
*/
#include <iostream>
using namespace std;

class base {
  public:
```

```
  public:
    int i;
};

// Класс derived1 наследует класс base.
class derived1 : public base { public: int j;};

// Класс derived2 наследует класс base.
class derived2 : public base { public: int k;};

/* Класс derived3 наследует оба класса derived1 и derived2. Это означает, что в классе derived3 существует две копии класса base!
*/
class derived3 : public derived1, public derived2 {
  public:
    int sum;
};

int main( )
{
  derived3 ob;

  ob.i = 10; // Это и есть неоднозначность: какой именно член i имеется в виду???
  ob.j = 20;
  ob.k = 30;

  //И здесь тоже неоднозначность с членом i.
  ob.sum = ob.i + ob.j + ob.k;

  // И здесь тоже неоднозначность с членом i.
  cout << ob.i << " ";
  cout << ob. j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Как отмечено в комментариях этой программы, оба класса *derived1* и *derived2* наследуют класс *base*. Но класс *derived3* наследует как класс *derived1*, так и класс *derived2*. В результате в объекте типа *derived3* присутствуют две копии класса *base*, поэтому, например, в таком выражении

```
ob.i = 20;
```

не ясно, на какую именно копию члена *i* здесь дана ссылка: на член, унаследованный от класса *derived1* или от класса *derived2*? Поскольку в объекте *ob* присутствуют обе копии класса *base*, то в нем существуют и два члена *ob.is*! Потому-то эта инструкция и является наследственно неоднозначной (существенно неопределенной).

Есть два способа исправить предыдущую программу. Первый состоит в применении оператора разрешения контекста (разрешения области видимости), с помощью которого можно "вручную" указать нужный член *i*. Например, следующая версия этой программы

можно "вручную" указать нужный член i. Например, следующая версия этой программы успешно скомпилируется и выполнится ожидаемым образом.

```
/* Эта программа использует оператор разрешения контекста для выбора нужного члена i.
*/
#include <iostream>
using namespace std;

class base {
  public:
    int i;
};

// Класс derived1 наследует класс base.
class derived1 : public base { public: int j;};

// Класс derived2 наследует класс base.
class derived2 : public base { public: int k;};

/* Класс derived3 наследует оба класса derived1 и derived2. Это означает, что в классе derived3 существует две копии класса base!
*/
class derived3 : public derived1, public derived2 {
  public:
    int sum;
};

int main( )
{
  derived3 ob;

  ob.derived1::i = 10; // Контекст разрешен, используется член i класса derived1.
  ob.j = 20;
  ob.k = 30;

  // Контекст разрешен и здесь.
  ob.sum = ob.derived1::i + ob.j + ob.k;

  // Неоднозначность ликвидирована и здесь.
  cout << ob.derived1::i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

Виртуальное наследование базового класса гарантирует, что в любом производном классе будет присутствовать только одна его копия.

Применение оператора *"::"* позволяет программе "ручным способом" выбрать версию

класса *base* (унаследованную классом *derived1*). Но после такого решения возникает интересный вопрос: а что, если в действительности нужна только одна копия класса *base*? Можно ли каким-то образом предотвратить включение двух копий в класс *derived3*? Ответ, как, наверное, вы догадались, положителен. Это решение достигается с помощью *виртуальных базовых классов*.

Если два (или больше) класса выведены из общего базового класса, мы можем предотвратить включение нескольких его копий в объекте, выведенном из этих классов, что реализуется путем объявления базового класса при его наследовании виртуальным. Для этого достаточно предварить имя наследуемого базового класса ключевым словом *virtual*.

Для иллюстрации этого процесса приведем еще одну версию предыдущей программы. На этот раз класс *derived3* содержит только одну копию класса *base*.

```
// Эта программа использует виртуальные базовые классы.
#include <iostream>
using namespace std;

class base {
  public:
  int i;
};

// Класс derived1 наследует класс base как виртуальный.
class derived1 : virtual public base { public: int j;};

// Класс derived2 наследует класс base как виртуальный.
class derived2 : virtual public base { public: int k;};

/* Класс derived3 наследует оба класса derived1 и derived2. На этот раз
он содержит только одну копию класса base.
*/
class derived3 : public derived1, public derived2 {
  public:
    int sum;
};

int main( )
{
  derived3 ob;
  ob.i = 10; // Теперь неоднозначности нет.
  ob.j = 20;
  ob.k = 30;

  // Теперь неоднозначности нет.
  ob.sum = ob.i + ob.j + ob.k;

  // Теперь неоднозначности, нет.
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
```

```
  return 0;
}
```

Как видите, ключевое слово *virtual* предваряет остальную часть спецификации наследуемого класса. Теперь оба класса *derived1* и *derived2* наследуют класс *base* как виртуальный, и поэтому при любом множественном их наследовании в производный класс будет включена только одна его копия. Следовательно, в классе *derived3* присутствует лишь одна копия класса *base*, а инструкция *ob.i = 10* теперь совершенно допустима и не содержит никакой неоднозначности.

И еще. Даже если оба класса *derived1* и *derived2* задают класс *base* как *virtual*-класс, он по-прежнему присутствует в объекте любого типа. Например, следующая последовательность инструкций вполне допустима.

```
// Определяем класс типа derived1.
derived1 myclass;
myclass.i = 88;
```

Разница между обычным базовым и виртуальным классами становится очевидной только тогда, когда этот базовый класс наследуется более одного раза. Если базовый класс объявляется виртуальным, то только один его экземпляр будет включен в объект наследующего класса. В противном случае в этом объекте будет присутствовать несколько его копий.

Глава 15: Виртуальные функции и полиморфизм

Одной из трех основных граней объектно-ориентированного программирования является полиморфизм. Применительно к C++ *полиморфизм* представляет собой термин, используемый для описания процесса, в котором различные реализации функции могут быть доступны посредством одного и того же имени. По этой причине полиморфизм иногда характеризуется фразой *"один интерфейс, много методов"*. Это означает, что ко всем функциям-членам общего класса можно получить доступ одним и тем же способом, несмотря на возможное различие в конкретных действиях, связанных с каждой отдельной операцией.

В C++ полиморфизм поддерживается как во время выполнения, так в период компиляции программы. Перегрузка операторов и функций — это примеры полиморфизма, относящегося ко времени компиляции. Но, несмотря на могущество механизма перегрузки операторов и функций, он не в состоянии решить все задачи, которые возникают в реальных приложениях объектно-ориентированного языка программирования. Поэтому в C++ также реализован полиморфизм периода выполнения на основе использования производных классов и *виртуальных функций*, которые и составляют основные темы этой главы.

Начнем же мы эту главу с краткого описания указателей на производные типы, поскольку именно они обеспечивают поддержку динамического полиморфизма.

Указатели на производные типы

Указатель на базовый класс может ссылаться на любой объект, выведенный из этого базового класса.

Фундаментом для динамического полиморфизма служит указатель на базовый класс. Указатели на базовые и производные классы связаны такими отношениями, которые не свойственны указателям других типов. Как было отмечено выше в этой книге, указатель одного типа, как правило, не может указывать на объект другого типа. Однако указатели на базовые классы и объекты производных классов — исключения из этого правила. В C++ указатель на базовый класс также можно использовать для ссылки на объект любого класса, выведенного из базового. Например, предположим, что у нас есть базовый класс *B_class* и класс *D_class*, который выведен из класса *B_class*. В C++ любой указатель, объявленный как указатель на класс *B_class*, может быть также указателем на класс *D_class*. Следовательно, после этих объявлений

```
B_class *p; // указатель на объект типа B_class
```

```
B_class B_ob; // объект типа B_class
D_class D_ob; // объект типа D_class
```

обе следующие инструкции абсолютно законны:

```
p = &B_ob; // p указывает на объект типа B_class
p = &D_ob; /* p указывает на объект типа D_class, который является объектом, выведенным из класса B_class. */
```

В этом примере указатель *p* можно использовать для доступа ко всем элементам объекта *D_ob*, выведенным из объекта *B_ob*. Однако к элементам, которые составляют специфическую "надстройку" (над базой, т.е. над базовым классом *B_class*) объекта *D_ob*, доступ с помощью указателя *p* получить нельзя.

В качестве более конкретного примера рассмотрим короткую программу, которая определяет базовый класс *B_class* и производный класс *D_class*. В этой программе простая иерархия классов используется для хранения имен авторов и названий их книг.

```
// Использование указателей на базовый класс для доступа к объектам производных классов.
#include <iostream>
#include <cstring>
using namespace std;

class B_class {
     char author[80];
   public:
     void put_author(char *s) { strcpy(author, s); }
     void show_author() { cout << author << "\n"; }
};

class D_class : public B_class {
   char title [80];
  public:
   void put_title(char *num) { strcpy(title, num);}
   void show_title() {
     cout << "Название: ";
     cout << title << "\n";
   }
};

int main()
{
  B_class *p;
  B_class B_ob;

  D_class *dp;
  D_class D_ob;

  p = &B_ob; // адрес объекта базового класса

  // Доступ к классу B_class через указатель.
```

```
  p->put_author("Эмиль Золя");

  // Доступ к классу D_class через "базовый" указатель.
  p = &D_ob;
  p->put_author("Уильям Шекспир");

  // Покажем, что каждый автор относится к  соответствующему объекту.
  B_ob.show_author();
  D_ob.show_author();
  cout << "\n";

   /* Поскольку функции put_title() и show_title() не являются частью базового класса, они недоступны через "базовый" указатель p, и поэтому к ним нужно обращаться либо непосредственно, либо, как показано здесь, через указатель на производный тип.
  */
  dp = &D_ob;
  dp->put_title("Буря");

   p->show_author(); // Здесь можно использовать либо указатель p, либо указатель dp.

  dp->show_title();

  return 0;
}
```

При выполнении эта программа отображает следующие результаты.

```
Эмиль Золя
Уильям Шекспир

Уильям Шекспир
Название: Буря
```

В этом примере указатель *p* определяется как указатель на класс *B_class*. Но он может также ссылаться на объект производного класса *D_class*, причем его можно использовать для доступа только к тем элементам производного класса, которые унаследованы от базового. Однако следует помнить, что через *"базовый"* указатель невозможно получить доступ к тем членам, которые специфичны для производного класса. Вот почему к функции *show_title()* обращение реализуется с помощью указателя *dp*, который является указателем на производный класс.

Если вам нужно с помощью указателя на базовый класс получить доступ к элементам, определенным производным классом, необходимо привести этот указатель к типу указателя на производный тип. Например, при выполнении этой строки кода действительно будет вызвана функция *show_title()* объекта *D_ob*:

```
((D_class *)p)->show_title();
```

Внешний набор круглых скобок используется для связи операции приведения типа с указателем *p*, а не с типом, возвращаемым функцией *show_title()*. Несмотря на то что в использовании такой операции формально нет ничего некорректного, этого по возможности следует избегать, поскольку подобные приемы попросту вносят в код программы путаницу. (На самом деле большинство C++-программистов считают такой стиль программирования неудачным.)

Кроме того, необходимо понимать, что хотя *"базовый"* указатель можно использовать для доступа к объектам любого производного типа, обратное утверждение неверно. Другими словами, используя указатель на производный класс, нельзя получить доступ к объекту базового типа.

Указатель инкрементируется и декрементируется относительно своего базового типа. Следовательно, если указатель на базовый класс используется для доступа к объекту производного типа, инкрементирование или декрементирование не заставит его ссылаться на следующий объект производного класса. Вместо этого он будет указывать на следующий объект базового класса. Таким образом, инкрементирование или декрементирование указателя на базовый класс следует расценивать как некорректную операцию, если этот указатель используется для ссылки на объект производного класса.

Тот факт, что указатель на базовый тип можно использовать для ссылки на любой объект, выведенный из базового, чрезвычайно важен и принципиален для C++. Как будет показано ниже, эта гибкость является ключевым моментом для способа реализации динамического полиморфизма в C++.

Ссылки на производные типы

Подобно указателям, ссылку на базовый класс также можно использовать для доступа к объекту производного типа. Эта возможность особенно часто применяется при передаче аргументов функциям. Параметр, который имеет тип ссылки на базовый класс, может принимать объекты базового класса, а также объекты любого другого типа, выведенного из него.

Виртуальные функции

Динамический полиморфизм возможен благодаря сочетанию двух средств: *наследования* и *виртуальных функций*. О механизме наследования вы узнали в предыдущей главе. Здесь же вы познакомитесь с виртуальными функциями.

Виртуальная функция — это функция, которая объявляется в базовом классе с использованием ключевого слова *virtual* и переопределяется в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственную версию виртуальной функции. Интересно рассмотреть ситуацию, когда виртуальная функция вызывается через указатель (или ссылку) на базовый класс. В этом случае C++ определяет, какую именно версию виртуальной функции необходимо вызвать, по *типу* объекта, адресуемого этим указателем. Причем следует иметь в виду, что это решение принимается во *время выполнения* программы. Следовательно, при указании на различные объекты будут вызываться и различные версии виртуальной функции. Другими словами, именно по типу адресуемого объекта (а не по типу самого указателя) определяется, какая версия виртуальной функции будет выполнена. Таким образом, если базовый класс содержит виртуальную функцию и если из этого базового класса выведено два (или больше) других класса, то при адресации различных типов объектов через указатель на базовый класс будут выполняться и различные версии виртуальной функции. Аналогичный механизм работает и при использовании ссылки на базовый класс.

Чтобы объявить функцию виртуальной, достаточно предварить ее объявление ключевым

словом virtual.

Функция объявляется виртуальной в базовом классе с помощью ключевого слова *virtual*. При переопределении виртуальной функции в производном классе ключевое слово *virtual* повторять не нужно (хотя это не будет ошибкой).

Класс, который включает виртуальную функцию, называется полиморфным классом.

Класс, который включает виртуальную функцию, называется полиморфным классом. Этот термин также применяется к классу, который наследует базовый класс, содержащий виртуальную функцию.

Рассмотрим следующую короткую программу, в которой демонстрируется использование виртуальных функций.

```
// Пример использования виртуальных функций.
#include <iostream>
using namespace std;

class base {
  public:
    virtual void who( ) {
      // объявление виртуальной функции
      cout << "Базовый класс.\n";
    }
};

class first_d : public base {
  public:
    void who( ) {
      // Переопределение функции who( ) для
      // класса first_d.
      cout << "Первый производный класс.\n";
    }
};

class second_d : public base {
  public:
    void who( ) {
      // Переопределение функции who( ) для
      // класса second_d.
      cout << "Второй производный класс.\n";
    }
};

int main( )
{
  base base_obj;
  base *p;
  first_d first_obj;
  second_d second_obj;
```

```
  second_d second_obj;

  p = &base_obj;
  p->who(); // доступ к функции who() класса base

  p = &first_obj;
  p->who(); // доступ к функции who() класса first_d

  p = &second_obj;
  p->who(); // доступ к функции who() класса second_d

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Базовый класс.
Первый производный класс.
Второй производный класс.
```

Теперь рассмотрим код этой программы подробно, чтобы понять, как она работает.

В классе *base* функция *who()* объявлена виртуальной. Это означает, что ее можно переопределить в производном классе (в классе, выведенном из *base*). И она действительно переопределяется в обоих производных классах *first_d* и *second_d*. В функции main() объявляются четыре переменные: *base_obj* (объект типа base), *p* (указатель на объект класса base), а также два объекта *first_obj* и *second_obj* двух производных классов *first_d* и *second_d* соответственно. Затем указателю *p* присваивается адрес объекта *base_obj*, и вызывается функция *who()*. Поскольку функция *who()* объявлена виртуальной, C++ во время выполнения программы определяет, к какой именно версии функции *who()* здесь нужно обратиться, причем решение принимается путем анализа типа объекта, адресуемого указателем *p*. В данном случае *p* указывает на объект типа *base*, поэтому сначала выполняется та версия функции *who()*, которая объявлена в классе *base*. Затем указателю *p* присваивается адрес объекта *first_obj*. Вспомните, что с помощью указателя на базовый класс можно обращаться к объекту любого его производного класса. Поэтому, когда функция *who()* вызывается во второй раз, C++ снова выясняет тип объекта, адресуемого указателем *p*, и, исходя из этого типа, определяет, какую версию функции *who()* нужно вызвать. Поскольку *p* здесь указывает на объект типа *first_d*, то выполняется версия функции *who()*, определенная в классе *first_d*. Аналогично после присвоения *p* адреса объекта *second_obj* вызывается версия функции *who()*, объявленная в классе *second_d*.

Узелок на память. *То, какая версия виртуальной функции действительно будет вызвана, определяется во время выполнения программы. Решение основывается исключительно на типе объекта, адресуемого указателем на базовый класс.*

Виртуальную функцию можно вызывать обычным способом (не через указатель), используя оператор *"точка"* и задавая имя вызывающего объекта. Это означает, что в предыдущем примере было бы синтаксически корректно обратиться к функции *who()* с помощью следующей инструкции:

```
first_obj.who();
```

Однако при вызове виртуальной функции таким способом игнорируются ее полиморфные атрибуты. И только при обращении к виртуальной функции через указатель на базовый класс достигается динамический полиморфизм.

Если виртуальная функция переопределяется в производном классе, ее называют переопределенной.

Поначалу может показаться, что переопределение виртуальной функции в производном классе представляет собой специальную форму перегрузки функций. Но это не так. В

классе представляет собой специальную форму перегрузки функций. Но это не так. В действительности мы имеем дело с двумя принципиально разными процессами. Прежде всего, версии перегруженной функции должны отличаться друг от друга типом и/или количеством параметров, в то время как тип и количество параметров у версий переопределенной функции должны в точности совпадать. И в самом деле, прототипы виртуальной функции и ее переопределений должны быть абсолютно одинаковыми. Если прототипы будут различными, то такая функция будет попросту считаться перегруженной, и ее *"виртуальная сущность"* утратится. Кроме того, виртуальная функция должна быть членом класса, для которого она определяется, а не его *"другом"*. Но в то же время виртуальная функция может быть *"другом"* другого класса. И еще: функциям деструкторов разрешается быть виртуальными, а функциям конструкторов — нет.

Наследование виртуальных функций

Атрибут virtual передается "по наследству".

Если функция объявляется как виртуальная, она остается такой независимо от того, через сколько уровней производных классов она может пройти. Например, если бы класс *second_d* был выведен из класса *first_d*, а не из класса *base*, как показано в следующем примере, то функция *who()* по-прежнему оставалась бы виртуальной, и механизм выбора соответствующей версии по-прежнему работал бы корректно.

```
// Этот класс выведен из класса first_d, а не из base.
class second_d : public first_d {
  public:
    void who( ) {
      // Переопределение функции who( ) для класса second_d.
      cout << "Второй производный класс.\n";
    }
};
```

Если производный класс не переопределяет виртуальную функцию, то используется функция, определенная в базовом классе. Например, проверим, как поведет себя версия предыдущей программы, если в классе *second_d* не будет переопределена функция *who()*.

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

class base {
  public:
    virtual void who( ) {
      cout << "Базовый класс.\n";
    }
};

class first_d : public base {
  public:
    void who( ) {
      cout << "Первый производный класс.\n";
    }
};

class second_d : public base {
  // Функция who( ) здесь не определена вообще.
```

```
};

int main( )
{
  base base_obj;
  base *p;
  first_d first_obj;
  second_d second_obj;

  p = &base_obj;
  p->who( ); // доступ к функции who( ) класса base

  p = &first_obj;
  p->who( ); // доступ к функции who( ) класса first_d

  p = &second_obj;
  p->who( ); /* Здесь выполняется обращение к функции who( ) класса base,
поскольку в классе second_d она не переопределена. */

  return 0;
}
```

Теперь при выполнении этой программы на экран выводится следующее.

```
Базовый класс.
Первый производный класс.
Базовый класс.
```

Как подтверждают результаты выполнения этой программы, поскольку функция *who()* не переопределена классом *second_d*, то при ее вызове с помощью инструкции *p->who()* (когда член *p* указывает на объект *second_obj*) выполняется та версия функции *who()*, которая определена в классе *base*.

Следует иметь в виду, что наследуемые свойства спецификатора *virtual* являются иерархическими. Поэтому, если предыдущий пример изменить так, чтобы класс *second_d* был выведен из класса *first_d*, а не из класса *base*, то при обращении к функции *who()* через объект типа *second_d* будет вызвана та ее версия, которая объявлена в классе *first_d*, поскольку этот класс является "ближайшим" (по иерархическим "меркам") к классу *second_d*, а не функция *who()* из тела класса *base*. Эти иерархические зависимости демонстрируются на примере следующей программы.

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

class base {
  public:
    virtual void who( ) {
      cout << "Базовый класс.\n";
    }
};

class first_d : public base {
  public:
```

```
  public:
    void who( ) {
      cout << "Первый производный класс.\n";
    }
};

// Класс second_d теперь выведен из класса first_d, а не из класса base.
class second_d : public first_d {
  // Функция who( ) не определена.
};

int main( )
{
  base base_obj;
  base *p;
  first_d first_obj;
  second_d second_obj;

  p = &base_obj;
  p->who( ); // доступ к функции who( ) класса base

  p = &first_obj;
  p->who( ); // доступ к функции who( ) класса first_d

  p = &second_obj;
   p->who( );  /* Здесь выполняется обращение к функции who( ) класса
first_d, поскольку в классе second_d она не переопределена. */

  return 0;
}
```

Эта программа генерирует такие результаты.

```
Базовый класс.
Первый производный класс.
Первый производный класс.
```

Как видите, класс *second_d* теперь использует версию функции *who()*, которая определена в классе *first_d*, поскольку она ближе всех в иерархической цепочке классов.

Зачем нужны виртуальные функции

Как отмечалось в начале этой главы, виртуальные функции в сочетании с производными типами позволяют C++ поддерживать динамический полиморфизм. Полиморфизм существенен для объектно-ориентированного программирования по одной важной причине: он обеспечивает возможность некоторому обобщенному классу определять функции, которые будут использовать все производные от него классы, причем производный класс может определить собственную реализацию некоторых или всех этих функций. Иногда эта идея выражается следующим образом: базовый класс диктует общий интерфейс, который будет иметь любой объект, выведенный из этого класса, но позволяет при этом производному классу определить метод, используемый для реализации этого интерфейса. Вот почему для описания полиморфизма часто используется фраза *"один интерфейс, множество методов"*.

Для успешного применения полиморфизма необходимо понимать, что базовый и производный классы образуют иерархию, развитие которой направлено от большей к

производный классы образуют иерархию, развитие которой направлено от большей к меньшей степени обобщения (т.е. от базового класса к производному). При корректной разработке базовый класс обеспечивает все элементы, которые производный класс может использовать напрямую. Он также определяет функции, которые производный класс должен реализовать самостоятельно. Это дает производному классу гибкость в определении собственных методов, но в то же время обязывает использовать общий интерфейс. Другими словами, поскольку формат интерфейса определяется базовым классом, любой производный класс должен разделять этот общий интерфейс. Таким образом, использование виртуальных функций позволяет базовому классу определять обобщенный интерфейс, который будет использован всеми производными классами.

Теперь у вас может возникнуть вопрос: почему же так важен общий интерфейс со множеством реализаций? Ответ снова возвращает нас к основной побудительной причине возникновения объектно-ориентированного программирования: такой интерфейс позволяет программисту справляться со все возрастающей сложностью программ. Например, если корректно разработать программу, то можно быть уверенным в том, что ко всем объектам, выведенным из базового класса, можно будет получить доступ единым (общим для всех) способом, несмотря на то, что конкретные действия у одного производного класса могут отличаться от действий у другого. Это означает, что программисту придется помнить только один интерфейс, а не великое их множество. Кроме того, производный класс волен использовать любые или все функции, предоставленные базовым классом. Другими словами, разработчику производного класса не нужно заново изобретать элементы, уже имеющиеся в базовом классе. Более того, отделение интерфейса от реализации позволяет создавать библиотеки классов, написанием которых могут заниматься сторонние организации. Корректно реализованные библиотеки должны предоставлять общий интерфейс, который программист может использовать для выведения классов в соответствии со своими конкретными потребностями. Например, как библиотека базовых классов Microsoft (Microsoft Foundation Classes — *MFC*), так и более новая библиотека классов *.NET Framework Windows Forms* поддерживают Windows-программирование. Использование этих классов позволяет писать программы, которые могут унаследовать множество функций, нужных любой Windows-программе. Вам понадобится лишь добавить в нее средства, уникальные для вашего приложения. Это — большое подспорье при программировании сложных систем.

Простое приложение виртуальных функций

Чтобы вы могли получить представление о силе принципа "один интерфейс, множество методов", рассмотрим следующую короткую программу. Она создает базовый класс *figure*, предназначенный для хранения размеров различных двумерных объектов и вычисления их площадей. Функция *set_dim()* является стандартной функцией-членом, поскольку эта операция подходит для всех производных классов. Однако функция *show_area()* объявлена как виртуальная, так как методы вычисления площади различных объектов будут разными. Программа использует базовый класс *figure* для выведения двух специальных классов *rectangle* и *triangle*.

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

class figure {
  protected:
    double x, y;
  public:
    void set_dim(double i, double j) {
      x = i;
      y = j;
    }
```

```
    virtual void show_area() {
      cout << "Для этого класса выражение вычисления ";
      cout << "площади не определено.\n";
    }
};

class triangle : public figure {
  public:
    void show_area() {
      cout << "Треугольник с высотой ";
      cout << x << " и основанием " << y;
      cout << " имеет площадь ";
      cout << x * 0.5 * y << ".\n";
    }
};

class rectangle : public figure {
  public:
    void show_area() {
      cout << "Прямоугольник с размерами ";
      cout << x << " x " << y;
      cout << " имеет площадь ";
      cout << x * y << ".\n";
    }
};

int main()
{
  figure *p; // создаем указатель на базовый тип

  triangle t; // создаем объекты производных типов
  rectangle r;

  p = &t;
  p->set_dim(10.0, 5.0);
  p->show_area();

  p = &r;
  p->set_dim(10.0, 5.0);
  p->show_area();

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Треугольник с высотой 10 и основанием 5 имеет площадь 25.
Прямоугольник с размерами 10 x 5 имеет площадь 50.
```

В этой программе обратите внимание на то, что при работе с классами *rectangle* и *triangle* используется одинаковый интерфейс, несмотря на то, что в них реализованы собственные методы вычисления площади соответствующих объектов.

Как вы думаете, используя объявление класса *figure*, можно вывести класс *circle* для вычисления площади круга по заданному значению радиуса? Ответ: да. Для этого достаточно создать новый производный тип, который бы вычислял площадь круга. Могущество виртуальных функций опирается на тот факт, что программист может легко вывести новый тип, который будет разделять общий интерфейс с другими *"родственными"* объектами. Вот, например, как это можно сделать в нашем случае:

```
class circle : public figure {
  public:
    void show_area( ) {
      cout << "Круг с радиусом ";
      cout << x;
      cout << " имеет площадь ";
      cout << 3.14 * x * x;
    }
};
```

Прежде чем опробовать класс *circle* в *"деле"*, рассмотрим внимательно определение функции *show_area()*. Обратите внимание на то, что в нем используется только одно значение переменной *x*, которая должна содержать радиус круга. (Вспомните, что площадь круга вычисляется по формуле nR^2.) Однако согласно определению функции *set_dim()* в классе *figure* ей передается два значения, а не одно. Поскольку классу *circle* не нужно второе значение, то что мы можем предпринять?

Есть два способа решить эту проблему. Первый (и одновременно наихудшим) состоит в том, что мы могли бы, работая с объектом класса *circle*, просто вызывать функцию *set_dim()*, передавая ей в качестве второго параметра фиктивное значение. Основной недостаток этого метода — отсутствие четкости в задании параметров и необходимость помнить о специальном исключении, которое нарушает действие принципа: *"один интерфейс, множество методов"*.

Есть более удачный способ решения этой проблемы, который заключается в предоставлении параметру функции *set_dim()* значения, действующего по умолчанию. В этом случае при вызове функции *set_dim()* для круга нужно задавать только радиус. При вызове же функции *set_dim()* для треугольника или прямоугольника задаются оба значения. Ниже показана программа, в которой реализован этот метод.

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

class figure {
  protected:
    double x, y;
  public:
    void set_dim(double i, double j=0) {
      x = i;
      y = j;
    }
```

```
    virtual void show_area( ) {
      cout << "Для этого класса выражение вычисления ";
      cout << "площади не определено.\n";
    }
};

class triangle : public figure {
  public:
    void show_area( ) {
      cout << "Треугольник с высотой ";
      cout << x << " и основанием " << y;
      cout << " имеет площадь ";
      cout << x * 0.5 * y << ".\n";
    }
};

class rectangle : public figure {
  public:
    void show_area( ) {
      cout << "Прямоугольник с размерами ";
      cout << x << " x " << y;
      cout << " имеет площадь ";
      cout << x * y << ".\n";
    }
};

class circle : public figure {
  public:
    void show_area( ) {
      cout << "Круг с радиусом ";
      cout << x;
      cout << " имеет площадь ";
      cout << 3.14 * x * x << ".\n";
    }
};

int main( )
{
  figure *p; // создаем указатель на базовый тип

  triangle t; // создаем объекты производных типов
  rectangle r;
  circle c;

  p = &t;
```

```
  p->set_dim(10.0, 5.0);
  p->show_area();

  p = &r;
  p->set_dim(10.0, 5.0);
  p->show_area();

  p = &c;
  p->set_dim(9.0);
  p->show_area();

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Треугольник с высотой 10 и основанием 5 имеет площадь 25.
Прямоугольник с размерами 10 x 5 имеет площадь 50.
Круг с радиусом 9 имеет площадь 254.34.
```

Важно! *Несмотря на то что виртуальные функции синтаксически просты для понимания, их настоящую силу невозможно продемонстрировать на коротких примерах. Как правило, могущество полиморфизма проявляется в больших сложных системах. По мере освоения C++ вам еще не раз представится случай убедиться в их полезности.*

Чисто виртуальные функции и абстрактные классы

Как вы могли убедиться, если виртуальная функция, которая не переопределена в производном классе, вызывается объектом этого производного класса, то используется версия, определенная в базовом классе. Но во многих случаях вообще нет смысла давать определение виртуальной функции в базовом классе. Например, в базовом классе *figure* (из предыдущего примера) определение функции *show_area()* — это просто заглушка. Она не вычисляет и не отображает площадь ни одного из объектов. Как вы увидите при создании собственных библиотек классов, в том, что виртуальная функция не имеет значащего определения в контексте базового класса, нет ничего необычного.

Чисто виртуальная функция — это виртуальная функция, которая не имеет определения в базовом классе.

Существует два способа обработки таких ситуаций. Первый (он показан в предыдущем примере программы) заключается в обеспечении функцией вывода предупреждающего сообщения. Возможно, такой подход и будет полезен в определенных ситуациях, но в большинстве случаев он попросту неприемлем. Например, можно представить себе виртуальные функции, без определения которых в существовании производного класса вообще нет никакого смысла. Рассмотрим класс *triangle*. Он абсолютно бесполезен, если в нем не определить функцию *show_area()*. В этом случае имеет смысл создать метод, который бы гарантировал, что производный класс действительно содержит все необходимые функции. В C++ для решения этой проблемы и предусмотрены чисто виртуальные функции.

Чисто виртуальная функция — это функция, объявленная в базовом классе, но не имеющая в нем никакого определения. Поэтому любой производный тип должен определить собственную версию этой функции, ведь у него просто нет никакой возможности использовать версию из базового класса (по причине ее отсутствия). Чтобы объявить чисто виртуальную функцию, используйте следующий общий формат:

```
virtual тип имя_функции(список_параметров) = 0;
```

Здесь под элементом *тип* подразумевается тип значения, возвращаемого функцией, а элемент *имя_функции*— ее имя. Обозначение *= 0* является признаком того, что функция

здесь объявляется как чисто виртуальная. Например, в следующей версии определения класса *figure* функция *show_area()* уже представлена как чисто виртуальная.

```
class figure {
   double x, y;
  public:
   void set_dim(double i, double j =0) {
     x = i;
     y = j;
   }
   virtual void show_area( ) =0; // чисто виртуальная функция
};
```

Объявив функцию чисто виртуальной, программист создает условия, при которых производный класс просто вынужден иметь определение собственной ее реализации. Без этого компилятор выдаст сообщение об ошибке. Например, попытайтесь скомпилировать эту модифицированную версию программы вычисления площадей геометрических фигур, в которой из класса *circle* удалено определение функции *show_area()*.

```
/* Эта программа не скомпилируется, поскольку в классе circle нет
переопределения функции show_area( ).
*/
#include <iostream>
using namespace std;

class figure {
  protected:
   double x, y;
  public:
   void set_dim(double i, double j) {
     x = i;
     y = j;
   }
   virtual void show_area( ) = 0; // чисто виртуальная функция
};

class triangle : public figure {
  public:
   void show_area( ) {
     cout << "Треугольник с высотой ";
     cout << x << " и основанием " << y;
     cout << " имеет площадь ";
     cout << x * 0.5 * y << ".\n";
   }
};

class rectangle : public figure {
```

```
  public:
    void show_area( ) {
      cout << "Прямоугольник с размерами ";
      cout << x << "x" << y;
      cout << " имеет площадь ";
      cout << x * y << ".\n";
    }
};

class circle : public figure {
  // Отсутствие определения функции show_area( )
  // вызовет сообщение об ошибке.
};

int main( )
{
  figure *p; // создаем указатель на базовый тип

  triangle t; // создаем объекты производных классов
  rectangle r;
  circle c; // Ошибка: создание этого объекта невозможно!

  p = & t;
  p->set_dim(10.0, 5.0);
  p->show_area( );

  p = & r;
  p->set_dim(10.0, 5.0);
  p->show_area( );

  return 0;
}
```

Класс, который содержит хотя бы одну чисто виртуальную функцию, называется абстрактным.

Если класс имеет хотя бы одну чисто виртуальную функцию, его называют *абстрактным*. Абстрактный класс характеризуется одной важной особенностью: у такого класса не может быть объектов. Абстрактный класс можно использовать только в качестве базового, из которого будут выводиться другие классы. Причина того, что абстрактный класс нельзя использовать для создания объектов, лежит, безусловно, в том, что его одна или несколько функций не имеют определения. Но даже если базовый класс является абстрактным, его все равно можно использовать для объявления указателей и ссылок, которые необходимы для поддержки динамического полиморфизма.

Сравнение раннего связывания с поздним

При обсуждении объектно-ориентированных языков обычно используются два термина: *раннее связывание* (early binding) и *позднее связывание* (late binding). В C++ эти термины связывают с событиями, которые происходят во время компиляции и в период выполнения программы соответственно.

При раннем связывании вызов функции подготавливается во время компиляции, а при

позднем — во время выполнения программы.

Раннее связывание означает, что вся информация, необходимая для вызова функции, известна при компиляции программы. Примерами раннего связывания могут служить вызовы стандартных функций и вызовы перегруженных функций (обычных и операторных). Из принципиальных достоинств раннего связывания можно назвать эффективность: оно работает быстрее позднего и часто требует меньших затрат памяти. Его основной недостаток — отсутствие гибкости.

Позднее связывание означает, что точное решение о вызове функции будет принято во время выполнения программы. Позднее связывание в C++ достигается за счет использования виртуальных функций и производных типов. Преимущество позднего связывания состоит в том, что оно обеспечивает большую степень гибкости. Его можно применять для поддержки общего интерфейса и разрешать при этом различным объектам, которые используют этот интерфейс, определять их собственные реализации. Более того, позднее связывание может помочь программисту в создании библиотек классов, характеризующихся многократным использованием и возможностью расширяться. Но к его недостаткам можно отнести, хотя и незначительное, но все же понижение скорости выполнения программ.

Чему отдать предпочтение — раннему или позднему связыванию, зависит от назначения вашей программы. (В действительности в большинстве крупных программ используются оба вида связывания.) Позднее связывание (его еще называют *динамическим*) — это одно из самых мощных средств C++. Однако за это могущество приходится расплачиваться потерями в скорости выполнения программ. Поэтому позднее связывание лучше всего использовать только в случае, когда оно существенно улучшает структуру и управляемость программой. Как и все сильные средства, позднее связывание, конечно, стоит использовать, но не злоупотребляя им. Вызванные им потери в производительности весьма незначительны, поэтому, когда ситуация требует позднего связывания, смело берите его на вооружение.

Полиморфизм и пуризм

На протяжении всей книги (и в частности, в этой главе) мы отмечаем различия между динамическим и статическим полиморфизмом. Статический полиморфизм (полиморфизм времени компиляции) реализуется в перегрузке функций и операторов. Динамический (полиморфизм времени выполнения программы) достигается за счет виртуальных функций. Самое общее определение полиморфизма заключено во фразе *"один интерфейс, множество методов"*, и все упомянутые выше *"орудия"* полиморфизма отвечают этому определению. Однако при использовании самого термина *полиморфизм* все же существуют некоторые разногласия.

Некоторые пуристы (в данном случае — борцы за чистоту терминологии объектно-ориентированного программирования) настаивают, чтобы этот термин использовался только для событий, которые происходят во время выполнения программ. Они утверждают, что полиморфизм поддерживается только виртуальными функциями. Частично эта точка зрения основывается на том факте, что самыми первыми полиморфическими языками программирования были интерпретаторы (для них характерно то, что все события относятся ко времени выполнения программы). Появление транслируемых полиморфических языков программирования расширило концепцию полиморфизма. Однако все еще не утихают заявления о том, что термин *полиморфизм* должен применяться исключительно к событиям периода выполнения. Большинство C++-программистов не согласны с этой точкой зрения и считают, что этот термин применим к обоим видам средств. Поэтому вы не должны удивляться, если кто-то в один прекрасный день станет спорить с вами на предмет использования этого термина!

Глава 16: Шаблоны

Шаблон — это одно из самых сложных и мощных средств в C++. Он не вошел в исходную спецификацию C++, и лишь несколько лет назад стал неотъемлемой частью программирования на C++. Шаблоны позволяют достичь одну из самых трудных целей в

программировании — создать многократно используемый код.

Используя шаблоны, можно создавать обобщенные функции и классы. В обобщенной функции (или классе) обрабатываемый ею (им) тип данных задается как параметр. Таким образом, одну функцию или класс можно использовать для разных типов данных, не предоставляя явным образом конкретные версии для каждого типа данных. Рассмотрению обобщенных функций и обобщенных классов посвящена данная глава.

Обобщенные функции

Обобщенная функция — *это функция, перегружающая сама себя*.

Обобщенная функция определяет общий набор операций, которые предназначены для применения к данным различных типов. Тип данных, обрабатываемых функцией, передается ей как параметр. Используя обобщенную функцию, к широкому диапазону данных можно применить единую общую процедуру. Возможно, вам известно, что многие алгоритмы имеют одинаковую логику для разных типов данных. Например, один и тот же алгоритм сортировки *Quicksort* применяется и к массиву целых чисел, и к массиву значений с плавающей точкой. Различие здесь состоит только в типе сортируемых данных. Создавая обобщенную функцию, можно определить природу алгоритма независимо от типа данных. После этого компилятор автоматически сгенерирует корректный код для типа данных, который в действительности используется при выполнении этой функции. По сути, создавая обобщенную функцию, вы создаете функцию, которая автоматически перегружает себя саму.

Обобщенная функция создается с помощью ключевого слова *template*. Обычное значение слова *"template"* точно отражает цель его применения в C++. Это ключевое слово используется для создания шаблона (или оболочки), который описывает действия, выполняемые функцией. Компилятору же остается *"дополнить недостающие детали"* в соответствии с заданным значением параметра. Общий формат определения шаблонной функции имеет следующий вид.

```
template <class Ttype> тип имя_функции (список_параметров)
{
  // тело функции
}
```

Определение обобщенной функции начинается с ключевого слова template.

Здесь элемент *Ttype* представляет собой *"заполнитель"* для типа данных, обрабатываемых функцией. Это имя может быть использовано в теле функции. Но оно означает всего лишь заполнитель, вместо которого компилятор автоматически подставит реальный тип данных при создании конкретной версии функции. И хотя для задания обобщенного типа в *template*-объявлении по традиции применяется ключевое слово *class*, можно также использовать ключевое слово *typename*.

В следующем примере создается обобщенная функция, которая меняет местами значения двух переменных, используемых при ее вызове. Поскольку общий процесс обмена значениями переменных не зависит от их типа, он является прекрасным кандидатом для создания обобщенной функции.

```
// Пример шаблонной функции.
#include <iostream>
using namespace std;

// Определение шаблонной функции.
template <class X> void swapargs(X &a, X &b)
{
  X temp;
```

```
  temp = a;
  a = b;
  b = temp;
}

int main( )
{
  int i = 10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Исходные значения i, j: " << i << ' '<< j << ' \ n ';
  cout << "Исходные значения x, y: " << x << ' '<< y << '\n';
  cout << "Исходные значения a, b: " << a << ' '<< b << ' \n';

  swapargs(i, j); // перестановка целых чисел
  swapargs(x, y); // перестановка значений с плавающей точкой
  swapargs(a, b); // перестановка символов

  cout << "После перестановки i, j: " << i << ' '<< j << ' \ n ';
  cout << "После перестановки x, y: " << x << ' '<< y << '\n';
  cout << "После перестановки a, b: " << a << ' '<< b << ' \n';

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Исходные значения i, j: 10 20
Исходные значения x, y: 10.1 23.3
Исходные значения a, b: x z
После перестановки i, j: 20 10
После перестановки x, y: 23.3 10.1
После перестановки a, b: z x
```

Итак, рассмотрим внимательно код программы. Строка

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору, во-первых, что создается шаблон, и, во-вторых, что здесь начинается обобщенное определение. Обозначение *X* представляет собой обобщенный тип, который используется в качестве *"заполнителя"*. За *template*-заголовком следует объявление функции *swapargs()*, в котором символ *X* означает тип данных для значений, которые будут меняться местами. В функции *main()* демонстрируется вызов функции *swapargs()* с использованием трех различных типов данных: *int*, *float* и *char*. Поскольку функция *swapargs()* является обобщенной, компилятор автоматически создает три версии функции *swapargs()*: одну для обмена целых чисел, вторую для обмена значений с плавающей точкой и третью для обмена символов.

Здесь необходимо уточнить некоторые важные термины, связанные с шаблонами. Во-первых, обобщенная функция (т.е. функция, объявление которой предваряется *template*-инструкцией) также называется *шаблонной функцией*. Оба термина используются в этой

книге взаимозаменяемо. Когда компилятор создает конкретную версию этой функции, то говорят, что создается ее *специализация* (или *конкретизация*). Специализация также называется *порожденной функцией* (generated function). Действие порождения функции определяют как ее *реализацию* (instantiating). Другими словами, порождаемая функция является конкретным экземпляром шаблонной функции.

Поскольку C++ не распознает символ конца строки в качестве признака конца инструкции, *template*-часть определения обобщенной функции может не находиться в одной строке с именем этой функции. В следующем примере показан еще один (довольно распространенный) способ форматирования функции *swapargs()*.

```
template <class X>
void swapargs(X &a, X &b)
{
  X temp;
  temp = a;
  a = b;
  b = temp;
}
```

При использовании этого формата важно понимать, что между *template*-инструкцией и началом определения обобщенной функции никакие другие инструкции находиться не могут. Например, следующий фрагмент кода не скомпилируется.

```
// Этот код не скомпилируется.
template <class X>
int i; // Здесь ошибка!
void swapargs(X &a, X &b)
{
  X temp;
  temp = a;
  a = b;
  b = temp;
}
```

Как отмечено в комментарии, *template*-спецификация должна стоять непосредственно перед определением функции. Между ними не может находиться ни инструкция объявления переменной, ни какая-либо другая инструкция.

Функция с двумя обобщенными типами

В *template*-инструкции можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятой. Например, в следующей программе создается шаблонная функция с двумя обобщенными типами.

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
  cout << x << ' ' << y << '\n';
}
```

```
int main()
{
  myfunc(10, "Привет");
  myfunc(0.23, 10L);
  return 0;
}
```

В этом примере при выполнении функции main(), когда компилятор генерирует конкретные экземпляры функции *myfunc()*, заполнители типов *type1* и *type2* заменяются сначала парой типов данных *int* и *char**, а затем парой *double* и *long* соответственно.

Узелок на память. *Создавая шаблонную функцию, вы, по сути, разрешаете компилятору генерировать столько различных версий этой функции, сколько необходимо для обработки различных способов, которые использует программа для ее вызова.*

Явно заданная перегрузка обобщенной функции

"Вручную" перегруженная версия обобщенной функции называется явной специализацией.

Несмотря на то что обобщенная функция сама перегружается по мере необходимости, это можно делать и явным образом. Формально этот процесс называется *явной специализацией*. При перегрузке обобщенная функция переопределяется *"в пользу"* этой конкретной версии. Рассмотрим, например, следующую программу, которая представляет собой переработанную версию первого примера из этой главы.

```
// Перегрузка шаблонной функции.
#include <iostream>
using namespace std;

template <class X>
void swapargs(X &a, X &b)
{
  X temp;
  temp = a;
  a = b;
  b = temp;
  cout << "Выполняется шаблонная функция swapargs.\n";
}

// Эта функция переопределяет обобщенную версию функции swapargs() для int-параметров.
void swapargs(int &a, int &b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
  cout << "Это int-специализация функции swapargs.\n";
}

int main()
```

```
int main()
{
  int i=10, j =20;
  double x=10.1, y=23.3;
  char a='x', b='z';

  cout << "Исходные значения i, j: " << i << ' '<< j << '\n';
  cout << "Исходные значения x, y: " << x << ' '<< y << '\n';
  cout << "Исходные значения a, b: " << a << ' '<< b << '\n';

  swapargs(i, j); // Вызывается явно перегруженная функция swapargs().
  swapargs(x, y); // Вызывается обобщенная функция swapargs().
  swapargs(a, b); // Вызывается обобщенная функция swapargs().

  cout << "После перестановки i, j: " << i << ' '<< j << '\n';
  cout << "После перестановки x, y: " << x << ' '<< y << '\n';
  cout << "После перестановки a, b: " << a << ' '<< b << '\n';

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Исходные значения i, j: 10 20
Исходные значения x, y: 10.1 23.3
Исходные значения a, b: x z
Это int-специализация функции swapargs.
Выполняется шаблонная функция swapargs.
Выполняется шаблонная функция swapargs.
После перестановки i, j: 20 10
После перестановки x, y: 23.3 10.1
После перестановки a, b: z x
```

Как отмечено в комментариях, при вызове функции *swapargs(i, j)* выполняется явно перегруженная версия функции *swapargs()*, определенная в программе. Компилятор в этом случае не генерирует эту версию обобщенной функции *swapargs()*, поскольку обобщенная функция переопределяется явно заданным вариантом перегруженной функции.

Для обозначения явной специализации функции можно использовать новый альтернативный синтаксис, содержащий ключевое слово *template*. Например, если задать специализацию с использованием этого альтернативного синтаксиса, перегруженная версия функции *swapargs()* из предыдущей программы будет выглядеть так.

```
// Использование нового синтаксиса задания специализации.
template<>
void swapargs<int> (int &a, int &b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
```

```
  cout << "Это int-специализация функции swapargs.\n";
}
```

Как видите, в новом синтаксисе для обозначения специализации используется конструкция *template<>*. Тип данных, для которых создается эта специализация, указывается в угловых скобках после имени функции. Для задания любого типа обобщенной функции используется один и тот же синтаксис. На данный момент ни один из синтаксических способов задания специализации не имеет никаких преимуществ перед другим, но с точки зрения перспективы развития языка, возможно, все же лучше использовать новый стиль.

Явная специализация шаблона позволяет спроектировать версию обобщенной функции в расчете на некоторую уникальную ситуацию, чтобы, возможно, воспользоваться преимуществами повышенного быстродействия программы только для одного типа данных. Но, как правило, если вам нужно иметь различные версии функции для разных типов данных, имеет смысл использовать перегруженные функции, а не шаблоны.

Перегрузка шаблона функции

Помимо создания явным образом перегруженных версий обобщенной функции, можно также перегружать саму спецификацию шаблона функции. Для этого достаточно создать еще одну версию шаблона, которая будет отличаться от остальных списком параметров. Рассмотрим пример.

```
// Объявление перегруженного шаблона функции.
#include <iostream>
using namespace std;

// Первая версия шаблона f().
template <class X>
void f(X a)
{
  cout << "Выполняется функция f(X a)\n";
}

// Вторая версия шаблона f().
template <class X, class Y>
void f(X a, Y b)
{
  cout << "Выполняется функция f(X a, Y b)\n";
}

int main()
{
  f(10); // Вызывается функция f(X).
  f(10, 20); // Вызывается функция f(X, Y).
  return 0;
}
```

Здесь шаблон для функции *f()* перегружается, чтобы обеспечить возможность приема как одного, так и двух параметров.

Использование стандартных параметров в шаблонных функциях

В шаблонных функциях можно смешивать стандартные параметры с обобщенными

параметрами типа. Эти параметры работают так же, как в любой другой функции. Рассмотрим пример.

```
// Использование стандартных параметров  в шаблонной функции.
#include <iostream>
using namespace std;

// Отображение данных заданное количество раз.
template<class X>
void repeat(X data, int times)
{
  do {
    cout << data << "\n";
    times--;
  }while(times);
}

int main( )
{
  repeat("Это тест.", 3);
  repeat(100, 5);
  repeat(99.0/2, 4);
  return 0;
}
```

Вот какие результаты генерирует эта программа.

```
Это тест.
Это тест.
Это тест.
100
100
100
100
100
49.5
49.5
49.5
49.5
```

В этой программе функция *repeat()* отображает свой первый аргумент столько раз, сколько задано ее вторым аргументом. Поскольку первый аргумент имеет обобщенный тип, функцию *repeat()* можно использовать для отображения данных любого типа. Параметр *times* — стандартный, он передается по значению. Смешанное задание обобщенных и необобщенных параметров, как правило, не вызывает никаких проблем и является обычной практикой программирования.

Ограничения при использовании обобщенных функций

Обобщенные функции подобны перегруженным функциям, но имеют больше ограничений по применению. При перегрузке функций в теле каждой из них обычно задаются различные

действия. Но обобщенная функция должна выполнять одно и то же действие для всех версий — отличие между версиями состоит только в типе данных. Рассмотрим пример, в котором перегруженные функции нельзя заменить обобщенной функцией, поскольку они выполняют различные действия,

```
void outdata(int i)
{
  cout << i;
}

void outdata(double d)
{
  cout << d * 3.1416;
}
```

Создание обобщенной функции abs()

Давайте-ка снова обратимся к функции *abs()*. Вспомните, что в главе 8 стандартные библиотечные функции *abs()*, *labs()* и *fabs()* были сгруппированы в три перегруженные функции с общим именем *myabs()*. Каждая из перегруженных версий функции *myabs()* предназначена для возврата абсолютного значения для данных "своего" типа. Несмотря на то что показанную в главе 8 перегрузку функции *abs()* можно считать шагом вперед по сравнению с использованием трех различных библиотечных функций (с различными именами), все же это не лучший способ создания функции, которая возвращает абсолютное значение заданного аргумента. Поскольку процедура возврата абсолютного значения числа одинакова для всех типов числовых значений, функция *abs()* может послужить прекрасным поводом для создания шаблонной функции. При наличии обобщенной версии функции *abs()* компилятор сможет автоматически создавать необходимую ее версию. Программист в этом случае освобождается от написания отдельных версий для каждого типа данных. (Кроме того, исходный код программы не будет загромождаться несколькими "вручную" перегруженными версиями.)

В следующей программе содержится обобщенная версия функции *myabs()*. Имеет смысл сравнить ее с перегруженными версиями, приведенными в главе 8. Нетрудно убедиться, что обобщенная версия короче и обладает большей гибкостью.

```
// Обобщенная версия функции myabs().
#include <iostream>
using namespace std;

template <class X>
X myabs(X val)
{
  return val < 0 ? -val : val;
}

int main()
{
  cout << myabs(-10) << "\n"; // для типа int
  cout << myabs(-10.0) << "\n"; // для типа double
  cout << myabs(-10L) << "\n"; // для типа long
  cout << myabs(-10.0F) << "\n"; // для типа float
```

```
  return 0;
}
```

В качестве упражнения было бы неплохо, если бы вы попытались найти другие библиотечные функции-кандидаты для переделки их в обобщенные функции. Помните, главное здесь то, что один и тот же алгоритм должен быть применим к широкому диапазону данных.

Обобщенные классы

Помимо обобщенных функций, можно также определить обобщенный класс. Для этого создается класс, в котором определяются все используемые им алгоритмы; при этом реальный тип обрабатываемых в нем данных будет задан как параметр при создании объектов этого класса.

Обобщенные классы особенно полезны в случае, когда в них используется логика, которую можно обобщить. Например, алгоритмы, которые поддерживают функционирование очереди целочисленных значений, также подходят и для очереди символов. Механизм, который обеспечивает поддержку связного списка почтовых адресов, также годится для поддержки связного списка, предназначенного для хранения данных о запчастях к автомобилям. После создания обобщенный класс сможет выполнять определенную программистом операцию (например, поддержку очереди или связного списка) для любого типа данных. Компилятор автоматически сгенерирует корректный тип объекта на основе типа, заданного при создании объекта.

Общий формат объявления обобщенного класса имеет следующий вид:

```
template <class Ttype> class имя_класса {
  .
  .
  .
}
```

Здесь элемент *Ttype* представляет собой *"заполнитель"* для имени типа, который будет задан при реализации класса. При необходимости можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятыми.

Создав обобщенный класс, можно создать его конкретный экземпляр, используя следующий общий формат.

```
имя_класса <тип> имя_объекта;
```

Здесь элемент *тип* означает имя типа данных, которые будут обрабатываться экземпляром обобщенного класса. Функции-члены обобщенного класса автоматически являются обобщенными. Поэтому вам не нужно использовать ключевое слово *template* для явного определения их таковыми.

В следующей программе класс *queue* (впервые представленный в главе 11) переделан в обобщенный. Это значит, что его можно использовать для организации очереди объектов любого типа. В данном примере создаются две очереди: для целых чисел и значений с плавающей точкой, но можно использовать данные и любого другого типа.

```
// Демонстрация использования обобщенного класса очереди.
#include <iostream>
using namespace std;

const int SIZE=100;

// Создание обобщенного класса queue.
template <class QType>
class queue{
```

```
    QType q[SIZE];
    int sloc, rloc;
  public:
    queue() { sloc = rloc =0; }
    void qput(QType i);
    QType qget();
};

// Занесение объекта в очередь.
template <class QType>
void queue<QType>::qput(QType i)
{
  if(sloc==SIZE) {
    cout << "Очередь заполнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}

// Извлечение объекта из очереди.
template <class QType>
QType queue<QType>::qget()
{
  if(rloc == sloc) {
    cout << "Очередь пуста.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}

int main()
{
  queue<int> a, b; // Создаем две очереди для целых чисел.

  a.qput(10);
  a.qput(20);
  b.qput(19);
  b.qput(1);

  cout << a.qget() << " ";
  cout << a.qget() << " ";
  cout << b.qget() << " ";
```

```
  cout << b.qget() << "\n";

  queue<double> c, d; // Создаем две очереди для double-значений.
  c.qput(10.12);
  c.qput(-20.0);
  d.qput(19.99);
  d.qput(0.986);

  cout << c.qget() << " ";
  cout << c.qget() << " ";
  cout << d.qget()<< " ";
  cout << d.qget() << "\n";

  return 0;
}
```

При выполнении этой программы получаем следующие результаты.

```
10 20 19 1
10.12 -20 19.99 0.986
```

В этой программе объявление обобщенного класса подобно объявлению обобщенной функции. Тип данных, хранимых в очереди, обобщен в объявлении класса. Он неизвестен до тех пор, пока не будет объявлен объект класса *queue*, который и определит реальный тип данных. После объявления конкретного экземпляра класса *queue* компилятор автоматически сгенерирует все функции и переменные, необходимые для обработки реальных данных. В данном примере объявляются два различных типа очереди: две очереди для хранения целых чисел и две очереди для значений типа *double*. Обратите особое внимание на эти объявления:

```
queue<int> a, b;
queue<double> c, d;
```

Заметьте, как указывается нужный тип данных: он заключается в угловые скобки. Изменяя тип данных при создании объектов класса *queue*, можно изменить тип данных, хранимых в очереди. Например, используя следующее объявление, можно создать еще одну очередь, которая будет содержать указатели на символы:

```
queue<char *> chrptrQ;
```

Можно также создавать очереди для хранения данных, тип которых создан программистом. Например, предположим, вы используете следующую структуру для хранения информации об адресе.

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
};
```

Тогда для того, чтобы с помощью класса *queue* сгенерировать очередь для хранения объектов типа *addr*, достаточно использовать такое объявление.

```
queue<addr> obj;
```

На примере класса *queue* нетрудно убедиться, что обобщенные функции и классы представляют собой мощные средства, которые помогут увеличить эффективность работы программиста, поскольку они позволяют определить общий формат объекта, который можно затем использовать с любым типом данных. Обобщенные функции и классы избавляют вас от утомительного труда по созданию отдельных реализаций для каждого типа данных, подлежащих обработке единым алгоритмом. Эту работу сделает за вас компилятор: он автоматически создаст конкретные версии определенного вами класса.

Пример класса с двумя обобщенными типами данных

Шаблонный класс может иметь несколько обобщенных типов данных. Для этого достаточно объявить все нужные типы данных в *template*-спецификации в виде элементов списка, разделяемых запятыми. Например, в следующей программе создается класс, который использует два обобщенных типа данных.

```
/* Здесь используется два обобщенных типа данных в определении класса.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2>
class myclass {
   Type1 i;
   Type2 j;
  public:
   myclass(Type1 a, Type2 b) { i = a; j = b; }

   void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('x', "Это тест.");

  ob1.show(); // отображение int- и double-значений
  ob2.show(); // отображение значений типа char и char *

  return 0;
}
```

Эта программа генерирует такие результаты.

```
10 0.23
X Это тест.
```

В данной программе объявляется два вида объектов. Объект *ob1* использует данные типа *int* и *double*, а объект *ob2* — символ и указатель на символ. Для этих ситуаций компилятор автоматически генерирует данные и функции, соответствующие способу создания объектов.

Создание обобщенного класса безопасного массива

Прежде чем двигаться дальше, рассмотрим еще одно приложение для обобщенного класса. Как было показано в главе 13, можно перегружать оператор *"[]"*, что позволяет создавать

собственные реализации массивов, в том числе и *¨безопасные массивы¨*, которые обеспечивают динамическую проверку нарушения границ. Как вы знаете, в C++ во время выполнения программы возможен выход за границы массива без выдачи сообщения об ошибке. Но если создать класс, который бы содержал массив, и разрешить доступ к этому массиву только через перегруженный оператор индексации (*"[]"*), то можно перехватить индекс, соответствующий адресу за пределами адресного пространства массива.

Объединив перегрузку оператора с обобщенным классом, можно создать обобщенный тип безопасного массива, который затем будет использован для создания безопасных массивов, предназначенных для хранения данных любого типа. Такой тип массива и создается в следующей программе.

```
// Пример создания и использования обобщенного безопасного массива.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType>
class atype {
   AType a[SIZE];
  public:
   atype() {
     register int i;
     for(i=0; i<SIZE; i++) a[i] = i;
   }

   AType &operator[](int i);
};

// Обеспечение контроля границ для класса atype.
 template <class AType>
AType &atype<AType>::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
    cout << "\n Значение индекса ";
    cout << i << " за пределами границ массива.\n";
  }
  return a [i];
}

int main()
{
  atype<int> intob; // массив int-значений
  atype<double> doubleob; // массив double-значений

  int i;

  cout << "Массив int-значений: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
```

```
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Массив double-значений: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // ошибка времени выполнения!
  return 0;
}
```

В этой программе сначала создается обобщенный тип безопасного массива, а затем демонстрируется его использование путем создания массива целых чисел и массива double-значений. Было бы неплохо, если бы вы попробовали создать массивы других типов. Как доказывает этот пример, одно из наибольших достоинств обобщенных классов состоит в том, что они позволяют только один раз написать код, отладить его, а затем применять его к данным любого типа, не переписывая его для каждого конкретного приложения.

Использование в обобщенных классах аргументов, не являющихся типами

В *template*-спецификации для обобщенного класса можно также задавать аргументы, не являющиеся типами. Это значит, что в шаблонной спецификации можно указывать то, что обычно принимается в качестве стандартного аргумента, например, аргумент типа int или аргумент-указатель. Синтаксис (он практически такой же, как при задании обычных параметров функции) включает определение типа и имени аргумента. Вот, например, как можно по-другому реализовать класс безопасного массива, представленного в предыдущем разделе.

```
// Использование в шаблоне аргументов, которые не являются типами.
#include <iostream>
#include <cstdlib>
using namespace std;

// Здесь элемент int size - это аргумент, не являющийся типом.
template <class AType, int size>
class atype {
   AType a[size]; // В аргументе size передается длина массива.
  public:
   atype() {
     register int i;
     for(i=0; i<size; i++) a[i] = i;
   }

   AType &operator[](int i);
};

// Обеспечение контроля границ для класса atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
```

```
{
  if(i<0 || i> size-1) {
    cout << "\n Значение индекса ";
    cout << i << " за пределами границ массива.\n";
    exit(1);
  }
  return a[i];
}

int main( )
{
  atype<int, 10> intob; // 10-элементный массив целых чисел
  atype<double, 15> doubleob; // 15-элементный массив double-значений

  int i;

  cout << "Массив целых чисел: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Массив double-значений: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // ошибка времени выполнения!
  return 0;
}
```

Рассмотрим внимательно *template*-спецификацию для класса *atype*. Обратите внимание на то, что аргумент *size* объявлен с указанием типа *int*. Этот параметр затем используется в теле класса *atype* для объявления размера массива *a*. Несмотря на то что в исходном коде программы член *size* имеет вид "переменной", его значение известно уже во время компиляции. Поэтому его можно успешно использовать для установки размера массива. Кроме того, значение "переменной" *size* используется для контроля выхода за границы массива в операторной функции *operator[]()*. Обратите также внимание на то, как в функции main() создается массив целых чисел и массив значений с плавающей точкой. При этом размер каждого из них определяется вторым параметром *template*-спецификации.

На тип параметров, которые не представляют типы, налагаются ограничения. В этом случае разрешено использовать только целочисленные типы, указатели и ссылки. Другие типы (например, float) не допускаются. Аргументы, которые передаются параметру, не являющемуся типом, должны содержать либо целочисленную константу, либо указатель или ссылку на глобальную функцию или объект. Таким образом, эти "нетиповые" параметры следует рассматривать как константы, поскольку их значения не могут быть изменены. Например, в теле функции *operator[]()* следующая инструкция недопустима:

```
size = 10; // ошибка
```

Поскольку параметры-*"нетипы"* обрабатываются как константы, их можно использовать для установки размерности массива, что существенно облегчает жизнь программисту.

Как показывает пример создания безопасного массива, использование *"нетиповых"*

Как показывает пример создания безопасного массива, использование *"нетиповых"* параметров весьма расширяет сферу применения шаблонных классов. И хотя информация, передаваемая через *"нетиповой"* аргумент, должна быть известна во время компиляции, малость этого ограничения несравнима с достоинствами, предлагаемыми такими параметрами.

Шаблонный класс *queue*, представленный выше в этой главе, также выиграл бы от применения к нему *"нетипового"* параметра, задающего размер очереди. В качестве упражнения попробуйте усовершенствовать класс *queue* самостоятельно.

Использование в шаблонных классах аргументов по умолчанию

Шаблонный класс может по умолчанию определять аргумент, соответствующий обобщенному типу. Например, в результате такой *template*-спецификации

```
template <class X=int>
class myclass { //...
};
```

будет использован тип *int*, если при создании объекта класса *myclass* отсутствует задание какого-то бы то ни было типа.

Для аргументов, которые не представляют тип в *template*-спецификации, также разрешается задавать значений по умолчанию. Они используются в случае, если при реализации класса значение для такого аргумента явно не указано. Аргументы по умолчанию для *"нетиповых"* параметров задаются с помощью синтаксиса, аналогичного используемому при задании аргументов по умолчанию для параметров функций.

Рассмотрим еще одну версию класса безопасного массива, в котором используются аргументы по умолчанию как для типа данных, так и для размера массива.

```
// Демонстрация использования шаблонных аргументов по умолчанию.
#include <iostream>
#include <cstdlib>
using namespace std;

/* Здесь параметр AType по умолчанию принимает тип int. а параметр size
по умолчанию устанавливается равным 10.
*/
template <class AType=int, int size=10>
class atype{
   AType a[size]; // Через параметр size передается размер массива.
  public:
   atype() {
     register int i;
     for(i=0; i<size; i++) a[i] = i;
   }

   AType &operator[](int i);
};

// Обеспечение контроля границ для класса atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
```

```
  if( i<0 || i> size-1) {
    cout << "\n Значение индекса ";
    cout << i << " за пределами границ массива.\n";
    exit(1);
  }
  return a[i];
}

int main( )
{
  atype<int, 100> intarray; /* 100-элементный массив целых чисел */
   atype<double> doublearray; /* 10-элементный массив double-значений (размер массива установлен по умолчанию) */
  atype<> defarray; /* 10-элементный массив int-значений (размер и тип int установлены по умолчанию) */

  int i;

  cout << "Массив целых чисел: ";
  for(i=0; i<100; i++ ) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << " ";
  cout << '\n';

  cout << "Массив double-значений: ";
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << " ";
  cout << '\n';

  cout << "Массив по умолчанию: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << " ";
  cout << '\n';

  return 0;
}
```

Обратите особое внимание на эту строку:

```
template <class AType=int, int size=10>
class atype {
```

Здесь параметр *AType* по умолчанию заменяется типом *int*, а параметр *size* по умолчанию устанавливается равным числу *10*. Как показано в этой программе, объекты класса *atype* можно создать тремя способами:

■ путем явного задания как типа, так и размера массива;

■ задав явно лишь тип массива, при этом его размер по умолчанию устанавливается равным *10* элементам;

■ вообще без задания типа и размера массива, при этом он по умолчанию будет хранить элементы типа *int*, а его размер по умолчанию устанавливается равным *10*.

Использование аргументов по умолчанию (особенно типов) делает шаблонные классы еще более гибкими. Тип, используемый по умолчанию, можно предусмотреть для наиболее употребительного типа данных, позволяя при этом пользователю ваших классов задавать при необходимости нужный тип данных.

Явно задаваемые специализации классов

Подобно шаблонным функциям можно создавать и специализации обобщенных классов. Для этого используется конструкция *template<>*, которая работает по аналогии с явно задаваемыми специализациями функций. Рассмотрим пример.

```
// Демонстрация специализации класса.
#include <iostream>
using namespace std;

template <class T>
class myclass {
   T x;
  public:
   myclass(T a) {
     cout << "В теле обобщенного класса myclass.\n";
     x = a;
   }

   T getx() { return x; }
};

// Явная специализация для типа int.
template <>
class myclass<int> {
   int x;
  public:
   myclass(int a) {
     cout << "В теле специализации myclass<int>.\n";
     x = a * a;
   }
   int getx() { return x; }
};

int main()
{
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";

  return 0;
}
```

При выполнении данная программа отображает такие результаты.

```
В теле обобщенного класса myclass.
double: 10.1
В теле специализации myclass<int>.
int: 25
```

В этой программе обратите особое внимание на следующую строку.

```
template <>
class myclass<int> {
```

Она уведомляет компилятор о том, что создается явная *int*-специализация класса *myclass*. Тот же синтаксис используется и для любого другого типа специализации класса.

Явная специализация классов расширяет диапазон применения обобщенных классов, поскольку она позволяет легко обрабатывать один или два специальных случая, оставляя все остальные варианты для автоматической обработки компилятором. Но если вы заметите, что у вас создается слишком много специализаций, то тогда, возможно, лучше вообще отказаться от создания шаблонного класса.

Глава 17: Обработка исключительных ситуации

Эта глава посвящена обработке исключительных ситуаций. *Исключительная ситуация* (или *исключение*) — это ошибка, которая возникает во время выполнения программы. Используя C++-подсистему обработки исключительных ситуаций, с такими ошибками вполне можно справляться. При их возникновении во время работы программы автоматически вызывается так называемый *обработчик исключений*. Теперь программист не должен обеспечивать проверку результата выполнения каждой конкретной операции или функции вручную. В этом-то и состоит принципиальное преимущество системы обработки исключений, поскольку именно она *"отвечает"* за код обработки ошибок, который прежде приходилось *"вручную"* вводить в и без того объемные программы.

В этой главе мы также возвращаемся к C++-операторам динамического распределения памяти: *new* и *delete*. Как разъяснялось выше в этой книге, если оператор *new* не может выделить требуемую память, он генерирует исключение. И здесь мы узнаем, как именно обрабатывается такое исключение. Кроме того, вы научитесь перегружать операторы *new* и *delete*, что позволит вам определять собственные схемы выделения памяти.

Основы обработки исключительных ситуаций

Обработка исключений — это системные средства, с помощью которых программа может справиться с ошибками времени выполнения.

Управление C++-механизмом обработки исключений зиждется на трех ключевых словах: *try*, *catch* и *throw*. Они образуют взаимосвязанную подсистему, в которой использование одного из них предполагает применение другого. Для начала будет полезно получить общее представление о роли, которую они играют в обработке исключительных ситуаций. Если кратко, то их работа состоит в следующем. Программные инструкции, которые вы считаете нужным проконтролировать на предмет исключений, помещаются в *try*-блок. Если исключение (т.е. ошибка) таки возникает в этом блоке, оно дает знать о себе *выбросом* определенного рода информации (с помощью ключевого слова *throw*). Это *выброшенное исключение* может быть перехвачено программным путем с помощью *catch*-блока и обработано соответствующим образом. А теперь подробнее.

Инструкция throw генерирует исключение, которое перехватывается catch-инструкцией.

Итак, код, в котором возможно возникновение исключительных ситуаций, должен выполняться в рамках *try*-блока. (Любая функция, вызываемая из этого *try*-блока, также подвергается контролю.) Исключения, которые могут быть выброшены контролируемым кодом, перехватываются *catch*-инструкцией, непосредственно следующей за *try*-блоком, в котором фиксируются эти *"выбросы"* исключений. Общий формат *try*- и *catch*-блоков

котором фиксируются эти "выбросы" исключений. Общий формат *try*- и *catch*-блоков выглядит так.

```
try {
  // try-блок (блок кода, подлежащий проверке на наличие ошибок)
}

catch (type1 arg) {
  // catch-блок (обработчик исключения типа type1)
}

catch {type2 arg) {
  // catch-блок (обработчик исключения типа type2)
}

catch {type3 arg) {
  // catch-блок (обработчик исключения типа type3)
}

  // ...

catch (typeN arg) {
  // catch-блок (обработчик исключения типа typeN)
}
```

Блок *try* должен содержать код, который, по вашему мнению, должен проверяться на предмет возникновения ошибок. Этот блок может включать лишь несколько инструкций некоторой функции либо охватывать весь код функции main() (в этом случае, по сути, "под колпаком" системы обработки исключений будет находиться вся программа).

После "выброса" исключение перехватывается соответствующей инструкцией *catch*, которая выполняет его обработку. С одним *try*-блоком может быть связана не одна, а несколько *catch*-инструкций. Какая именно из них будет выполнена, определяется типом исключения. Другими словами, будет выполнена та *catch*-инструкция, тип исключения которой (т.е. тип данных, заданный в *catch*-инструкции) совпадает с типом сгенерированного исключения (а все остальные будут проигнорированы). После перехвата исключения параметр *arg* примет его значение. Таким путем могут перехватываться данные любого типа, включая объекты классов, созданных программистом.

Чтобы исключение было перехвачено, необходимо обеспечить его "выброс" в try-блоке.

Общий формат инструкции *throw* выглядит так:

```
throw exception;
```

Здесь с помощью элемента *exception* задается исключение, сгенерированное инструкцией *throw*. Если это исключение подлежит перехвату, то инструкция *throw* должна быть выполнена либо в самом блоке *try*, либо в любой вызываемой из него функции (т.е. прямо или косвенно).

На заметку. *Если в программе обеспечивается "выброс" исключения, для которого не предусмотрена соответствующая catch-инструкция, произойдет аварийное завершение программы, вызываемое стандартной библиотечной функцией terminate(). По умолчанию функция terminate() вызывает функцию abort() для остановки программы, но при желании можно определить собственный обработчик ее завершения. За подробностями относительно обработки этой ситуации следует обратиться к документации, прилагаемой к вашему компилятору*.

Рассмотрим простой пример обработки исключений средствами языка C++

Рассмотрим простой пример обработки исключений средствами языка C++.

```
// Простой пример обработки исключений.
#include <iostream>
using namespace std;

int main()
{
  cout << "НАЧАЛО\n";
  try {
    // начало try-блока
    cout << "В try-блоке\n";
    throw 99; // генерирование ошибки
    cout << "Эта инструкция не будет выполнена.";
  }

  catch (int i) {
    // перехват ошибки
    cout << "Перехват исключения. Его значение равно: ";
    cout << i << "\n";
  }

  cout << "КОНЕЦ";

  return 0;
}
```

При выполнении эта программа отображает следующие результаты.

```
НАЧАЛО В try-блоке
Перехват исключения. Его значение равно: 99
КОНЕЦ
```

Рассмотрим внимательно код этой программы. Как видите, здесь *try*-блок содержит три инструкции, а инструкция *catch(int i)* предназначена для обработки исключения целочисленного типа. В этом *try*-блоке выполняются только две из трех инструкций: *cout* и *throw*. После генерирования исключения управление передается *catch*-выражению, при этом выполнение *try*-блока прекращается. Необходимо понимать, что *catch*-инструкция не вызывается, а просто с нее продолжается выполнение программы после "выброса" исключения. (Стек программы автоматически настраивается в соответствии с создавшейся ситуацией.) Поэтому *cout*-инструкция, следующая после *throw*-инструкции, никогда не выполнится.

После выполнения *catch*-блока управление программой передается инструкции, следующей за этим блоком. Поэтому ваш обработчик исключения должен исправить ошибку, вызвавшую его возникновение, чтобы программа могла нормально продолжить выполнение. В случаях, когда ошибку исправить нельзя, *catch*-блок обычно завершается обращением к функциям *exit()* или *abort()*. (Функции *exit()* и *abort()* описаны в разделе "Копнем глубже" ниже в этой главе.)

Как упоминалось выше, тип исключения должен совпадать с типом, заданным в *catch*-инструкции. Например, если в предыдущей программе тип *int*, указанный в *catch*-выражении, заменить типом *double*, то исключение перехвачено не будет, и произойдет аварийное завершение программы. Вот как выглядят последствия внесения такого изменения.

```
// Этот пример работать не будет.
#include <iostream>
using namespace std;

int main()
{
  cout << "НАЧАЛО\n";

  try {
    // начало try-блока
    cout << "В try-блоке\n";
    throw 99; // генерирование ошибки
    cout << "Эта инструкция не будет выполнена.";
  }

  catch (double i) {
    // Перехват исключения типа int не состоится.
    cout << "Перехват исключения. Его значение равно: ";
    cout << i << "\n";
  }
  cout << "КОНЕЦ";

  return 0;
}
```

Такие результаты выполнения этой программы объясняются тем, что исключение целочисленного типа не перехватывается инструкцией *catch (double i)*.

```
НАЧАЛО
В try-блоке
Abnormal program termination
```

Функции exit() и abort()

Функции *exit()* и *abort()* входят в состав стандартной библиотеки C++ и часто используются в программировании на C++. Обе они обеспечивают завершение программы, но по-разному.

Вызов функции *exit()* немедленно приводит к *"правильному"* прекращению программы. (*"Правильное"* окончание означает выполнение стандартной последовательности действий по завершению работы.) Обычно этот способ завершения работы используется для остановки программы при возникновении неисправимой ошибки, которая делает дальнейшее ее выполнение бессмысленным или опасным. Для использования функции *exit()* требуется включить в программу заголовок *<cstdlib>*. Ее прототип выглядит так.

```
void exit(int status);
```

Поскольку функция *exit()* вызывает немедленное завершение программы, она не передает управление вызывающему процессу и не возвращает никакого значения. Тем не менее вызывающему процессу в качестве кода завершения передается значение параметра *status*. По соглашению нулевое значение параметра *status* говорит об успешном окончании работы программы. Любое другое его значение свидетельствует о завершении программы по ошибке. Для индикации успешного окончания можно также использовать константу *EXIT_SUCCESS*, а для индикации ошибки— константу *EXIT_FAILURE*. Эти константы определены в заголовке *<cstdlib>*.

Прототип функции *abort()* выглядит так:

```
void abort( );
```

Аналогично *exit()* функция *abort()* вызывает немедленное завершение программы. Но в отличие от функции *exit()* она не возвращает операционной системе никакой информации о статусе завершения и не выполняет стандартной (*"правильной"*) последовательности действий при остановке программы. Для использования функции *abort()* требуется включить в программу заголовок *<cstdlib>*. Функцию *abort()* можно назвать аварийным *"стоп-краном"* для C++-программы. Ее следует использовать только после возникновения неисправимой ошибки.

Последнее сообщение об аварийном завершении программы (*Abnormal program termination*) может отличаться от приведенного в результатах выполнения предыдущего примера. Это зависит от используемого вами компилятора.

Исключение, сгенерированное функцией, вызванной из *try*-блока, может быть перехвачено этим же *try*-блоком. Рассмотрим, например, следующую вполне корректную программу.

```
/* Генерирование исключения из функции, вызываемой из try-блока.
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "В функции Xtest( ), значение test равно: "<< test << "\n";
  if(test) throw test;
}

int main( )
{
  cout << "НАЧАЛО\n";
  try {
    // начало try-блока
    cout << "В try-блоке\n";
    Xtest (0);
    Xtest (1);
    Xtest (2);
  }

  catch (int i) {
    // перехват ошибки
    cout << "Перехват исключения. Его значение равно: ";
    cout << i << "\n";
  }

  cout << "КОНЕЦ";

  return 0;
}
```

Эта программа генерирует такие результаты.

```
НАЧАЛО В try-блоке
В функции Xtest(), значение test равно: 0
В функции Xtest(), значение test равно: 1
Перехват исключения. Его значение равно: 1
КОНЕЦ
```

Блок *try* может быть локализован в рамках функции. В этом случае при каждом ее выполнении запускается и обработка исключений, связанная с этой функцией. Рассмотрим следующую простую программу.

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

/* Функционирование блоков try/catch возобновляется при каждом входе в функцию.
*/
void Xhandler(int test)
{
  try {
    if(test) throw test;
  }
  catch(int i) {
    cout << "Перехват! Исключение №: " << i << '\n';
  }
}

int main()
{
  cout << "НАЧАЛО\n ";

  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);

  cout << "КОНЕЦ";

  return 0;
}
```

При выполнении этой программы отображаются такие результаты.

```
НАЧАЛО
Перехват! Исключение №:1
Перехват! Исключение №:2
Перехват! Исключение №:3
КОНЕЦ
```

Как видите, программа сгенерировала три исключения. После каждого исключения

функция *Xhandler()* передавала управление в функцию main(). Когда она снова вызывалась, возобновлялась и обработка исключения.

В общем случае *try*-блок возобновляет свое функционирование при каждом входе в него. Поэтому *try*-блок, который является частью цикла, будет запускаться при каждом повторении этого цикла.

Перехват исключений классового типа

Исключение может иметь любой тип, в том числе и тип класса, созданного программистом. В реальных программах большинство исключений имеют именно тип класса, а не встроенный тип. Вероятно, тип класса больше всего подходит для описания ошибки, которая потенциально может возникнуть в программе. Как показано в следующем примере, информация, содержащаяся в объекте класса исключений, позволяет упростить обработку исключений.

```
// Использование класса исключений.
#include <iostream>
#include <cstring>
using namespace std;

class MyException {
  public:
    char str_what[80];

    MyException( ) { *str_what =0; }
    MyException(char *s) { strcpy(str_what, s);}
};

int main( )
{
  int a, b;
  try {
    cout << "Введите числитель и знаменатель: ";
      cin >> a >> b;

    if( !b) throw MyException("Делить на нуль нельзя!");
    else
      cout << "Частное равно " << a/b << "\n";
  }

  catch (MyException e) {
    // перехват ошибки
    cout << e.str_what << "\n";
  }

  return 0;
}
```

Вот один из возможных результатов выполнения этой программы.

```
Введите числитель и знаменатель: 10 0
Делить на нуль нельзя!
```

После запуска программы пользователю предлагается ввести числитель и знаменатель

После запуска программы пользователю предлагается ввести числитель и знаменатель. Если знаменатель равен нулю, создается объект класса *MyException*, который содержит информацию о попытке деления на нуль. Таким образом, класс *MyException* инкапсулирует информацию об ошибке, которая затем используется обработчиком исключений для уведомления пользователя о случившемся.

Безусловно, реальные классы исключений гораздо сложнее класса *MyException*. Как правило, создание классов исключений имеет смысл в том случае, если они инкапсулируют информацию, которая бы позволила обработчику исключений эффективно справиться с ошибкой и по возможности восстановить работоспособность программы.

Использование нескольких catch-инструкций

Как упоминалось выше, с *try*-блоком можно связывать не одну, а несколько *catch*-инструкций. В действительности именно такая практика и является обычной. Но при этом все *catch*-инструкции должны перехватывать исключения различных типов. Например, в приведенной ниже программе обеспечивается перехват как целых чисел, так и указателей на символы.

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

// Здесь возможен перехват исключений различных типов.
void Xhandler(int test)
{
  try {
    if(test) throw test;
    else throw "Значение равно нулю.";
  }

  catch (int i) {
  cout << "Перехват! Исключение №: " << i << '\n';
  }

  catch(char *str) {
    cout << "Перехват строки: ";
    cout << str << '\n';
  }
}

int main()
{
  cout << "НАЧАЛО\n";

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  cout << "КОНЕЦ";

  return 0;
}
```

Эта программа генерирует такие результаты.

```
НАЧАЛО
Перехват! Исключение №: 1
Перехват! Исключение №: 2
Перехват строки: Значение равно нулю.
Перехват! Исключение №: 3
КОНЕЦ
```

Как видите, каждая *catch*-инструкция отвечает только за исключение *"своего"* типа. В общем случае *catch*-выражения проверяются в порядке следования, и выполняется только тот *catch*-блок, в котором тип заданного исключения совпадает с типом сгенерированного исключения. Все остальные *catch*-блоки игнорируются.

Перехват исключений базового класса

Важно понимать, как выполняются *catch*-инструкции, связанные с производными классами. Дело в том, что *catch*-выражение для базового класса *"отреагирует совпадением"* на исключение любого производного типа (т.е. типа, выведенного из этого базового класса). Следовательно, если нужно перехватывать исключения как базового, так и производного типов, в *catch*-последовательности *catch*-инструкцию для производного типа необходимо поместить перед *catch*-инструкцией для базового типа. В противном случае *catch*-выражение для базового класса будет перехватывать (помимо "своих") и исключения всех производных классов. Рассмотрим, например, следующую программу:

```
// Перехват исключений базовых и производных типов.
#include <iostream>
using namespace std;

class B {
};

class D: public B {
};

int main()
{
  D derived;

  try {
    throw derived;
  }

  catch(B b) {
    cout << "Перехват исключения базового класса.\n";
  }

  catch(D d) {
  cout << "Этот перехват никогда не произойдет.\n";
  }

  return 0;
}
```

Поскольку здесь объект *derived* — это объект класса *D*, который выведен из базового класса *B*, то исключение типа *derived* будет всегда перехватываться первым *catch*-выражением; вторая же *catch*-инструкция при этом никогда не выполнится. Одни компиляторы отреагируют на такое положение вещей предупреждающим сообщением. Другие могут выдать сообщение об ошибке. В любом случае, чтобы исправить ситуацию, достаточно поменять порядок следования этих *catch*-инструкций на противоположный.

Варианты обработки исключений

Помимо рассмотренных, существуют и другие C++-средства обработки исключений, которые создают определенные удобства для программистов. О них и пойдет речь в этом разделе.

Перехват всех исключений

Иногда имеет смысл создать обработчик для перехвата всех исключений, а не исключений только определенного типа. Для этого достаточно использовать такой формат *catch*-блока.

```
catch (...) {
  // Обработка всех исключений
}
```

Здесь заключенное в круглые скобки многоточие обеспечивает совпадение с любым типом данных.

Использование формата *catch(...)* иллюстрируется в следующей программе.

```
// В этой программе перехватываются исключения всех типов.
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try {
    if(test==0) throw test; // генерирует int-исключение
    if(test==1) throw 'a'; // генерирует char-исключение
    if(test==2) throw 123.23; // генерирует double-исключение
  }

  catch (...) { // перехват всех исключений
    cout << "Перехват!\n";
  }
}

int main()
{
  cout << "НАЧАЛО\n";

  Xhandler (0);
  Xhandler (1);
  Xhandler (2);

  cout << "КОНЕЦ";

  return 0;
}
```

```
}
```

Эта программа генерирует такие результаты.

```
НАЧАЛО
Перехват!
Перехват!
Перехват!
КОНЕЦ
```

Как видите, все три *throw*-исключения перехвачены с помощью одной-единственной *catch*-инетрукции.

Зачастую имеет смысл использовать инструкцию *catch(...)* в качестве последнего *"рубежа"* *catch*-последовательности. В этом случае она обеспечивает перехват исключений *"всех остальных"* типов (т.е. не предусмотренных предыдущими *catch*-выражениями). Например, рассмотрим еще одну версию предыдущей программы, в которой явным образом обеспечивается перехват исключений целочисленного типа, а перехват всех остальных возможных исключений *"взваливается на плечи"* инструкции *catch(...)*.

```
/* Использование формата catch (...) в качестве варианта "все остальное".
*/
#include <iostream>
using namespace std;

void Xhandler(int test)
{
  try {
    if(test==0) throw test; // генерирует int-исключение
    if(test==1) throw 'a'; // генерирует char-исключение
    if(test==2) throw 123.23; // генерирует double-исключение
  }

  catch(int i) {
    // перехватывает int-исключение
    cout << "Перехват " << i << '\n';
  }

  catch(...) {
    // перехватывает все остальные исключения
    cout << "Перехват-перехват!\n";
  }
}

int main()
{
  cout << "НАЧАЛО\n";

  Xhandler(0);
  Xhandler(1);
```

```
  Xhandler(1);
  Xhandler(2);

  cout << "КОНЕЦ";

  return 0;
}
```

Результаты, сгенерированные при выполнении этой программы, таковы.

```
НАЧАЛО
Перехват 0
Перехват-перехват!
Перехват-перехват!
КОНЕЦ
```

Как подтверждает этот пример, использование формата *catch(...)* в качестве "последнего оплота" *catch*-последовательности— это удобный способ перехватить все исключения, которые вам не хочется обрабатывать в явном виде. Кроме того, перехватывая абсолютно все исключения, вы предотвращаете возможность аварийного завершения программы, которое может быть вызвано каким-то непредусмотренным (а значит, необработанным) исключением.

Ограничения, налагаемые на тип исключений, генерируемых функциями

Существуют средства, которые позволяют ограничить тип исключений, которые может генерировать функция за пределами своего тела. Можно также оградить функцию от генерирования каких бы то ни было исключений вообще. Для формирования этих ограничений необходимо внести в определение функции *throw*-выражение. Общий формат определения функции с использованием *throw*-выражения выглядит так.

```
тип имя_функции(список_аргументов) throw(список_имен_типов)
{
  // . . .
}
```

Здесь элемент *список_имен_типов* должен включать только те имена типов данных, которые разрешается генерировать функции (элементы списка разделяются запятыми). Генерирование исключения любого другого типа приведет к аварийному окончанию программы. Если нужно, чтобы функция вообще не могла генерировать исключения, используйте в качестве этого элемента пустой список.

На заметку. *При попытке сгенерировать исключение, которое не поддерживается функцией, вызывается стандартная библиотечная функция unexpected(). По умолчанию она вызывает функцию abort(), которая обеспечивает аварийное завершение программы. Но при желании можно задать собственный обработчик процесса завершения. За подробностями обращайтесь к документации, прилагаемой к вашему компилятору.*

На примере следующей программы показано, как можно ограничить типы исключений, которые способна генерировать функция.

```
/* Ограничение типов исключений, генерируемых функцией.
*/
#include <iostream>
using namespace std;

/* Эта функция может генерировать  исключения только типа int, char и
double.
```

```
*/
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // генерирует int-исключение
  if(test==1) throw 'a'; // генерирует char-исключение
  if(test==2) throw 123.23; // генерирует double-исключение
}

int main()
{
  cout << "НАЧАЛО\n";

  try {
     Xhandler(0);  //  Попробуйте  также  передать  функции  Xhandler()
аргументы 1 и 2.
  }

  catch(int i) {
    cout << "Перехват int-исключения.\n";
  }

  catch(char c) {
    cout << "Перехват char-исключения.\n";
  }

  catch(double d) {
    cout << "Перехват double-исключения.\n";
  }

  cout << "КОНЕЦ";

  return 0;
}
```

В этой программе функция *Xhandler()* может генерировать исключения только типа *int*, *char* и *double*. При попытке сгенерировать исключение любого другого типа произойдет аварийное завершение программы (благодаря вызову функции *unexpected()*). Чтобы убедиться в этом, удалите из *throw*-списка, например, тип *int* и перезапустите программу.

Важно понимать, что диапазон исключений, разрешенных для генерирования функции, можно ограничивать только типами, генерируемыми ею в *try*-блоке, из которого была вызвана. Другими словами, любой *try*-блок, расположенный в теле самой функции, может генерировать исключения любого типа, если они перехватываются в теле той же функции. Ограничение применяется только для ситуаций, когда *"выброс"* исключений происходит за пределы функции.

Следующее изменение помешает функции *Xhandler()* генерировать любые изменения.

```
// Эта функция вообще не может генерировать исключения!
void Xhandler(int test) throw()
{
  /* Следующие инструкции больше не работают. Теперь они могут вызвать
лишь аварийное завершение программы. */
```

```
лишь аварийное завершение программы. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

На заметку. *На момент написания этой книги среда Visual C++ не обеспечивала для функции запрет генерировать исключения, тип которых не задан в throw-выражении. Это говорит о нестандартном поведении данной среды. Тем не менее вы все равно можете задавать "ограничивающее" throw-выражение, но оно в этом случае будет играть лишь уведомительную роль.*

Повторное генерирование исключения

Для того чтобы повторно сгенерировать исключение в его обработчике, воспользуйтесь *throw*-инструкцией без указания типа исключения. В этом случае текущее исключение будет передано во внешнюю *try/catch*-последовательность. Чаще всего причиной для такого выполнения инструкции *throw* служит стремление позволить доступ к одному исключению нескольким обработчикам. Например, первый обработчик исключений будет сообщать об одном аспекте исключения, а второй — о другом. Исключение можно повторно сгенерировать только в *catch*-блоке (или в любой функции, вызываемой из этого блока). При повторном генерировании исключение не будет перехватываться той же *catch*-инструкцией. Оно распространится на ближайшую *try/catch*-последовательность.

Повторное генерирование исключения демонстрируется в следующей программе (в данном случае повторно генерируется тип *char* *).

```
// Пример повторного генерирования исключения.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "Привет"; // генерирует исключение типа char *
  }
  catch(char *) { // перехватывает исключение типа char *
    cout << "Перехват исключения в функции Xhandler.\n";
     throw; // Повторное генерирование исключения типа char *, которое будет перехвачено  вне функции Xhandler.
  }
}

int main()
{
  cout << "НАЧАЛО\n";

  try {
    Xhandler();
  }

  catch(char *) {
```

```
    cout << "Перехват исключения в функции main().\n";
  }

  cout << "КОНЕЦ";
  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
НАЧАЛО
Перехват исключения в функции Xhandler.
Перехват исключения в функции main().
КОНЕЦ
```

Обработка исключений, сгенерированных оператором new

В главе 9 вы узнали, что оператор *new* генерирует исключение, если не удается удовлетворить запрос на выделение памяти. Поскольку тема исключений рассматривается только в этой главе, описание обработки исключений этого типа было отложено *"на потом"*. Вот теперь настало время об этом поговорить.

Для начала необходимо отметить, что в этом разделе описывается поведение оператора *new* в соответствии со стандартом C++. Как было отмечено в главе 9, действия, выполняемые системой при неуспешном использовании оператора *new*, с момента изобретения языка C++ изменялись уже несколько раз. Сначала оператор *new* возвращал при неудаче значение *null*. Позже такое поведение было заменено генерированием исключения. Кроме того, несколько раз менялось имя этого исключения. Наконец, было решено, что оператор *new* будет генерировать исключения по умолчанию, но в качестве альтернативного варианта он может возвращать и нулевой указатель. Следовательно, оператор *new* в разное время был реализован различными способами. И хотя все современные компиляторы реализуют оператор *new* в соответствии со стандартом C++, компиляторы более "почтенного" возраста могут содержать отклонения от него. Если приведенные здесь примеры программ не работают с вашим компилятором, обратитесь к документации, прилагаемой к компилятору, и поинтересуйтесь, как именно он реализует функционирование оператора *new*.

Согласно стандарту C++ при невозможности удовлетворить запрос на выделение памяти, требуемой оператором *new*, генерируется исключение типа *bad_alloc*. Если ваша программа не перехватит его, она будет досрочно завершена. Хотя такое поведение годится для коротких примеров программ, в реальных приложениях необходимо перехватывать это исключение и разумно обрабатывать его. Чтобы получить доступ к исключению типа *bad_alloc*, нужно включить в программу заголовок *<new>*.

Рассмотрим пример использования оператора *new*, заключенного в *try/catch*-блок для отслеживания неудачных результатов запроса на выделение памяти.

```
// Обработка исключений, генерируемых оператором new.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  try {
    p = new int[32]; // запрос на выделение памяти для 32-элементного int-массива
  }
```

```
  }
  catch (bad_alloc xa) {
    cout << "Память не выделена.\n";
    return 1;
  }
  for(i=0; i<32; i++) p[i] = i;
  for(i=0; i<32; i++ ) cout << p[i] << " ";
  delete [] p; // освобождение памяти
  return 0;
}
```

При неудачном выполнении оператора *new* исключение в этой программе будет перехвачено *catch*-инструкцией. Этот же подход можно использовать для отслеживания любых ошибок, связанных с использованием оператора *new*: достаточно заключить каждую *new*-инструкцию в *try*-блок.

Альтернативная форма оператора new — nothrow

Стандарт C++ при неудачной попытке выделения памяти вместо генерирования исключения также позволяет оператору *new* возвращать значение *null*. Эта форма использования оператора *new* особенно полезна при компиляции старых программ с применением современного C++-компилятора. Это средство также очень полезно при замене вызовов функции *malloc()* оператором *new*. (Это обычная практика при переводе C-кода на язык C++.) Итак, этот формат оператора *new* выглядит следующим образом.

```
p_var = new(nothrow) тип;
```

Здесь элемент *p_var*— это указатель на переменную типа *mun*. Этот *nothrow*-формат оператора *new* работает подобно оригинальной версии оператора *new*, которая использовалась несколько лет назад. Поскольку оператор *new (nothrow)* возвращает при неудаче значение *null*, его можно "внедрить" в старый код программы, не прибегая к обработке исключений. Однако в новых программах на C++ все же лучше иметь дело с исключениями.

В следующем примере показано, как используется альтернативный вариант *new (nothrow)*. Нетрудно догадаться, что перед вами вариация на тему предыдущей программы.

```
// Использование nothrow-версии оператора new.
#include <iostream>
#include <new>
using namespace std;
int main()
{
  int *p, i;
  p = new(nothrow) int[32]; // использование nothrow-версии
  if(!p) {
    cout << "Память не выделена.\n";
    return 1;
  }
  for(i=0; i<32; i++) p[i] = i;
```

```
  for(i=0; i<32; i++) p[i] = i;
  for(i=0; i<32; i++ ) cout << p[i] << " ";

  delete [] p; // освобождение памяти

  return 0;
}
```

Здесь при использовании *nothrow*-версии после каждого запроса на выделение памяти необходимо проверять значение указателя, возвращаемого оператором *new*.

Перегрузка операторов new и delete

Поскольку *new* и *delete* — операторы, их также можно перегружать. Несмотря на то что перегрузку операторов мы рассматривали в главе 13, тема перегрузки операторов *new* и *delete* была отложена до знакомства с темой исключений, поскольку правильно перегруженная версия оператора *new* (та, которая соответствует стандарту C++) должна в случае неудачи генерировать исключение типа *bad_alloc*. По ряду причин вам имеет смысл создать собственную версию оператора *new*. Например, создайте процедуры выделения памяти, которые, если область кучи окажется исчерпанной, автоматически начинают использовать дисковый файл в качестве виртуальной памяти. В любом случае реализация перегрузки этих операторов не сложнее перегрузки любых других.

Ниже приводится скелет функций, которые перегружают операторы *new* и *delete*.

```
// Выделение памяти для объекта.
void *operator new(size_t size)
{
  /* В случае невозможности выделить память генерируется исключение типа bad_alloc. Конструктор вызывается автоматически. */
  return pointer_to_memory;
}

// Удаление объекта.
void operator delete(void *p)
{
  /* Освобождается память, адресуемая указателем p. Деструктор вызывается автоматически. */
}
```

Тип *size_t* специально определен, чтобы обеспечить хранение размера максимально возможной области памяти, которая может быть выделена для объекта. (Тип *size_t*, по сути, —это целочисленный тип без знака.) Параметр *size* определяет количество байтов памяти, необходимых для хранения объекта, для которого выделяется память. Другими словами, это объем памяти, который должна выделить ваша версия оператора *new*. Перегруженная функция *new* должна возвращать указатель на выделяемую ею память или генерировать исключение типа *bad_alloc* в случае возникновении ошибки. Помимо этих ограничений, перегруженная функция *new* может выполнять любые нужные действия. При выделении памяти для объекта с помощью оператора *new* (его исходной версии или вашей собственной) автоматически вызывается конструктор объекта.

Функция *delete* получает указатель на область памяти, которую необходимо освободить. Затем она должна вернуть эту область памяти системе. При удалении объекта автоматически вызывается его деструктор.

Чтобы выделить память для массива объектов, а затем освободить ее, необходимо использовать следующие форматы операторов *new* и *delete*.

```
// Выделение памяти для массива объектов.
void *operator new[](size_t size)
{
  /* В случае невозможности выделить память генерируется исключение типа bad_alloc. Каждый конструктор вызывается автоматически. */
  return pointer_to_memory;
}

// Удаление массива объектов.
void operator delete[](void *p)
{
   /* Освобождается память, адресуемая указателем p. При этом автоматически вызывается деструктор для каждого элемента массива. */
}
```

При выделении памяти для массива автоматически вызывается конструктор каждого объекта, а при освобождении массива автоматически вызывается деструктор каждого объекта. Это значит, что для выполнения этих действий не нужно явным образом программировать их.

Операторы *new* и *delete*, как правило, перегружаются относительно класса. Ради простоты в следующем примере используется не новая схема распределения памяти, а перегруженные функции *new* и *delete*, которые просто вызывают С-ориентированные функции выделения памяти *malloc()* и *free()*. (В своем собственном приложении вы вольны реализовать любой метод выделения памяти.)

Чтобы перегрузить операторы *new* и *delete* для конкретного класса, достаточно сделать эти перегруженные операторные функции членами этого класса. В следующем примере программы операторы *new* и *delete* перегружаются для класса *three_d*. Эта перегрузка позволяет выделить память для объектов и массивов объектов, а затем освободить ее.

```
// Демонстрация перегруженных операторов new и delete.
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class three_d {
   int x, y, z; // 3-мерные координаты
  public:
   three_d() {
     x = y = z = 0;
     cout << "Создание объекта 0, 0, 0\n";
   }

   three_d(int i, int j, int k) {
     x = i;
     y = j;
     z = k;
     cout << "Создание объекта " << i << ", ";
     cout << j << ", " << k;
```

```
    cout << '\n';
  }

  ~three_d() { cout << "Разрушение объекта\n"; }

  void *operator new(size_t size);
  void *operator new[](size_t size);
  void operator delete(void *p);
  void operator delete[](void *p);

  void show();
};

// Перегрузка оператора new для класса three_d.
void *three_d::operator new(size_t size)
{
  void *p;

  cout <<"Выделение памяти для объекта класса three_d.\n";
  p = malloc(size);

  // Генерирование исключения в случае неудачного выделения памяти.
  if(!p) {
    bad_alloc ba;
    throw ba;
  }

  return p;
}

// Перегрузка оператора new для массива объектов типа three_d.
void *three_d::operator new[](size_t size)
{
  void *p;

  cout <<"Выделение памяти для массива three_d-объектов.";
  cout << "\n";

  // Генерирование исключения при неудаче.
  p = malloc(size);
  if(!p) {
    bad_alloc ba;
    throw ba;
  }
  return p;
}

// Перегрузка оператора delete для класса three_d.
```

```
void three_d::operator delete(void *p)
{
  cout << "Удаление объекта класса three_d.\n";
  free(p);
}

// Перегрузка оператора delete для массива объектов типа three_d.
void three_d::operator delete[](void *p)
{
  cout << "Удаление массива объектов типа three_d.\n";
  free(p);
}

// Отображение координат X, Y, Z.
void three_d::show( )
{
  cout << x << ", ";
  cout << y << ", ";
  cout << z << "\n";
}

int main( )
{
  three_d *p1, *p2;

  try {
    p1 = new three_d[3]; // выделение памяти для массива
    p2 = new three_d(5, 6, 7); // выделение памяти для объекта
  }

  catch (bad_alloc ba) {
    cout << "Ошибка при выделении памяти.\n";
    return 1;
  }

  p1[1].show( );
  p2->show( );

  delete [] p1; // удаление массива
  delete p2; // удаление объекта

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Выделение памяти для массива three_d-объектов.
```

```
Создание объекта 0, 0, 0
Создание объекта 0, 0, 0
Создание объекта 0, 0, 0
Выделение памяти для объекта класса three_d.
Создание объекта 5, 6, 7
0, 0, 0
5, 6, 7
Разрушение объекта
Разрушение объекта
Разрушение объекта
Удаление массива объектов типа three_d.
Разрушение объекта
Удаление объекта класса three_d.
```

Первые три сообщения *Создание объекта 0, 0, 0* выданы конструктором класса *three_d* (который не имеет параметров) при выделении памяти для трехэлементного массива. Как упоминалось выше, при выделении памяти для массива автоматически вызывается конструктор каждого элемента. *Сообщение Создание объекта 5, 6, 7* выдано конструктором класса *three_d* (который принимает три аргумента) при выделении памяти для одного объекта. Первые три сообщения *Разрушение объекта* выданы деструктором в результате удаления трехэлементного массива, поскольку при этом автоматически вызывался деструктор каждого элемента массива. Последнее сообщение *Разрушение объекта* выдано при удалении одного объекта класса *three_d*. Важно понимать, что, если операторы *new* и *delete* перегружены для конкретного класса, то в результате их использования для данных других типов будут задействованы оригинальные версии операторов *new* и *delete*. Это означает, что при добавлении в функцию main() следующей строки будет выполнена стандартная версия оператора *new*.

```
int *f = new int; // Используется стандартная версия оператора new.
```

И еще. Операторы *new* и *delete* можно перегружать глобально. Для этого достаточно объявить их операторные функции вне классов. В этом случае стандартные версии C++-операторов *new* и *delete* игнорируются вообще, и во всех запросах на выделение памяти используются их перегруженные версии. Безусловно, если вы при этом определите версию операторов *new* и *delete* для конкретного класса, то эти "классовые" версии будут применяться при выделении памяти (и ее освобождении) для объектов этого класса. Во всех же остальных случаях будут использоваться глобальные операторные функции.

Перегрузка nothrow-версии оператора new

Можно также создать перегруженные *nothrow*-версии операторов *new* и *delete*. Для этого используйте такие схемы.

```
// Перегрузка nothrow-версии оператора new.
void *operator new(size_t size, const nothrow_t &n)
{
  // Выделение памяти.
  if(success) return pointer_to_memory;
  else return 0;
}

// Перегрузка nothrow-версии оператора new для массива.
void *operator new[](size_t size, const nothrow_t &n)
{
```

```
{
  // Выделение памяти.
  if(success) return pointer_to_memory;
  else return 0;
}

// Перегрузка nothrow-версии оператора delete.
void operator delete(void *p, const nothrow_t &n)
{
  // Освобождение памяти.
}

// Перегрузка nothrow-версии оператора delete для массива.
void operator delete[](void *p, const nothrow_t &n)
{
  // Освобождение памяти.
}
```

Тип *nothrow_t* определяется в заголовке *<new>*. Параметр типа *nothrow_t* не используется. В качестве упражнения поэкспериментируйте с *nothrow*-версиями операторов *new* и *delete* самостоятельно.

Глава 18: C++-система ввода-вывода

С самого начала книги мы использовали C++-систему ввода-вывода, но не давали подробных пояснений по этому поводу. Поскольку C++-система ввода-вывода построена на иерархии классов, ее теорию и детали невозможно освоить, не рассмотрев сначала классы, наследование и механизм обработки исключений. Теперь настало время для подробного изучения C++-средств ввода-вывода.

В этой главе рассматриваются средства как консольного, так и файлового ввода-вывода. Необходимо сразу отметить, что C++-система ввода-вывода — довольно обширная тема, и здесь описаны лишь самые важные и часто применяемые средства. В частности, вы узнаете, как перегрузить операторы *"<<"* и *">>"* для ввода и вывода объектов созданных вами классов, а также как отформатировать выводимые данные и использовать манипуляторы ввода-вывода. Завершает главу рассмотрение средств файлового ввода-вывода.

Сравнение старой и новой C++-систем ввода-вывода

В настоящее время существуют две версии библиотеки объектно-ориентированного ввода-вывода, причем обе широко используются программистами: более старая, основанная на оригинальных спецификациях языка C++, и новая, определенная стандартом языка C++. Старая библиотека ввода-вывода поддерживается за счет заголовочного файла *<iostream.h>*, а новая — посредством заголовка *<iostream>*. Новая библиотека ввода-вывода, по сути, представляет собой обновленную и усовершенствованную версию старой. Основное различие между ними состоит в реализации, а не в том, как их нужно использовать.

С точки зрения программиста, есть два существенных различия между старой и новой C++-библиотеками ввода-вывода. Во-первых, новая библиотека содержит ряд дополнительных средств и определяет несколько новых типов данных. Таким образом, новую библиотеку ввода-вывода можно считать супермножеством старой. Практически все программы, написанные для старой библиотеки, успешно компилируются при использовании новой, не требуя внесения каких-либо значительных изменений. Во-вторых, старая библиотека ввода-вывода была определена в глобальном пространстве имен, а новая использует пространство имен *std*. (Вспомните, что пространство имен *std* используется всеми библиотеками стандарта C++.) Поскольку старая библиотека ввода-вывода уже устарела, в этой книге описывается только новая, но большая часть информации применима и к старой.

Потоки C++

Поток — *это последовательный логический интерфейс, который связан с физическим файлом*.

Принципиальным для понимания C++-системы ввода-вывода является то, что она опирается на понятие потока. *Поток* (*stream*) — это общий логический интерфейс с различными устройствами, составляющими компьютер. Поток либо синтезирует информацию, либо потребляет ее и связывается с любым физическим устройством с помощью C++-системы ввода-вывода. Характер поведения всех потоков одинаков, несмотря на различные физические устройства, с которыми они связываются. Поскольку потоки действуют одинаково, то практически ко всем типам устройств можно применить одни и те же функции и операторы ввода-вывода. Например, методы, используемые для записи данных на экран, также можно использовать для вывода их на принтер или для записи в дисковый файл.

В самой общей форме поток можно назвать логическим интерфейсом с файлом. C++-определение термина *"файл"* можно отнести к *дисковому файлу*, *экрану*, *клавиатуре*, *порту*, *файлу на магнитной ленте* и пр. Хотя файлы отличаются по форме и возможностям, все потоки одинаковы. Достоинство этого подхода (с точки зрения программиста) состоит в том, что одно устройство компьютера может *"выглядеть"* подобно любому другому. Это значит, что поток обеспечивает интерфейс, согласующийся *со всеми* устройствами.

Поток связывается с файлом при выполнении операции открытия файла, а отсоединяется от него с помощью операции закрытия.

Существует два типа потоков: *текстовый* и *двоичный*. *Текстовый поток* используется для ввода-вывода символов. При этом могут происходить некоторые преобразования символов. Например, при выводе символ новой строки может быть преобразован в последовательность символов: возврата каретки и перехода на новую строку. Поэтому может не быть взаимно-однозначного соответствия между тем, что посылается в поток, и тем, что в действительности записывается в файл. *Двоичный поток* можно использовать с данными любого типа, причем в этом случае никакого преобразования символов не выполняется, и между тем, что посылается в поток, и тем, что потом реально содержится в файле, существует взаимно-однозначное соответствие.

Текущая позиция — *это место в файле, с которого будет выполняться следующая операция доступа к файлу*.

Говоря о потоках, необходимо понимать, что вкладывается в понятие "текущей позиции". *Текущая позиция* — это место в файле, с которого будет выполняться следующая операция доступа к файлу. Например, если длина файла равна 100 байт, и известно, что уже прочитана половина этого файла, то следующая операция чтения произойдет на байте 50, который в данном случае и является текущей позицией.

Итак, в языке C++ механизм ввода-вывода функционирует с использованием логического интерфейса, именуемого потоком. Все потоки имеют аналогичные свойства, которые позволяют выполнять одинаковые функции ввода-вывода, независимо от того, с файлом какого типа существует связь. Под файлом понимается реальное физическое устройство, которое содержит данные. Если файлы различаются между собой, то потоки — нет. (Конечно, некоторые устройства могут не поддерживать все операции ввода-вывода, например операции с произвольной выборкой, поэтому и связанные с ними потоки тоже не будут поддерживать эти операции.)

Встроенные C++-потоки

В C++ содержится ряд встроенных потоков (*cin*, *cout*, *cerr* и *clog*), которые автоматически открываются, как только программа начинает выполняться. Как вы знаете, *cin* — это стандартный входной, а *cout* — стандартный выходной поток. Потоки *cerr* и *clog* (они предназначены для вывода информации об ошибках) также связаны со стандартным выводом данных. Разница между ними состоит в том, что поток *clog* буферизирован, а поток *cerr* — нет. Это означает, что любые выходные данные, посланные в поток *cerr*, будут немедленно выведены, а при использовании потока *clog* данные сначала записываются в буфер, и реальный их вывод происходит только тогда, когда буфер полностью заполняется. Обычно потоки *cerr* и *clog* используются для записи информации об отладке или ошибках.

В C++ также предусмотрены двухбайтовые (16-битовые) символьные версии стандартных

потоков, именуемые *wcin*, *wcout*, *wcerr* и *wclog*. Они предназначены для поддержки таких языков, как китайский, для представления которых требуются большие символьные наборы. В этой книге двухбайтовые стандартные потоки не используются.

По умолчанию стандартные C++-потоки связываются с консолью, но программным способом их можно перенаправить на другие устройства или файлы. Перенаправление может также выполнить операционная система.

Классы потоков

Как вы узнали в главе 2, C++-система ввода-вывода использует заголовок *<iostream>*, в котором для поддержки операций ввода-вывода определена довольно сложная иерархия классов. Эта иерархия начинается с системы шаблонных классов. Как отмечалось в главе 16, шаблонный класс определяет форму, не задавая в полном объеме данные, которые он должен обрабатывать. Имея шаблонный класс, можно создавать его конкретные экземпляры. Для библиотеки ввода-вывода стандарт C++ создает две специализации шаблонных классов: одну для 8-, а другую для 16-битовых ("широких") символов. В этой книге описываются классы только для 8-битовых символов, поскольку они используются гораздо чаще.

C++-система ввода-вывода построена на двух связанных, но различных иерархиях шаблонных классов. Первая выведена из класса низкоуровневого ввода-вывода *basic_streambuf*. Этот класс поддерживает базовые низкоуровневые операции ввода и вывода и обеспечивает поддержку для всей C++-системы ввода-вывода. Если вы не собираетесь заниматься программированием специализированных операций ввода-вывода, то вам вряд ли придется использовать напрямую класс *basic_streambuf*. Иерархия классов, с которой C++-программистам наверняка предстоит работать вплотную, выведена из класса *basic_ios*. Это — класс высокоуровневого ввода-вывода, который обеспечивает форматирование, контроль ошибок и предоставляет статусную информацию, связанную с потоками ввода-вывода. (Класс *basic_ios* выведен из класса *ios_base*, который определяет ряд нешаблонных свойств, используемых классом *basic_ios*.) Класс *basic_ios* используется в качестве базового для нескольких производных классов, включая классы *basic_istream*, *basic_ostream* и *basic_iostream*. Эти классы используются для создания потоков, предназначенных для *ввода* данных, *вывода* и *ввода-вывода* соответственно.

Как упоминалось выше, библиотека ввода-вывода создает две специализированные иерархии шаблонных классов: одну для 8-, а другую для 16-битовых символов. Ниже приводится список имен шаблонных классов и соответствующих им "символьных" версий.

Шаблонные классы	Символьные классы
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

В остальной части этой книги используются имена символьных классов, поскольку именно они применяются в программах. Те же имена используются и старой библиотекой ввода-вывода. Вот поэтому старая и новая библиотеки совместимы на уровне исходного кода.

И еще: класс *ios* содержит множество функций-членов и переменных, которые управляют основными операциями над потоками или отслеживают результаты их выполнения. Поэтому имя класса *ios* будет употребляться в этой книге довольно часто. И помните: если включить в программу заголовок *<iostream>*, она будет иметь доступ к этому важному классу.

Перегрузка операторов ввода-вывода

В примерах из предыдущих глав при необходимости выполнить операцию ввода или вывода данных, связанных с классом, создавались функции-члены, назначение которых и состояло лишь в том, чтобы ввести или вывести эти данные. Несмотря на то что в самом этом решении нет ничего неправильного, в C++ предусмотрен более удачный способ выполнения операций ввода-вывода *"классовых"* данных: путем перегрузки операторов ввода-вывода *"<<"* и *">>"*.

Оператор "<<" выводит информацию в поток, а оператор ">>" вводит информацию из потока.

В языке C++ оператор *"<<"* называется *оператором вывода* или *вставки*, поскольку он вставляет символы в поток. Аналогично оператор *">>"* называется *оператором ввода* или *извлечения*, поскольку он извлекает символы из потока.

Как вы знаете, операторы ввода-вывода уже перегружены (в заголовке *<iostream>*), чтобы они могли выполнять операции потокового ввода или вывода данных любых встроенных C++-типов. Здесь вы узнаете, как определить эти операторы для собственных классов.

Создание перегруженных операторов вывода

В качестве простого примера рассмотрим создание оператора вывода для следующей версии класса *three_d*.

```
class three_d {
  public:
    int x, y, z; // 3-мерные координаты
    three_d(int a, int b, int c) { x = a; y = b; z = c; }
};
```

Чтобы создать операторную функцию вывода для объектов типа *three_d*, необходимо перегрузить оператор *"<<"*. Вот один из возможных способов.

```
/* Отображение координат X, Y, Z (оператор вывода для класса three_d).
*/
ostream &operator<<(ostream &stream, three_d obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream; // возвращает параметр stream
}
```

Рассмотрим внимательно эту функцию, поскольку ее содержимое характерно для многих функций вывода данных. Во-первых, отметьте, что согласно объявлению она возвращает ссылку на объект типа *ostream*. Это позволяет несколько операторов вывода объединить в одном составном выражении. Затем обратите внимание на то, что эта функция имеет два параметра. Первый представляет собой ссылку на поток, который используется в левой части оператора. Вторым является объект, который стоит в правой части этого оператора. (При необходимости второй параметр также может иметь тип ссылки на объект.) Само тело функции состоит из инструкций вывода трех значений координат, содержащихся в объекте типа *three_d*, и инструкции возврата потока *stream*.

Перед вами короткая программа, в которой демонстрируется использование оператора вывода.

```
// Использование перегруженного оператора вывода.
#include <iostream>
```

```
using namespace std;

class three_d {
  public:
    int x, y, z; // 3-мерные координаты

    three_d(int a, int b, int c) { x = a; y = b; z = c; }
};

/* Отображение координат X, Y, Z (оператор вывода для класса three_d).
*/
ostream &operator<<(ostream &stream, three_d obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream; // возвращает параметр stream
}

int main()
{
  three_d a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);
  cout << a << b << c;
  return 0;
}
```

При выполнении эта программа возвращает следующие результаты:

```
1, 2, 3
3, 4, 5
5, 6, 7
```

Если удалить код, относящийся конкретно к классу *three_d*, останется *"скелет"*, подходящий для любой функции вывода данных.

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // код, относящийся к конкретному классу
  return stream; // возвращает параметр stream
}
```

Как уже отмечалось, для параметра *obj* разрешается использовать передачу по ссылке. В широком смысле конкретные действия функции вывода определяются программистом. Но если вы хотите следовать профессиональному стилю программирования, то ваша функция вывода должна все-таки выводить информацию. И потом, всегда нелишне убедиться в том, что она возвращает параметр *stream*.

Прежде чем переходить к следующему разделу, подумайте, почему функция вывода для класса *three_d* не была закодирована таким образом.

```
/* Версия ограниченного применения (использованию не подлежит).
```

```
*/
ostream &operator<<(ostream &stream, three_d obj)
{
  cout << obj.x << ", ";
  cout << obj.y << ", ";
  cout << obj.z << "\n";
  return stream; // возвращает параметр stream
}
```

В этой версии функции жестко закодирован поток *cout*. Это ограничивает круг ситуаций, в которых ее можно использовать. Помните, что оператор *"<<"* можно применить к любому потоку и что поток, который использован в *"<<"*-выражении, передается параметру *stream*. Следовательно, вы должны передавать функции поток, который корректно работает во всех случаях. Только так можно создать функцию вывода данных, которая подойдет для использования в любых выражениях ввода-вывода.

Использование функций-"друзей" для перегрузки операторов вывода

В предыдущей программе перегруженная функция вывода не была определена как член класса *three_d*. В действительности ни функция вывода, ни функция ввода не могут быть членами класса. Дело здесь вот в чем. Если операторная функция является членом класса, левый операнд (неявно передаваемый с помощью указателя *this*) должен быть объектом класса, который сгенерировал обращение к этой операторной функции. И это изменить нельзя. Однако при перегрузке операторов вывода левый операнд должен быть потоком, а правый — объектом класса, данные которого подлежат выводу. Следовательно, перегруженные операторы вывода не могут быть функциями-членами.

В связи с тем, что операторные функции вывода не должны быть членами класса, для которого они определяются, возникает серьезный вопрос: как перегруженный оператор вывода может получить доступ к закрытым элементам класса? В предыдущей программе переменные *x*, *y z* были определены как открытые, и поэтому оператор вывода без проблем мог получить к ним доступ. Но ведь сокрытие данных — важная часть объектно-ориентированного программирования, и требовать, чтобы все данные были открытыми, попросту нелогично. Однако существует решение и для этой проблемы: оператор вывода можно сделать *"другом"* класса. Если функция является *"другом"* некоторого класса, то она получает легальный доступ к его *private*-данным. Как можно объявить *"другом"* класса перегруженную функцию вывода, покажем на примере класса *three_d*.

```
// Использование "дружбы" для перегрузки оператора "<<"
#include <iostream>
using namespace std;

class three_d {
    int x, y, z; // 3-мерные координаты (теперь это private-члены)
  public:
    three_d(int a, int b, int c) { x = a; y = b; z = c; }

    friend ostream &operator<<(ostream &stream, three_d obj);
};

// Отображение координат X, Y, Z (оператор вывода для класса three_d).
ostream &operator<<(ostream &stream, three_d obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
```

```
  stream << obj.y <<  ,  ;
  stream << obj.z << "\n";
  return stream; // возвращает поток
}

int main()
{
  three_d a(1, 2, 3), b(3, 4, 5), c (5, 6, 7);
  cout << a << b << c;
  return 0;
}
```

Обратите внимание на то, что переменные *x*, *y* и *z* в этой версии программы являются закрытыми в классе *three_d*, тем не менее, операторная функция вывода обращается к ним напрямую. Вот где проявляется великая сила *"дружбы"*: объявляя операторные функции ввода и вывода *"друзьями"* класса, для которого они определяются, мы тем самым поддерживаем принцип инкапсуляции объектно-ориентированного программирования.

Перегрузка операторов ввода

Для перегрузки операторов ввода используйте тот же метод, который мы применяли при перегрузке оператора вывода. Например, следующий оператор ввода обеспечивает ввод трехмерных координат. Обратите внимание на то, что он также выводит соответствующее сообщение для пользователя.

```
/* Прием трехмерных координат (оператор ввода для класса three_d).
*/
istream &operator>>(istream &stream, three_d &obj)
{
  cout << "Введите координаты X, Y и Z:
  stream >> obj.x >> obj.y >> obj.z;
  return stream;
}
```

Оператор ввода должен возвращать ссылку на объект типа *istream*. Кроме того, первый параметр должен представлять собой ссылку на объект типа *istream*. Этот тип принадлежит потоку, указанному слева от оператора *">>"*. Второй параметр является ссылкой на переменную, которая принимает вводимое значение. Поскольку второй параметр — ссылка, он может быть модифицирован при вводе информации.

Общий формат оператора ввода имеет следующий вид.

```
istream &operator>>(istream &stream, object_type &obj)
{
  // код операторной функции ввода данных
  return stream;
}
```

Использование функции ввода данных для объектов типа *three_d* демонстрируется в следующей программе.

```
// Использование перегруженного оператора ввода.
#include <iostream>
using namespace std;
```

```
using namespace std;

class three_d {
   int x, y, z; // 3-мерные координаты
  public:
   three_d(int a, int b, int c) { x = a; y = b; z = c; }

   friend ostream &operator<<(ostream &stream, three_d obj);
   friend istream &operator>>(istream &stream, three_d &obj);
};

// Отображение координат X, Y, Z (оператор вывода для класса three_d).
ostream &operator<<(ostream &stream, three_d obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream; // возвращает параметр stream
}

// Прием трехмерных координат (оператор ввода для класса three_d).
istream &operator>>(istream &stream, three_d &obj)
{
  cout << "Введите координаты X, Y и Z: ";
  stream >> obj.x >> obj.y >> obj.z;
  return stream;
}

int main()
{
  three_d a(1, 2, 3);

  cout << a;
  cin >> a;
  cout << a;

  return 0;
}
```

Вот как выглядит один из возможных результатов выполнения этой программы.

```
1, 2, 3
Введите координаты X, Y и Z: 5 6 7
5, 6, 7
```

Подобно функциям вывода, функции ввода не должны быть членами класса, для обработки данных которого они предназначены. Они могут быть *"друзьями"* этого класса или просто независимыми функциями.

За исключением того, что функция ввода должна возвращать ссылку на объект типа *istream*, тело этой функции может содержать все, что вы считаете нужным в нее включить. Но

логичнее использовать операторы ввода все же по прямому назначению, т.е. для выполнения операций ввода.

Сравнение С- и С++-систем ввода-вывода

Как вы знаете, предшественник С++, язык С, оснащен одной из самых гибких (среди структурированных языков) и при этом очень мощных систем ввода-вывода. (Не будет преувеличением сказать, что среди всех известных структурированных языков С-система ввода-вывода не имеет себе равных.) Почему же тогда, спрашивается, в С++ определяется собственная система ввода-вывода, если в ней продублирована большая часть того, что содержится в С (имеется в виду мощный набор С-функций ввода-вывода)? Ответить на этот вопрос нетрудно. Дело в том, что С-система ввода-вывода не обеспечивает никакой поддержки для объектов, определяемых пользователем. Например, если создать в С такую структуру

```
struct my_struct {
  int count;
  char s [80];
  double balance;
} cust;
```

то существующую в С систему ввода-вывода невозможно настроить так, чтобы она могла выполнять операции ввода-вывода непосредственно над объектами типа *my_struct*. Но поскольку центром объектно-ориентированного программирования являются именно объекты, имеет смысл, чтобы в С++ функционировала такая система ввода-вывода, которую можно было бы динамически *"обучать"* обращению с любыми объектами, создаваемыми программистом. Именно поэтому для С++ и была изобретена новая объектно-ориентированная система ввода-вывода. Как вы уже могли убедиться, С++-подход к вводу-выводу позволяет перегружать операторы *"<<"* и *">>"*, чтобы они могли работать с классами, создаваемыми программистами.

И еще. Поскольку С++ является супермножеством языка С, все содержимое С-системы ввода-вывода включено в С++. (См. приложение А, в котором представлен обзор С-ориентированных функций ввода-вывода.) Поэтому при переводе С-программ на язык С++ вам не нужно изменять все инструкции ввода-вывода подряд. Работающие С-инструкции скомпилируются и будут успешно работать и в новой С++-среде. Просто вы должны учесть, что старая С-система ввода-вывода не обладает объектно-ориентированными возможностями.

Форматированный ввод-вывод данных

До сих пор при вводе или выводе информации в наших примерах программ действовали параметры форматирования, которые по умолчанию использует С++-система ввода-вывода. Но программист может сам управлять форматом представления данных, причем двумя способами. Первый способ предполагает использование функций-членов класса *ios*, а второй— функций специального типа, именуемых *манипуляторами* (manipulator). Мы же начнем освоение возможностей форматирования с функций-членов класса *ios*.

Форматирование данных с использованием функций-членов класса ios

В системе ввода-вывода С++ каждый поток связан с набором флагов форматирования, которые управляют процессом форматирования информации. В классе *ios* объявляется перечисление *fmtflags*, в котором определены следующие значения. (Точнее, эти значения определены в классе *ios_base*, который, как упоминалось выше, является базовым для класса *ios*.)

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	

fixed	oct	showpos	

Эти значения используются для установки или очистки флагов форматирования с помощью таких функций, как *setf()* и *unsetf()*. При использовании старого компилятора может оказаться, что он не определяет тип перечисления *fmtflags*. В этом случае флаги форматирования будут кодироваться как целочисленные *long*-значения.

Если флаг *skipws* установлен, то при потоковом вводе данных ведущие *"пробельные"* символы, или символы пропуска (т.е. пробелы, символы табуляции и новой строки), отбрасываются. Если же флаг *skipws* сброшен, пробельные символы не отбрасываются.

Если установлен флаг *left*, выводимые данные выравниваются по левому краю, а если установлен флаг *right* — по правому. Если установлен флаг *internal*, числовое значение дополняется пробелами, которыми заполняется поле между ним и знаком числа или символом основания системы счисления. Если ни один из этих флагов не установлен, результат выравнивается по правому краю по умолчанию.

По умолчанию числовые значения выводятся в десятичной системе счисления. Однако основание системы счисления можно изменить. Установка флага *oct* приведет к выводу результата в восьмеричном представлении, а установка флага *hex* — в шестнадцатеричном. Чтобы при отображении результата вернуться к десятичной системе счисления, достаточно установить флаг *dec*.

Установка флага *showbase* приводит к отображению обозначения основания системы счисления, в которой представляются числовые значения. Например, если используется шестнадцатеричное представление, то значение *1F* будет отображено как *0x1F*.

По умолчанию при использовании экспоненциального представления чисел отображается строчной вариант буквы *"e"*. Кроме того, при отображении шестнадцатеричного значения используется также строчная буква *"x"*. После установки флага *uppercase* отображается прописной вариант этих символов.

Установка флага *showpos* вызывает отображение ведущего знака *"плюс"* перед положительными значениями.

Установка флага *showpoint* приводит к отображению десятичной точки и хвостовых нулей для всех чисел с плавающей точкой — нужны они или нет.

После установки флага *scientific* числовые значения с плавающей точкой отображаются в экспоненциальном представлении. Если установлен флаг *fixed*, вещественные значения отображаются в обычном представлении. Если не установлен ни один из этих флагов, компилятор сам выбирает соответствующий метод представления.

При установленном флаге *unitbuf* содержимое буфера сбрасывается на диск после каждой операции вывода данных.

Если установлен флаг *boolalpha*, значения булева типа можно вводить или выводить, используя ключевые слова *true* и *false*.

Поскольку часто приходится обращаться к полям *oct*, *dec* и *hex*, на них допускается коллективная ссылка *ios::basefield*. Аналогично поля *left*, *right* и *internal* можно собирательно назвать *ios::adjustfield*. Наконец, поля *scientific* и *fixed* можно назвать *ios::floatfield*.

Чтобы установить флаги форматирования, обратитесь к функции setf().

Для установки любого флага используется функция *setf()*, которая является членом класса *ios*. Вот как выглядит ее формат.

```
fmtflags setf(fmtflags flags);
```

Эта функция возвращает значение предыдущих установок флагов форматирования и устанавливает их в соответствии со значением, заданным параметром *flags*. Например, чтобы установить флаг *showbase*, можно использовать эту инструкцию.

```
stream.setf(ios::showbase);
```

Здесь элемент *stream* означает поток, параметры форматирования которого вы хотите изменить. Обратите внимание на использование префикса *ios::* для уточнения принадлежности параметра *showbase*. Поскольку параметр *showbase* представляет собой перечислимую константу, определенную в классе *ios*, то при обращении к ней необходимо указывать имя класса *ios*. Этот принцип относится ко всем флагам форматирования. В следующей программе функция *setf()* используется для установки флагов *showpos* и *scientific*.

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

int main( )
{
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);
  cout << 123 << " " << 123.23 << " ";
  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
+123 +1.232300e+002
```

С помощью операции *ИЛИ* можно установить сразу несколько нужных флагов форматирования в одном вызове функции *setf()*. Например, предыдущую программу можно сократить, объединив по *ИЛИ* флаги *scientific* и *showpos*, поскольку в этом случае выполняется только одно обращение к функции *setf()*.

```
cout.setf(ios::scientific | ios::showpos);
```

Чтобы сбросить флаг, используйте функцию *unsetf()*, прототип которой выглядит так.

```
void unsetf(fmtflags flags);
```

Для очистки флагов форматирования используется функция unsetf().

В этом случае будут обнулены флаги, заданные параметром *flags*. (При этом все другие флаги остаются в прежнем состоянии.)

Чтобы получить текущие установки флагов форматирования, используйте функцию flags().

Для того чтобы узнать текущие установки флагов форматирования, воспользуйтесь функцией *flags()*, прототип которой имеет следующий вид.

```
fmtflags flags( );
```

Эта функция возвращает текущее значение флагов форматирования для вызывающего потока.

При использовании следующего формата вызова функции *flags()* устанавливаются значения флагов форматирования в соответствии с содержимым параметра *flags* и возвращаются их предыдущие значения.

```
fmtflags flags(fmtflags flags);
```

Чтобы понять, как работают функции *flags()* и *unsetf()*, рассмотрим следующую программу. Она включает функцию *showflags()*, которая отображает состояние флагов форматирования.

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

void showflags(ios::fmtflags f);

int main( )
```

```
{
  ios::fmtflags f;

  f = cout.flags();

  showflags(f);
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);

  f = cout.flags();
  showflags(f);

  cout.unsetf(ios:scientific);

  f = cout.flags();
  showflags(f);

  return 0;
}

void showflags(ios::fmtflags f)
{
  long i;
  for(i=0x4000; i; i=i>>1)
    if(i & f) cout << "1";
    else cout << "0";
  cout << "\n";
}
```

При выполнении эта программа отображает такие результаты. (Между этими и вашими результатами возможно расхождение, вызванное использованием различных компиляторов.)

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1

0 0 1 0 0 1 0 0 0 1 0 0 0 0 1

0 0 0 0 0 1 0 0 0 1 0 0 0 0 1
```

В предыдущей программе обратите внимание на то, что тип *fmtflags* указан с префиксом *ios ::*. Дело в том, что тип *fmtflags* определен в классе *ios*. В общем случае при использовании имени типа или перечислимой константы, определенной в некотором классе, необходимо указывать соответствующее имя вместе с именем класса.

Установка ширины поля, точности и символов заполнения

Помимо флагов форматирования можно также устанавливать ширину поля, символ заполнения и количество цифр после десятичной точки (точность). Для этого достаточно использовать следующие функции.

```
streamsize width(streamsize len);

char fill(char ch);

streamsize precision(streamsize num);
```

Функция *width()* возвращает текущую ширину поля и устанавливает новую равной значению

параметра *len*. Ширина поля, которая устанавливается по умолчанию, определяется количеством символов, необходимых для хранения данных в каждом конкретном случае. Функция *fill()* возвращает текущий символ заполнения (по умолчанию используется пробел) и устанавливает в качестве нового текущего символа заполнения значение, заданное параметром *ch*. Этот символ используется для дополнения результата символами, недостающими для достижения заданной ширины поля. Функция *precision()* возвращает текущее количество цифр, отображаемых после десятичной точки, и устанавливает новое текущее значение точности равным содержимому параметра *num*. (По умолчанию после десятичной точки отображается шесть цифр.) Тип *streamsize* определен как целочисленный тип.

Рассмотрим программу, которая демонстрирует использование этих трех функций.

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

int main()
{
  cout.setf(ios::showpos);
  cout.setf(ios::scientific);
  cout << 123 << " " << 123.23 << "\n";

  cout.precision(2); // Две цифры после десятичной точки.
  cout.width(10); // Всё поле состоит из 10 символов.
  cout << 123 << " ";
  cout.width(10); // Установка ширины поля равной 10.
  cout << 123.23 << "\n";

  cout.fill('#'); // Для заполнителя возьмем символ "#"
  cout.width(10); // и установим ширину поля равной 10.
  cout << 123 << " ";
  cout.width(10); // Установка ширины поля равной 10.
  cout << 123.23;

  return 0;
}
```

Эта программа генерирует такие результаты.

```
+123 +1.232300e+002
         +123 +1.23e+002
######+123 +1.23e+002
```

В некоторых реализациях необходимо устанавливать значение ширины поля перед выполнением каждой операции вывода. Поэтому функция *width()* в предыдущей программе вызывалась несколько раз.

В системе ввода-вывода C++ определены и перегруженные версии функций *width()*, *precision()* и *fill()*, которые не изменяют текущие значения соответствующих параметров форматирования и используются только для их получения. Вот как выглядят их прототипы,

```
char fill();
streamsize width();
streamsize precision();
```

Манипуляторы позволяют встраивать инструкции форматирования в выражение ввода-вывода.

В C++-системе ввода-вывода предусмотрен и второй способ изменения параметров форматирования, связанных с потоком. Он реализуется с помощью специальных функций, называемых *манипуляторами*, которые можно включать в выражение ввода-вывода. Стандартные манипуляторы описаны в табл. 18.1.

Таблица 18.1. Стандартные C++-манипуляторы ввода-вывода

Манипулятор	Назначение	Функция
`boolalpha`	Устанавливает флаг `boolalpha`	Ввод-вывод
`dec`	Устанавливает флаг `dec`	Ввод-вывод
`endl`	Выводит символ новой строки и "сбрасывает" поток, т.е. переписывает содержимое буфера, связанного с потоком, на соответствующее устройство	Вывод
`ends`	Вставляет в поток нулевой символ (`'\0'`)	Вывод
`fixed`	Устанавливает флаг `fixed`	Вывод
`flush`	"Сбрасывает" поток	Вывод
`hex`	Устанавливает флаг `hex`	Ввод-вывод
`internal`	Устанавливает флаг `internal`	Вывод
`left`	Устанавливает флаг `left`	Вывод
`noboolalpha`	Обнуляет флаг `boolalpha`	Ввод-вывод
`noshowbase`	Обнуляет флаг `showbase`	Вывод
`noshowpoint`	Обнуляет флаг `showpoint`	Вывод
`noshowpos`	Обнуляет флаг `showpos`	Вывод
`noskipws`	Обнуляет флаг `skipws`	Ввод

Окончание табл. 18.1

Манипулятор	Назначение	Функция
`nounitbuf`	Обнуляет флаг `unitbuf`	Вывод
`nouppercase`	Обнуляет флаг `uppercase`	Вывод
`oct`	Устанавливает флаг `oct`	Ввод-вывод
`resetiosflags(` `fmtflags f)`	Обнуляет флаги, заданные в параметре *f*	Ввод-вывод
`right`	Устанавливает флаг `right`	Вывод
`scientific`	Устанавливает флаг `scientific`	Вывод
`setbase(` `int base)`	Устанавливает основание системы счисления равной значению *base*	Вывод
`setfill(` `int ch)`	Устанавливает символ-заполнитель равным значению параметра *ch*	Вывод
`setiosflags(` `fmtflags f)`	Устанавливает флаги, заданные в параметре *f*	Ввод-вывод
`setprecision(` `int p)`	Устанавливает количество цифр точности (после десятичной точки)	Вывод
`setw(int w)`	Устанавливает ширину поля равной значению параметра *w*	Вывод
`showbase`	Устанавливает флаг `showbase`	Вывод

showpoint	Устанавливает флаг showpoint	Вывод
showpos	Устанавливает флаг showpos	Вывод
skipws	Устанавливает флаг skipws	Ввод
unitbuf	Устанавливает флаг unitbuf	Вывод
uppercase	Устанавливает флаг uppercase	Вывод
ws	Пропускает ведущие "пробельные" символы	Ввод

При использовании манипуляторов, которые принимают аргументы, необходимо включить в программу заголовок *<iomanip>*.

Манипулятор используется как часть выражения ввода-вывода. Вот пример программы, в которой показано, как с помощью манипуляторов можно управлять форматированием выводимых данных.

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

int main()
{
  cout << setprecision (2) << 1000.243 << endl;
  cout << setw(20) << "Всем привет! ";
  return 0;
}
```

Результаты выполнения этой программы таковы.

```
1e+003
Всем привет!
```

Обратите внимание на то, как используются манипуляторы в цепочке операций ввода-вывода. Кроме того, отметьте, что, если манипулятор вызывается без аргументов (как, например, манипулятор *endl* в нашей программе), то его имя указывается без пары круглых скобок.

В следующей программе используется манипулятор *setiosflags()* для установки флагов *scientific* и *showpos*.

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

int main()
{
  cout << setiosflags(ios::showpos);
  cout << setiosflags(ios::scientific);
  cout << 123 << " " << 123.23;
  return 0;
}
```

Вот результаты выполнения данной программы.

```
+123 +1.232300e+002
```

А в этой программе демонстрируется использование манипулятора *ws*, который пропускает ведущие *"пробельные"* символы при вводе строки в массив *s*:

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

int main()
{
  char s[80];
  cin >> ws >> s;
  cout << s;
  return 0;
}
```

Создание собственных манипуляторных функций

Программист может создавать собственные манипуляторные функции. Существует два типа манипуляторных функций: принимающие и не принимающие аргументы. Для создания параметризованных манипуляторов используются методы, рассмотрение которых выходит за рамки этой книги. Однако создание манипуляторов, которые не имеют параметров, не вызывает особых трудностей.

Все манипуляторные функции вывода данных без параметров имеют следующую структуру.

```
ostream &manip_name(ostream &stream)
{
  // код манипуляторной функции
  return stream;
}
```

Здесь элемент *manip_name* означает имя манипулятора. Важно понимать, что, несмотря на то, что манипулятор принимает в качестве единственного аргумента указатель на поток, который он обрабатывает, при использовании манипулятора в результирующем выражении ввода-вывода аргументы не указываются вообще.

В следующей программе создается манипулятор *setup()*, который устанавливает флаг выравнивания по левому краю, ширину поля равной 10 и задает в качестве заполняющего символа знак доллара.

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

ostream &setup(ostream &stream)
{
  stream.setf(ios::left);
  stream << setw(10) << setfill ('$');
  return stream;
}

int main()
{
  cout << 10 << " " << setup << 10;
  return 0;
```

```
}
```

Собственные манипуляторы полезны по двум причинам. Во-первых, иногда возникает необходимость выполнять операции ввода-вывода с использованием устройства, к которому ни один из встроенных манипуляторов не применяется (например, плоттер). В этом случае создание собственных манипуляторов сделает вывод данных на это устройство более удобным. Во-вторых, может оказаться, что у вас в программе некоторая последовательность инструкций повторяется несколько раз. И тогда вы можете объединить эти операции в один манипулятор, как показано в предыдущей программе.

Все манипуляторные функции ввода данных без параметров имеют следующую структуру.

```
istream &manip_name(istream &stream)
{
  // код манипуляторной функции
  return stream;
}
```

Например, в следующей программе создается манипулятор *prompt()*. Он настраивает входной поток на прием данных в шестнадцатеричном представлении и отображает для пользователя наводящее сообщение.

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

istream &prompt(istream &stream)
{
  cin >> hex;
  cout << "Введите число в шестнадцатеричном формате: ";
  return stream;
}

int main()
{
  int i;

  cin >> prompt >> i;
  cout << i;

  return 0;
}
```

Помните: очень важно, чтобы ваш манипулятор возвращал потоковый объект (элемент *stream*). В противном случае этот манипулятор нельзя будет использовать в составном выражении ввода или вывода.

Файловый ввод-вывод

В C++-системе ввода-вывода также предусмотрены средства для выполнения соответствующих операций с использованием файлов. Файловые операции ввода-вывода можно реализовать после включения в программу заголовка *<fstream>*, в котором определены все необходимые для этого классы и значения.

Как открыть и закрыть файл

В C++ файл открывается путем связывания его с потоком. Как вы знаете, существуют потоки трех типов: *ввода*, *вывода* и *ввода-вывода*. Чтобы открыть входной поток, необходимо объявить потоковый объект типа *ifstream*. Для открытия выходного потока нужно объявить поток класса *ofstream*. Поток, который предполагается использовать для операций как ввода, так и вывода, должен быть объявлен как объект класса *fstream*. Например, при выполнении следующего фрагмента кода будет создан входной поток, выходной и поток, позволяющий выполнение операций в обоих направлениях.

```
ifstream in; // входной поток

ofstream out; // выходной поток

fstream both; // поток ввода-вывода
```

Чтобы открыть файл, используйте функцию open().

Создав поток, его нужно связать с файлом. Это можно сделать с помощью функции *open()*, причем в каждом из трех потоковых классов есть своя функция-член *open()*. Представим их прототипы.

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);

void ofstream::open(const char *filename, ios::openmode mode = ios::out
| ios::trunc);

void fstream::open(const char * filename, ios::openmode mode = ios::in |
ios::out);
```

Здесь элемент *filename* означает имя файла, которое может включать спецификатор пути. Элемент *mode* определяет способ открытия файла. Он должен принимать одно или несколько значений перечисления *openmode*, которое определено в классе *ios*.

```
ios::app

ios::ate

ios::rbinary

ios::in

ios::out

ios::trunc
```

Несколько значений перечисления *openmode* можно объединять посредством логического сложения (*ИЛИ*).

На заметку. *Параметр mode для функции fstream::open() может не устанавливаться по умолчанию равным значению in | out (это зависит от используемого компилятора). Поэтому при необходимости этот параметр вам придется задавать в явном виде*.

Включение значения *ios::app* в параметр *mode* обеспечит присоединение к концу файла всех выводимых данных. Это значение можно применять только к файлам, открытым для вывода данных. При открытии файла с использованием значения *ios::ate* поиск будет начинаться с конца файла. Несмотря на это, операции ввода-вывода могут по-прежнему выполняться по всему файлу.

Значение *ios::in* говорит о том, что данный файл открывается для ввода данных, а значение *ios::out* обеспечивает открытие файла для вывода данных.

Значение *ios::binary* позволяет открыть файл в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. Как упоминалось выше, в текстовом режиме могут происходить некоторые преобразования символов (например, последовательность, состоящая из символов возврата каретки и перехода на новую строку, может быть преобразована в символ новой строки). При открытии файла в двоичном режиме никакого преобразования символов не выполняется. Следует иметь в виду, любой файл, содержащий форматированный текст или еще необработанные данные, можно открыть как в двоичном,

так и в текстовом режиме. Единственное различие между этими режимами состоит в преобразовании (или нет) символов.

Использование значения *ios::trunc* приводит к разрушению содержимого файла, имя которого совпадает с параметром *filename*, а сам этот файл усекается до нулевой длины. При создании выходного потока типа *ofstream* любой существующий файл с именем *filename* автоматически усекается до нулевой длины.

При выполнении следующего фрагмента кода открывается обычный выходной файл.

```
ofstream out;
out.open("тест");
```

Поскольку параметр *mode* функции *open()* по умолчанию устанавливается равным значению, соответствующему типу открываемого потока, в предыдущем примере вообще нет необходимости задавать его значение.

Не открытый в результате неудачного выполнения функции *open()* поток при использовании в булевом выражении устанавливается равным значению ЛОЖЬ. Этот факт может служить для подтверждения успешного открытия файла, например, с помощью такой if-инструкции.

```
if(!mystream) {
  cout << "Не удается открыть файл.\n";
  // обработка ошибки
}
```

Прежде чем делать попытку получения доступа к файлу, следует всегда проверять результат вызова функции *open()*.

Можно также проверить факт успешного открытия файла с помощью функции *is_open()*, которая является членом классов *fstream*, *ifstream* и *ofstream*. Вот ее прототип,

```
bool is_open();
```

Эта функция возвращает значение ИСТИНА, если поток связан с открытым файлом, и ЛОЖЬ — в противном случае. Например, используя следующий код, можно узнать, открыт ли в данный момент потоковый объект *mystream*.

```
if(!mystream.is_open()) {
  cout << "Файл не открыт.\n";
  // ...
}
```

Хотя вполне корректно использовать функцию *open()* для открытия файла, в большинстве случаев это делается по-другому, поскольку классы *ifstream*, *ofstream* и *fstream* включают конструкторы, которые автоматически открывают заданный файл. Параметры у этих конструкторов и их значения (действующие по умолчанию) совпадают с параметрами и соответствующими значениями функции *open()*. Поэтому чаще всего файл открывается так, как показано в следующем примере,

```
ifstream mystream("myfile"); // файл открывается для ввода
```

Если по какой-то причине файл открыть невозможно, потоковая переменная, связываемая с этим файлом, устанавливается равной значению ЛОЖЬ.

Чтобы закрыть файл, вызовите функцию close().

Чтобы закрыть файл, используйте функцию-член *close()*. Например, чтобы закрыть файл, связанный с потоковым объектом *mystream*, используйте такую инструкцию,

```
mystream.close();
```

Функция *close()* не имеет параметров и не возвращает никакого значения.

Чтение и запись текстовых файлов

Чтение и запись текстовых файлов

Проще всего считывать данные из текстового файла или записывать их в него с помощью операторов "<<" и ">>". Например, в следующей программе выполняется запись в файл *test* целого числа, значения с плавающей точкой и строки.

```
// Запись данных в файл.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  ofstream out("test");

  if(!out) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  out << 10 << " " << 123.23 << "\n";
  out << "Это короткий текстовый файл.";

  out.close();

  return 0;
}
```

Следующая программа считывает целое число, float-значение, символ и строку из файла, созданного при выполнении предыдущей программой.

```
// Считывание данных из файла.
#include <iostream>
#include <fstream>
 using namespace std;

int main()
{
  char ch;
  int i;
  float f;
  char str[80];

  ifstream in("test");
  if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  in >> i;
  in >> f;
  in >> ch;
```

```
  in >> str;

  cout << i << " " << f << " " << ch << "\n";
  cout << str;

  in.close();
  return 0;
}
```

Следует иметь в виду, что при использовании оператора *">>"* для считывания данных из текстовых файлов происходит преобразование некоторых символов. Например, *"пробельные"* символы опускаются. Если необходимо предотвратить какие бы то ни было преобразования символов, откройте файл в двоичном режиме доступа. Кроме того, помните, что при использовании оператора *">>"* для считывания строки ввод прекращается при обнаружении первого *"пробельного"* символа.

Неформатированный ввод-вывод данных в двоичном режиме

Форматированные текстовые файлы (подобные тем, которые использовались в предыдущих примерах) полезны во многих ситуациях, но они не обладают гибкостью неформатированных двоичных файлов. Поэтому C++ поддерживает ряд функций файлового ввода-вывода в двоичном режиме, которые могут выполнять операции без форматирования данных.

Для выполнения двоичных операций файлового ввода-вывода необходимо открыть файл с использованием спецификатора режима *ios::binary*. Необходимо отметить, что функции обработки неформатированных файлов могут работать с файлами, открытыми в текстовом режиме доступа, но при этом могут иметь место преобразования символов, которые сводят на нет основную цель выполнения двоичных файловых операций.

Функция get() считывает символ из файла, а функция put() записывает символ в файл.

В общем случае существует два способа записи неформатированных двоичных данных в файл и считывания их из файла. Первый состоит в использовании функции-члена *put()* (для записи байта в файл) и функции-члена *get()* (для считывания байта из файла). Второй способ предполагает применение *"блочных"* C++-функций ввода-вывода *read()* и *write()*. Рассмотрим каждый способ в отдельности.

Использование функций get() и put()

Функции *get()* и *put()* имеют множество форматов, но чаще всего используются следующие их версии:

```
istream &get(char &ch);
ostream &put(char ch);
```

Функция *get()* считывает один символ из соответствующего потока и помещает его значение в переменную *ch*. Она возвращает ссылку на поток, связанный с предварительно открытым файлом. При достижении конца этого файла значение ссылки станет равным нулю. Функция *put()* записывает символ *ch* в поток и возвращает ссылку на этот поток.

При выполнении следующей программы на экран будет выведено содержимое любого заданного файла. Здесь используется функция *get()*.

```
/* Отображение содержимого файла с помощью функции get().
*/
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Применение: имя_программы <имя_файла>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  while(in) {
     /* При достижении конца файла потоковый объект in примет значение
false. */
    in.get(ch);
    if(in) cout << ch;
  }

  in.close();
  return 0;
}
```

При достижении конца файла потоковый объект *in* примет значение *ЛОЖЬ*, которое остановит выполнение цикла *while*.

Существует более короткий вариант цикла, предназначенного для считывания и отображения содержимого файла.

```
while(in.get(ch)) cout << ch;
```

Этот вариант также имеет право на существование, поскольку функция *get()* возвращает потоковый объект *in*, который при достижении конца файла примет значение *false*.

В следующей программе для записи строки в файл используется функция *put()*.

```
/* Использование функции put() для записи строки в файл.
*/
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char *p = "Всем привет!";

  ofstream out("test", ios::out | ios::binary);
  if(!out) {
    cout << "Не удается открыть файл.\n";
    return 1;
```

```
    return 1;
  }

  while(*p) out.put(*p++);

  out.close();
  return 0;
}
```

Считывание и запись в файл блоков данных

Чтобы считывать и записывать в файл блоки двоичных данных, используйте функции-члены *read()* и *write()*. Их прототипы имеют следующий вид.

```
istream &read(char *buf, streamsize num);

ostream &write(const char *buf, int streamsize num);
```

Функция *read()* считывает *num* байт данных из связанного с файлом потока и помещает их в буфер, адресуемый параметром *buf*. Функция *write()* записывает *num* байт данных в связанный с файлом поток из буфера, адресуемого параметром *buf*. Как упоминалось выше, тип *streamsize* определен как некоторая разновидность целочисленного типа. Он позволяет хранить самое большое количество байтов, которое может быть передано в процессе любой операции ввода-вывода.

Функция read() вводит блок данных, а функция write() выводит его.

При выполнении следующей программы сначала в файл записывается массив целых чисел, а затем он же считывается из файла.

```
// Использование функций read() и write().
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  int n[5] = {1, 2, 3, 4, 5};
  register int i;

  ofstream out("test", ios::out | ios::binary);
  if(!out) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  out.write((char *) &n, sizeof n);
  out.close();

  for(i=0; i<5; i++) // очищаем массив
    n[i] = 0;

  ifstream in ("test", ios::in | ios::binary);
  if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
```

```
  }

  in.read((char *) &n, sizeof n);
  for(i=0; i<5; i++) // Отображаем значения, считанные из файла.
    cout << n[i] << " ";

  in.close();
  return 0;
}
```

Обратите внимание на то, что в инструкциях обращения к функциям *read()* и *write()* выполняются операции приведения типа, которые обязательны при использовании буфера, определенного не в виде символьного массива.

Функция gcount() возвращает количество символов, считанных при выполнении последней операции ввода данных.

Если конец файла будет достигнут до того, как будет считано *num* символов, функция *read()* просто прекратит выполнение, а буфер будет содержать столько символов, сколько удалось считать до этого момента. Точное количество считанных символов можно узнать с помощью еще одной функции-члена *gcount()*, которая имеет такой прототип.

```
streamsize gcount( );
```

Функция *gcount()* возвращает количество символов, считанных в процессе выполнения последней операции ввода данных.

Обнаружение конца файла

Обнаружить конец файла можно с помощью функции-члена *eof()*, которая имеет такой прототип.

```
bool eof( );
```

Эта функция возвращает значение *true* при достижении конца файла; в противном случае она возвращает значение *false*.

Функция eof() позволяет обнаружить конец файла.

В следующей программе для вывода на экран содержимого файла используется функция *eof()*.

```
/* Обнаружение конца файла с помощью функции eof( ).
*/
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;

  if(argc!=2) {
    cout << "Применение: имя_программы <имя_файла>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
```

```
  if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  while(!in.eof()) {
    // использование функции eof()
    in.get(ch);
    if( !in.eof()) cout << ch;
  }

  in.close();
  return 0;
}
```

Пример сравнения файлов

Следующая программа иллюстрирует мощь и простоту применения в C++ файловой системы. Здесь сравниваются два файла с помощью функций двоичного ввода-вывода *read()*, *eof()* и *gcount()*. Программа сначала открывает сравниваемые файлы для выполнения двоичных операций (чтобы не допустить преобразования символов). Затем из каждого файла по очереди считываются блоки информации в соответствующие буферы и сравнивается их содержимое. Поскольку объем считанных данных может быть меньше размера буфера, в программе используется функция *gcount()*, которая точно определяет количество считанных в буфер байтов. Нетрудно убедиться в том, что при использовании файловых C++-функций для выполнения этих операций потребовалась совсем небольшая по размеру программа.

```
// Сравнение файлов.
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  register int i;

  unsigned char buf1[1024], buf2[1024];
  if(argc!=3) {
    cout << "Применение: имя_программы <имя_файла1> "<< "<имя_файла2>\n";
    return 1;
  }

  ifstream f1(argv[1], ios::in | ios::binary);
  if(!f1) {
    cout << "Не удается открыть первый файл.\n";
    return 1;
  }

  ifstream f2(argv[2], ios::in | ios::binary);
  if(!f2) {
```

```
  if(!f2) {
    cout << "Не удается открыть второй файл.\n";
    return 1;
  }

  cout << "Сравнение файлов ...\n";

  do {
    f1.read((char *) buf1, sizeof buf1);
    f2.read((char *) buf2, sizeof buf2);

    if(f1.gcount() != f2.gcount()) {
      cout << "Файлы имеют разные размеры.\n";
      f1.close();
      f2.close();
      return 0;
    }

    // Сравнение содержимого буферов.
    for(i=0; i<f1.gcount(); i++)
      if(buf1[i] != buf2[i]) {
        cout << "Файлы различны.\n";
        f1.close();
        f2.close();
        return 0;
      }
  }while(!f1.eof() && !f2.eof());
  cout << "Файлы одинаковы.\n";

  f1.close();
  f2.close();

  return 0;
}
```

Проведите эксперимент. Размер буфера в этой программе жестко установлен равным *1024*. В качестве упражнения замените это значение *const*-переменной и опробуйте другие размеры буферов. Определите оптимальный размер буфера для своей операционной среды.

Использование других функций двоичного ввода-вывода

Помимо приведенного выше формата использования функции *get()* существуют и другие ее перегруженные версии. Приведем прототипы для трех из них, которые используются чаще всего.

```
istream &get(char *buf, streamsize num);

istream &get(char *buf, streamsize num, char delim);

int get();
```

Первая версия позволяет считывать символы в массив, заданный параметром *buf*, до тех пор, пока либо не будет считано *num-1* символов, либо не встретится символ новой строки, либо не будет достигнут конец файла. После выполнения функции *get()* массив, адресуемый параметром *buf*, будет иметь завершающий нуль-символ. Символ новой строки, если таковой обнаружится во входном потоке, не извлекается. Он остается там до тех пор, пока не выполнится следующая операция ввода-вывода.

Вторая версия предназначена для считывания символов в массив, адресуемый параметром *buf*, до тех пор, пока либо не будет считано *num-1* символов, либо не обнаружится символ, заданный параметром *delim*, либо не будет достигнут конец файла. После выполнения функции *get()* массив, адресуемый параметром *buf*, будет иметь завершающий нуль-символ. Символ-разделитель (заданный параметром *delim*), если таковой обнаружится во входном потоке, не извлекается. Он остается там до тех пор, пока не выполнится следующая операция ввода-вывода.

Третья перегруженная версия функции *get()* возвращает из потока следующий символ. Он содержится в младшем байте значения, возвращаемого функцией. Следовательно, значение, возвращаемое функцией *get()*, можно присвоить переменной типа *char*. При достижении конца файла эта функция возвращает значение *EOF*, которое определено в заголовке *<iostream>*.

Функцию *get()* полезно использовать для считывания строк, содержащих пробелы. Как вы знаете, если для считывания строки используется оператор *">>"*, процесс ввода останавливается при обнаружении первого же пробельного символа. Это делает оператор *">>"* бесполезным для считывания строк, содержащих пробелы. Но эту проблему, как показано в следующей программе, можно обойти с помощью функции *get(buf,num)*.

```
/* Использование функции get() для считывания строк содержащих пробелы.
*/
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
  char str[80];
  cout << "Введите имя: ";
    cin.get (str, 79);
  cout << str << '\n';

  return 0;
}
```

Здесь в качестве символа-разделителя при считывании строки с помощью функции *get()* используется символ новой строки. Это делает поведение функции *get()* во многом сходным с поведением стандартной функции *gets()*. Однако преимущество функции *get()* состоит в том, что она позволяет предотвратить возможный выход за границы массива, который принимает вводимые пользователем символы, поскольку в программе задано максимальное количество считываемых символов. Это делает функцию *get()* гораздо безопаснее функции *gets()*.

Рассмотрим еще одну функцию, которая позволяет вводить данные. Речь идет о функции *getline()*, которая является членом каждого потокового класса, предназначенного для ввода информации. Вот как выглядят прототипы версий этой функции,

```
istream &getline(char *buf, streamsize num);
```

```
istream &getline(char *buf, streamsize num, char delim);
```

Функция getline() представляет собой еще один способ ввода данных.

При использовании первой версии символы считываются в массив, адресуемый указателем *buf*, до тех пор, пока либо не будет считано *num-1* символов, либо не встретится символ новой строки, либо не будет достигнут конец файла. После выполнения функции *getline()* массив, адресуемый параметром *buf*, будет иметь завершающий нуль-символ. Символ новой строки, если таковой обнаружится во входном потоке, при этом извлекается, но не помещается в массив *buf*.

Вторая версия предназначена для считывания символов в массив, адресуемый параметром *buf*, до тех пор, пока либо не будет считано *num-1* символов, либо не обнаружится символ, заданный параметром *delim*, либо не будет достигнут конец файла. После выполнения функции *getline()* массив, адресуемый параметром *buf*, будет иметь завершающий нуль-символ. Символ-разделитель (заданный параметром *delim*), если таковой обнаружится во входном потоке, извлекается, но не помещается в массив *buf*.

Как видите, эти две версии функции *getline()* практически идентичны версиям *get (buf, num)* и *get (buf, num, delim)* функции *get()*. Обе считывают символы из входного потока и помещают их в массив, адресуемый параметром *buf*, до тех пор, пока либо не будет считано *num-1* символов, либо не обнаружится символ, заданный параметром *delim*. Различие между функциями *get()* и *getline()* состоит в том, что функция *getline()* считывает и удаляет символ-разделитель из входного потока, а функция *get()* этого не делает.

Функция peek() считывает следующий символ из входного потока, не удаляя его.

Следующий символ из входного потока можно получить и не удалять его из потока с помощью функции *peek()*. Вот как выглядит ее прототип.

```
int peek( );
```

Функция *peek()* возвращает следующий символ потока, или значение *EOF*, если достигнут конец файла. Считанный символ возвращается в младшем байте значения, возвращаемого функцией. Поэтому значение, возвращаемое функцией *peek()*, можно присвоить переменной типа *char*.

Функция putback() возвращает считанный символ во входной поток.

Последний символ, считанный из потока, можно вернуть в поток, используя функцию *putback()*. Ее прототип выглядит так.

```
istream &putback(char c);
```

Здесь параметр *c* содержит символ, считанный из потока последним.

Функция flush() сбрасывает на диск содержимое файловых буферов.

При выводе данных немедленной их записи на физическое устройство, связанное с потоком, не происходит. Подлежащая выводу информация накапливается во внутреннем буфере до тех пор, пока этот буфер не заполнится целиком. И только тогда его содержимое переписывается на диск. Однако существует возможность немедленной перезаписи на диск хранимой в буфере информации, не дожидаясь его заполнения. Это средство состоит в вызове функции *flush()*. Ее прототип имеет такой вид.

```
ostream &flush( );
```

К вызовам функции *flush()* следует прибегать в случае, если программа предназначена для выполнения в неблагоприятных средах (для которых характерны частые отключения электричества, например).

Произвольный доступ

До сих пор мы использовали файлы, доступ к содержимому которых был организован строго последовательно, байт за байтом. Но в C++ также можно получать доступ к файлу в произвольном порядке. В этом случае необходимо использовать функции *seekg()* и *seekp()*. Вот их прототипы.

```
istream &seekg(off_type offset, seekdir origin);

ostream &seekp(off_type offset, seekdir origin);
```

Используемый здесь целочисленный тип *off_type* (он определен в классе ios) позволяет хранить самое большое допустимое значение, которое может иметь параметр *offset*. Тип *seekdir* определен как перечисление, которое имеет следующие значения.

Значение	Описание
ios::beg	Начало файла
ios::cur	Текущая позиция
ios::end	Конец файла

Функция seekg() перемещает указатель, "отвечающий" за ввод данных, а функция seekp() — указатель, "отвечающий" за вывод.

В C++-системе ввода-вывода предусмотрена возможность управления двумя указателями, связанными с файлом. Эти так называемые *cin*- и *put*-указатели определяют, в каком месте файла должна выполниться следующая операция ввода и вывода соответственно. При каждом выполнении операции ввода или вывода соответствующий указатель автоматически перемещается в указанную позицию. Используя функции *seekg()* и *seekp()*, можно получать доступ к файлу в произвольном порядке.

Функция *seekg()* перемещает текущий *get*-указатель соответствующего файла на *offset* байт относительно позиции, заданной параметром *origin*. Функция *seekp()* перемещает текущий *put*-указатель соответствующего файла на *offset* байт относительно позиции, заданной параметром *origin*.

В общем случае произвольный доступ для операций ввода-вывода должен выполняться только для файлов, открытых в двоичном режиме. Преобразования символов, которые могут происходить в текстовых файлах, могут привести к тому, что запрашиваемая позиция файла не будет соответствовать его реальному содержимому.

В следующей программе демонстрируется использование функции *seekp()*. Она позволяет задать имя файла в командной строке, а за ним — конкретный байт, который нужно в нем изменить. Программа затем записывает в указанную позицию символ *"X"*. Обратите внимание на то, что обрабатываемый файл должен быть открыт для выполнения операций чтения-записи.

```
/* Демонстрация произвольного доступа к файлу.
*/
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  if(argc!=3) {
    cout << "Применение: имя_программы " << "<имя_файла> <байт>\n";
    return 1;
  }

  fstream out(argv[1], ios::in | ios::out | ios::binary);
  if(!out) {
    cout << "Не удается открыть файл.\n";
```

```
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  out.seekp(atoi(argv[2]), ios::beg);
  out.put('X');
  out.close();

  return 0;
}
```

В следующей программе показано использование функции *seekg()*. Она отображает содержимое файла, начиная с позиции, заданной в командной строке.

```
/* Отображение содержимого файла с заданной стартовой позиции.
*/
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
  char ch;
  if(argc!=3) {
        cout << "Применение: имя_программы "<< "<имя_файла> <стартовая_позиция>\n";
    return 1;
  }

  ifstream in(argv[1], ios::in | ios::binary);
  if(!in) {
    cout << "Не удается открыть файл.\n";
    return 1;
  }

  in.seekg(atoi(argv[2]), ios::beg);
  while(in.get (ch)) cout << ch;

  return 0;
}
```

Функция tellg() возвращает текущую позицию get-указателя, а функция tellp() — текущую позицию put-указателя.

Текущую позицию каждого файлового указателя можно определить с помощью этих двух функций.

```
pos_type tellg();

pos_type tellp();
```

Здесь используется тип *pos_type* (он определен в классе ios), позволяющий хранить самое большое значение, которое может возвратить любая из этих функций.

Существуют перегруженные версии функций *seekg()* и *seekp()*, которые перемещают файловые указатели в позиции, заданные значениями, возвращаемыми функциями *tellg()* и *tellp()* соответственно. Вот как выглядят их прототипы,

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

Проверка статуса ввода-вывода

C++-система ввода-вывода поддерживает статусную информацию о результатах выполнения каждой операции ввода-вывода. Текущий статус потока ввода-вывода описывается в объекте типа *iostate*, который представляет собой перечисление (оно определено в классе ios), включающее следующие члены.

Имя	Значение
ios::goodbit	Ошибки отсутствуют
ios::eofbit	1 при обнаружении конца файла; 0 в противном случае
ios::failbit	1 при возникновении исправимой ошибки ввода-вывода; 0 в противном случае
ios::badbit	1 при возникновении неисправимой ошибки ввода-вывода; 0 в противном случае

Статусную информацию о результате выполнения операций ввода-вывода можно получать двумя способами. Во-первых, можно вызвать функцию *rdstate()*, которая является членом класса *ios*. Она имеет такой прототип.

```
iostate rdstate();
```

Функция *rdstate()* возвращает текущий статус флагов ошибок. Нетрудно догадаться, что, судя по приведенному выше списку флагов, функция *rdstate()* возвратит значение *goodbit* при отсутствии каких бы то ни было ошибок. В противном случае она возвращает соответствующий флаг ошибки.

Во-вторых, о наличии ошибки можно узнать с помощью одной или нескольких следующих функций-членов класса *ios*.

```
bool bad();
bool eof();
bool fail();
bool good();
```

Функция *eof()* рассматривалась выше. Функция *bad()* возвращает значение ИСТИНА, если в результате выполнения операции ввода-вывода был установлен флаг *badbit*. Функция *fail()* возвращает значение ИСТИНА, если в результате выполнения операции ввода-вывода был установлен флаг *failbit*. Функция *good()* возвращает значение ИСТИНА, если при выполнении операции ввода-вывода ошибок не произошло. В противном случае они возвращают значение ЛОЖЬ.

Если при выполнении операции ввода-вывода произошла ошибка, то, возможно, прежде чем продолжать выполнение программы, имеет смысл сбросить флаги ошибок. Для этого используйте функцию *clear()* (член класса ios), прототип которой выглядит так.

```
void clear (iostate flags = ios::goodbit);
```

Если параметр *flags* равен значению *goodbit* (оно устанавливается по умолчанию), все флаги ошибок очищаются. В противном случае флаги устанавливаются в соответствии с заданным вами значением.

Прежде чем переходить к следующему разделу, стоит опробовать функции, которые сообщают данные о состоянии флагов ошибок, внеся в предыдущие примеры программ код проверки ошибок.

Использование перегруженных операторов ввода-вывода при работе с файлами

Выше в этой главе вы узнали, как перегружать операторы ввода и вывода для собственных классов, а также как создавать собственные манипуляторы. В приведенных выше примерах программ выполнялись только операции консольного ввода-вывода. Но поскольку все C++-потоки одинаковы, одну и ту же перегруженную функцию вывода данных, например, можно использовать для вывода информации как на экран, так и в файл, не внося при этом никаких существенных изменений. Именно в этом и заключаются основные достоинства C++-системы ввода-вывода.

В следующей программе используется перегруженный (для класса *three_d*) оператор вывода для записи значений координат в файл *threed*.

```
/* Использование перегруженного оператора ввода-вывода для записи объектов класса three_d в файл.
*/
#include <iostream>
#include <fstream>
using namespace std;

class three_d {
   int x, y, z; // 3-мерные координаты; они теперь закрыты
 public:
   three_d(int a, int b, int c) { x = a; y = b; z = c; }

     friend ostream &operator<<(ostream &stream, three_d obj); /* Отображение координат X, Y, Z (оператор вывода для класса three_d). */
};

ostream &operator<<(ostream &stream, three_d obj)
{
 stream << obj.x << ", ";
 stream << obj.y << ", ";
 stream << obj.z << "\n";
 return stream; // возвращает поток
}

int main()
{
 three_d a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);
 ofstream out("threed");
 if(!out) {
   cout << "Не удается открыть файл.";
   return 1;
 }

 out << a << b << c;

 out.close();
 return 0:
```

```
}
```

Если сравнить эту версию операторной функции вывода данных для класса *three_d* с той, что была представлена в начале этой главы, можно убедиться в том, что для "настройки" ее на работу с дисковыми файлами никаких изменений вносить не пришлось. Если операторы ввода и вывода определены корректно, они будут успешно работать с любым потоком.

Важно! *Прежде чем переходить к следующей главе, не пожалейте времени и поработайте с С++-функциями ввода-вывода. Создайте собственный класс, а затем определите для него операторы ввода и вывода. А еще создайте собственные манипуляторы.*

Глава 19: Динамическая идентификация типов и операторы приведения типа

В этой главе рассматриваются два средства C++, которые поддерживают современное объектно-ориентированное программирование: *динамическая идентификация типов* (run-time type identification - RTTI) и набор дополнительных *операторов приведения типа*. Ни одно из этих средств не было частью оригинальной спецификации C++, но оба они были добавлены с целью усиления поддержки полиморфизма времени выполнения. Под RTTI понимается возможность проведения идентификации типа объекта во время выполнения программы. Рассматриваемые здесь операторы приведения типа предлагают программисту более безопасные способы выполнения этой операции. Как будет показано ниже, один из них, *dynamic_cast*, непосредственно связан с RTTI-идентификацией, поэтому операторы приведения типа и RTTI имеет смысл рассматривать в одной главе.

Динамическая идентификация типов (RTTI)

С динамической идентификацией типов вы, возможно, незнакомы, поскольку это средство отсутствует в таких неполиморфных языках, как C. В неполиморфных языках попросту нет необходимости в получении информации о типе во время выполнения программы, так как тип каждого объекта известен во время компиляции (т.е. еще при написании программы). Но в таких полиморфных языках, как C++, возможны ситуации, в которых тип объекта неизвестен в период компиляции, поскольку точная природа этого объекта не будет определена до тех пор, пока программа на начнет выполняться. Как вы знаете, C++ реализует полиморфизм посредством использования иерархии классов, виртуальных функций и указателей на базовые классы. Указатель на базовый класс можно использовать для ссылки на члены как этого базового класса, так и на члены любого объекта, выведенного из него. Следовательно, не всегда заранее известно, на объект какого типа будет ссылаться указатель на базовый класс в произвольный момент времени. Это выяснится только при выполнении программы — при использовании одного из средств динамической идентификации типов.

Для получения типа объекта во время выполнения программы используйте оператор typeid.

Чтобы получить тип объекта во время выполнения программы, используйте оператор *typeid*. Для этого необходимо включить в программу заголовок *<typeinfo>*. Самый распространенный формат использования оператора *typeid* таков.

```
typeid(object)
```

Здесь элемент *object* означает объект, тип которого нужно получить. Можно запрашивать не только встроенный тип, но и тип класса, созданного программистом. Оператор *typeid* возвращает ссылку на объект типа *type_info*, который описывает тип объекта *object*.

В классе *type_info* определены следующие public-члены.

```
bool operator = (const type_info &ob);

bool operator !=(const type_info &ob);

bool before(const type_info &ob);
```

```
const char *name( );
```

Перегруженные операторы *"=="* и *"!="* служат для сравнения типов. Функция *before()* возвращает значение *true*, если вызывающий объект в порядке сопоставления стоит перед объектом (элементом *ob*), используемым в качестве параметра. (Эта функция предназначена в основном для внутреннего использования. Возвращаемое ею значение результата не имеет ничего общего с наследованием или иерархией классов.) Функция *name()* возвращает указатель на имя типа.

Рассмотрим простой пример использования оператора *typeid*.

```
// Пример использования оператора typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class myclass {
  // . . .
};

int main( )
{
  int i, j;
  float f;
  myclass ob;

  cout << "Тип переменной i: " << typeid(i).name( );
  cout << endl;
  cout << "Тип переменной f: " << typeid(f).name( );
  cout << endl;
  cout << "Тип переменной ob: " << typeid(ob).name( );
  cout << "\n\n";

  if(typeid(i) == typeid(j))
    cout << "Типы переменных i и j одинаковы.\n";

  if(typeid(i) != typeid(f))
    cout << "Типы переменных i и f неодинаковы.\n";

  return 0;
}
```

При выполнении этой программы получены такие результаты.

```
Тип переменной i: int
Тип переменной f: float
Тип переменной ob: class myclass

Типы переменных i и j одинаковы.
Типы переменных i и f неодинаковы.
```

Если оператор *typeid* применяется к указателю на полиморфный базовый класс (вспомните: полиморфный класс — это класс, который содержит хотя бы одну виртуальную функцию), он автоматически возвращает тип реального объекта, на который тот указывает: будь то

объект базового класса или объект класса, выведенного из базового.

Следовательно, оператор *typeid* можно использовать для динамического определения типа объекта, адресуемого указателем на базовый класс. Применение этой возможности демонстрируется в следующей программе.

```
/* Пример применения оператора typeid к иерархии полиморфных классов.
*/
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
  virtual void f() {}; // делаем класс Base полиморфным
  // . . .
};

class Derived1: public Base {
  // . . .
};

class Derived2: public Base {
  // ...
};

int main()
{
  Base *p, baseob;
  Derived1 ob1;
  Derived2 ob2;

  p = &baseob;
  cout << "Переменная p указывает на объект типа ";
  cout << typeid(*p).name() << endl;

  p = &ob1;
  cout << "Переменная p указывает на объект типа ";
  cout << typeid(*p).name() << endl;

  p = &ob2;
  cout << "Переменная p указывает на объект типа ";
  cout << typeid(*p).name() << endl;

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы:

```
Переменная p указывает на объект типа Base
Переменная p указывает на объект типа Derived1
```

```
Переменная p указывает на объект типа Derived2
```

Если оператор *typeid* применяется к указателю на базовый класс полиморфного типа, тип реально адресуемого объекта, как подтверждают эти результаты, будет определен во время выполнения программы.

Во всех случаях применения оператора *typeid* к указателю на неполиморфную иерархию классов будет получен указатель на базовый тип, т.е. то, на что этот указатель реально указывает, определить нельзя. В качестве эксперимента превратите в комментарий виртуальную функцию *f()* в классе *Base* и посмотрите на результат. Вы увидите, что тип каждого объекта после внесения в программу этого изменения будет определен как *Base*, поскольку именно этот тип имеет указатель *p*.

Поскольку оператор *typeid* обычно применяется к разыменованному указателю (т.е. к указателю, к которому уже применен оператор *"*"*), для обработки ситуации, когда этот разыменованный указатель оказывается нулевым, создано специальное исключение. В этом случае оператор *typeid* генерирует исключение типа *bad_typeid*.

Ссылки на объекты иерархии полиморфных классов работают подобно указателям. Если оператор *typeid* применяется к ссылке на полиморфный класс, он возвращает тип объекта, на который она реально ссылается, и это может быть объект не базового, а производного типа. Описанное средство чаще всего используется при передаче объектов функциям по ссылке. Например, в следующей программе функция *WhatType()* объявляет ссылочный параметр на объекты типа *Base*. Это значит, что функции *WhatType()* можно передавать ссылки на объекты типа *Base* или ссылки на объекты любых классов, производных от *Base*. Оператор *typeid*, примененный к такому параметру, возвратит реальный тип объекта, переданного функции.

```
/* Применение оператора typeid к ссылочному параметру.
*/
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
  virtual void f() {}; // делаем класс Base полиморфным
  // . . .
};

class Derived1: public Base {
  // . . .
};

class Derived2: public Base {
  // . . .
};

/* Демонстрируем применение оператора typeid к ссылочному параметру.
*/
void WhatType(Base &ob)
{
  cout << "Параметр ob ссылается на объект типа ";
  cout << typeid(ob).name( ) << endl;
}
```

```
int main( )
{
  int i;
  Base baseob;
  Derived1 obi;
  Derived2 ob2;

  WhatType(baseob);
  WhatType(ob1);
  WhatType(ob2);

  return 0;
}
```

Эта программа генерирует такие результаты.

```
Параметр ob ссылается на объект типа Base
Параметр ob ссылается на объект типа Derived1
Параметр ob ссылается на объект типа Derived2
```

Существует еще одна версия применения оператора *typeid*, которая в качестве аргумента принимает имя типа. Формат ее таков.

```
type id(имя_типа)
```

Например, следующая инструкция совершенно допустима.

```
cout << typeid(int).name( );
```

Назначение этой версии оператора *typeid* — получить объект типа *type_info* (который описывает заданный тип данных), чтобы его можно было использовать в инструкции сравнения типов.

Пример RTTI-приложения

В следующей программе показано, насколько полезной может быть средство динамической идентификации типов (RTTI). Здесь используется модифицированная версия иерархии классов геометрических фигур из главы 15, которая вычисляет площадь круга, треугольника и прямоугольника. В данной программе определена функция *factory()*, предназначенная для создания экземпляра круга, треугольника или прямоугольника. Эта функция возвращает указатель на созданный объект. (Функция, которая генерирует объекты, иногда называется *фабрикой объектов*.) Конкретный тип создаваемого объекта определяется в результате обращения к функции *rand()* C++-генератора случайных чисел. Таким образом, мы не можем знать заранее, объект какого типа будет сгенерирован. Программа создает десять объектов и подсчитывает количество созданных фигур каждого типа. Поскольку при вызове функции *factory()* может быть сгенерирована фигура любого типа, для определения типа реально созданного объекта в программе используется оператор *typeid*.

```
/* Демонстрация использования средства динамической идентификации типов.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

class figure {
  protected:
```

```
    double x, y;
  public:
    figure(double i, double j) {
      x = i;
      y = j;
    }
    virtual double area( ) = 0;
};

class triangle : public figure {
  public:
    triangle(double i, double j) : figure(i, j) {}
    double area( ) {
      return x * 0.5 * y;
    }
};

class rectangle : public figure {
  public:
    rectangle(double i, double j) : figure (i, j) {}
    double area( ) { return x * y;}
};

class circle : public figure {
  public:
    circle(double i, double j=0) : figure(i, j) {}
    double area( ) {return 3.14 * x * x;}
};

// Фабрика объектов класса figure.
figure *factory( )
{
  switch(rand( ) % 3 ) {
    case 0: return new circle (10.0);
    case 1: return new triangle (10.1, 5.3);
    case 2: return new rectangle (4.3, 5.7);
  }
  return 0;
}

int main( )
{
  figure *p; // указатель на базовый класс
  int i;
```

```
  int t=0, r=0, c=0;

  // генерируем и подсчитываем объекты
  for(i=0; i<10; i++) {
    p = factory(); // генерируем объект
    cout << "Объект имеет тип " << typeid(*p).name();
    cout << ". ";

    // учитываем этот объект
    if(typeid(*p) == typeid(triangle)) t++;
    if(typeid(*p) == typeid(rectangle)) r++;
    if(typeid(*p) == typeid(circle)) c++;

    // отображаем площадь фигуры
    cout << "Площадь равна " << p->area() << endl;
  }

  cout << endl;
  cout << "Сгенерированы такие объекты:\n";
  cout << " треугольников: " << t << endl;
  cout << " прямоугольников: " << r << endl;
  cout << " кругов: " << c << endl;

  return 0;
}
```

Возможный результат выполнения этой программы таков.

```
Объект имеет тип class rectangle. Площадь равна 24.51
Объект имеет тип class rectangle. Площадь равна 24.51
Объект имеет тип class triangle. Площадь равна 26.765
Объект имеет тип class triangle. Площадь равна 26.765
Объект имеет тип class rectangle. Площадь равна 24.51
Объект имеет тип class triangle. Площадь равна 26.765
Объект имеет тип class circle. Площадь равна 314
Объект имеет тип class circle. Площадь равна 314
Объект имеет тип class triangle. Площадь равна 26.765
Объект имеет тип class rectangle. Площадь равна 24.51

Сгенерированы такие объекты:
  треугольников: 4
  прямоугольников: 4
  кругов: 2
```

Применение оператора typeid к шаблонным классам

Оператор *typeid* можно применить и к шаблонным классам. Тип объекта, который является экземпляром шаблонного класса, определяется частично на основании того, какие именно данные используются для его обобщенных данных при реализации объекта. Таким образом

данные используются для его обобщенных данных при реализации объекта. Таким образом, два экземпляра одного и того же шаблонного класса, которые создаются с использованием различных данных, имеют различный тип. Рассмотрим простой пример.

```
/* Использование оператора typeid с шаблонными классами.
*/
#include <iostream>
using namespace std;

template <class T>
class myclass {
   T a;
  public:
   myclass(T i) { a = i; }
   // . . .
};

int main()
{
  myclass<int> o1(10), o2(9);
  myclass<double> o3(7.2);

  cout << "Объект o1 имеет тип ";
  cout << typeid(o1).name() << endl;

  cout << "Объект o2 имеет тип ";
  cout << typeid(o2).name() << endl;

  cout << "Объект o3 имеет тип ";
  cout << typeid(o3).name() << endl;

  cout << endl;

  if(typeid(o1) == typeid(o2))
    cout << "Объекты o1 и o2 имеют одинаковый тип.\n";

  if(typeid(o1) == typeid(o3)) cout << "Ошибка\n";
  else cout << "Объекты o1 и o3 имеют разные типы.\n";

  return 0;
}
```

Эта программа генерирует такие результаты.

```
Объект o1 имеет тип class myclass<int>
Объект o2 имеет тип class myclass<int>
Объект o3 имеет тип class myclass<double>

Объекты o1 и o2 имеют одинаковый тип.
Объекты o1 и o3 имеют разные типы.
```

Как видите, несмотря на то, что два объекта являются экземплярами одного и того же шаблонного класса, если их параметризованные данные не совпадают, они не эквивалентны

по типу. В этой программе объект *ob* имеет тип *myclass<int>*, а объект *ob3* — тип *myclass<double>*. Таким образом, это объекты разного типа.

Рассмотрим еще один пример применения оператора *typeid* к шаблонным классам, а именно модифицированную версию программы определения геометрических фигур из предыдущего раздела. На этот раз класс figure мы сделали шаблонным.

```
// Шаблонная версия figure-иерархии.
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T>
class figure
{
  protected:
    T x, y;
  public:
    figure(T i, T j) {
      x = i;
      y = j;
    }
    virtual T area() = 0;
};

template <class T>
class triangle : public figure<T>
{
  public:
    triangle(T i, T j) : figure<T>(i, j) {}
    T area() {
      return x * 0.5 * y;
    }
};

template <class T>
class rectangle : public figure<T>
{
  public:
    rectangle(T i, T j) : figure<T>(i, j) {}
    T area() {
      return x * y;
    }
};

template <class T>
class circle : public figure<T>
```

```
{
  public:
    circle(T i, T j=0) : figure<T>(i, j) {}
    T area() {
      return 3.14 * x * x;
    }
};

// Фабрика объектов, генерируемых из класса figure.
figure<double> *generator()
{
  switch(rand() % 3 ) {
    case 0: return new circle<double> (10.0);
    case 1: return new triangle<double>(10.1, 5.3);
    case 2: return new rectangle<double> (4.3, 5.7);
  }
  return 0;
}

int main()
{
  figure<double> *p;
  int i;
  int t=0, c=0, r=0;

  // генерируем и подсчитываем объекты
  for(i=0; i<10; i++) {
    p = generator();
    cout << "Объект имеет тип " << typeid(*p).name();
    cout << ". ";

    // учитываем объект
    if(typeid(*p) == typeid(triangle<double>)) t++;
    if(typeid(*p) == typeid(rectangle<double>)) r++;
    if(typeid(*p) == typeid(circle<double>)) c++;

    cout << "Площадь равна " << p->area() << endl;
  }

  cout << endl;
  cout << "Сгенерированы такие объекты:\n";
  cout << " треугольников: " << t << endl;
  cout << " прямоугольников: " << r << endl;
  cout << " кругов: " << c << endl;

  return 0;
```

```
  return 0;
}
```

Вот как выглядит возможный результат выполнения этой программы.

```
Объект имеет тип class rectangle<double>. Площадь равна 24.51
Объект имеет тип class rectangle<double>. Площадь равна 24.51
Объект имеет тип class triangle<double>. Площадь равна 26.765
Объект имеет тип class triangle<double>. Площадь равна 26.765
Объект имеет тип class rectangle<double>. Площадь равна 24.51
Объект имеет тип class triangle<double>. Площадь равна 26.765
Объект имеет тип class circle<double>. Площадь равна 314
Объект имеет тип class circle<double>. Площадь равна 314
Объект имеет тип class triangle<double>. Площадь равна 26.765
Объект имеет тип class rectangle<double>. Площадь равна 24.51

Сгенерированы такие объекты:
  треугольников: 4
  прямоугольников: 4
  кругов: 2
```

Динамическая идентификация типов используется не в каждой программе. Но при работе с полиморфными типами это средство позволяет узнать тип объекта, обрабатываемого в любой произвольный момент времени.

Операторы приведения типов

В C++ определено пять операторов приведения типов. Первый оператор (он описан выше в этой книге), применяемый в обычном (традиционном) стиле, был с самого начала встроен в C++. Остальные четыре (*dynamic_cast*, *const_cast*, *reinterpret_cast* и *static_cast*) были добавлены в язык всего несколько лет назад. Эти операторы предоставляют дополнительные *"рычаги управления"* характером выполнения операций приведения типа. Рассмотрим каждый из них в отдельности.

Оператор dynamic_cast

Оператор dynamic_cast выполняет операцию приведения полиморфных типов во время выполнения программы.

Возможно, самым важным из новых операторов является оператор динамического приведения типов *dynamic_cast*. Во время выполнения программы он проверяет обоснованность предлагаемой операции. Если в момент его вызова заданная операция оказывается недопустимой, приведение типов не производится. Общий формат применения оператора *dynamic_cast* таков.

dynamic_cast<type> (expr)

Здесь элемент *type* означает новый тип, который является целью выполнения этой операции, а элемент *expr*— выражение, приводимое к этому новому типу. Тип *type* должен быть представлен указателем или ссылкой, а выражение *expr* должно приводиться к указателю или ссылке. Таким образом, оператор *dynamic_cast* можно использовать для преобразования указателя одного типа в указатель другого или ссылки одного типа в ссылку другого.

Этот оператор в основном используется для динамического выполнения операций приведения типа среди полиморфных типов. Например, если даны полиморфные классы *B* и *D*, причем класс *D* выведен из класса *B*, то с помощью оператора *dynamic_cast* всегда можно преобразовать указатель *D** в указатель *B**, поскольку указатель на базовый класс всегда

можно использовать для указания на объект класса, выведенного из базового. Однако оператор *dynamic_cast* может преобразовать указатель *B** в указатель *D** только в том случае, если адресуемым объектом действительно является объект класса *D*. И, вообще, оператор *dynamic_cast* будет успешно выполнен только при условии, если разрешено полиморфное приведение типов, т.е. если указатель (или ссылка), приводимый к новому типу, может указывать (или ссылаться) на объект этого нового типа или объект, выведенный из него. В противном случае, т.е. если заданную операцию приведения типов выполнить нельзя, результат действия оператора *dynamic_cast* оценивается как нулевой, если в этой операции участвуют указатели. (Если же попытка выполнить эту операцию оказалась неудачной при участии в ней ссылок, генерируется исключение типа *bad_cast*.)

Рассмотрим простой пример. Предположим, что класс *Base* — полиморфный, а класс *Derived* выведен из класса *Base*.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // Указатель на базовый класс указывает на объект класса Derived.

dp = dynamic_cast<Derived *> (bp); // Приведение к указателю на производный класс разрешено.
if(dp) cout << "Приведение типа прошло успешно!";
```

Здесь приведение указателя *bp* (на базовый класс) к указателю *dp* (на производный класс) успешно выполняется, поскольку *bp действительно* указывает на объект класса *Derived*. Поэтому при выполнении этого фрагмента кода будет выведено сообщение *Приведение типа прошло успешно!*. Но в следующем фрагменте кода попытка совершить операцию приведения типа будет неудачной, поскольку *bp* в действительности указывает на объект класса *Base*, и неправомерно приводить указатель на базовый класс к типу указателя на производный, если адресуемый им объект не является на самом деле объектом производного класса.

```
bp = &b_ob; /* Указатель на базовый класс ссылается на объект класса Base. */

dp = dynamic_cast<Derived *> (bp); // ошибка!
if(!dp) cout << "Приведение типа выполнить не удалось";
```

Поскольку попытка выполнить операцию приведения типа оказалась неудачной, при выполнении этого фрагмента кода будет выведено сообщение *Приведение типа выполнить не удалось*.

В следующей программе демонстрируются различные ситуации применения оператора *dynamic_cast*.

```
// Использование оператора dynamic_cast.
#include <iostream>
using namespace std;

class Base {
  public:
    virtual void f() { cout << "В классе Base.\n"; }
    // . . .
};

class Derived : public Base {
```

```
  public:
    void f() { cout << "В классе Derived.\n"; }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
     cout << "Приведение типов " <<"(из Derived * в Derived *) реализовано.\n";
    dp->f();
  }
  else  cout <<"Ошибка\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&d_ob);
  if(bp) {
     cout << "Приведение типов " <<"(из Derived * в Base *) реализовано.\n";
    bp->f();
  }
  else  cout << "Ошибка\n";

  cout << endl;

  bp = dynamic_cast<Base *> (&b_ob);
  if(bp) {
    cout << "Приведение типов " <<"(из Base * в Base *) реализовано.\n";
    bp->f();
  }
  else  cout << "Ошибка\n";

  cout << endl;

  dp = dynamic_cast<Derived *> (&b_ob);
  if(dp) cout <<"Ошибка\n";
  else
     cout <<"Приведение типов " <<"(из Base * в Derived *) не реализовано.\n";

  cout << endl;

  bp = &d_ob; // bp указывает на объект класса Derived
  dp = dynamic_cast<Derived *> (bp);
  if(dp) {
```

```
    cout << "Приведение bp к типу Derived *\n" << "реализовано, поскольку
bp действительно\n" << "указывает на объект класса Derived.\n";
    dp->f();
  }
  else  cout << "Ошибка\n";

  cout << endl;

  bp = &b_ob; // bp указывает на объект класса Base
  dp = dynamic_cast<Derived *> (bp);
  if(dp) cout << "Ошибка";
  else {
    cout <<"Теперь приведение bp к типу Derived *\n" <<"не реализовано,
поскольку bp\n" <<"в действительности указывает на объект\n" <<"класса
Base.\n";
  }

  cout << endl;

  dp = &d_ob; // dp указывает на объект класса Derived
  bp = dynamic_cast<Base *> (dp);
  if(bp) {
    cout <<"Приведение dp к типу Base * реализовано.\n";
    bp->f();
  }
  else  cout <<"Ошибка\n";

  return 0;
}
```

Программа генерирует такие результаты.

```
Приведение типов (из Derived* в Derived*) реализовано.
В классе Derived.

Приведение типов (из Derived* в Base*) реализовано.
В классе Derived.

Приведение типов (из Base* в Base*) реализовано.
В классе Base.

Приведение типов (из Base* в Derived*) не реализовано.

Приведение bp к типу Derived* реализовано, поскольку bp действительно
указывает на объект класса Derived.
В классе Derived.

Теперь приведение bp к типу Derived* не реализовано, поскольку bp в
действительности указывает на объект класса Base.
```

```
Приведение dp к типу Base * реализовано.
В классе Derived.
```

Оператор *dynamic_cast* можно иногда использовать вместо оператора *typeid*. Например, предположим, что класс *Base* — полиморфный и является базовым для класса *Derived*, тогда при выполнении следующего фрагмента кода указателю *dp* будет присвоен адрес объекта, адресуемого указателем *bp*, но только в том случае, если этот объект действительно является объектом класса *Derived*.

```
Base *bp;
Derived *dp;
// . . .
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

В этом случае используется обычная операция приведения типов. Здесь это вполне безопасно, поскольку инструкция if проверяет законность операции приведения типов с помощью оператора *typeid* до ее реального выполнения. То же самое можно сделать более эффективно, заменив операторы *typeid* и инструкцию if оператором

```
dynamic_cast:
dp = dynamic_cast<Derived *> (bp);
```

Поскольку оператор *dynamic_cast* успешно выполняется только в том случае, если объект, подвергаемый операции приведения к типу, уже является объектом либо заданного типа, либо типа, выведенного из заданного, то после завершения этой инструкции указатель *dp* будет содержать либо нулевое значение, либо указатель на объект типа *Derived*. Кроме того, поскольку оператор *dynamic_cast* успешно выполняется только в том случае, если заданная операция приведения типов законна, то в определенных ситуациях ее логику можно упростить. В следующей программе показано, как оператор *typeid* можно заменить оператором *dynamic_cast*. Здесь выполняется один и тот же набор операций дважды: с использованием сначала оператора *typeid*, а затем оператора *dynamic_cast*.

```
/* Использование оператора dynamic_cast вместо оператора typeid.
*/
#include <iostream>
#include <typeinfo>
using namespace std;

class Base {
  public:
    virtual void f() {}
};

class Derived : public Base {
  public:
    void derivedOnly() {
      cout << "Это объект класса Derived.\n";
    }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;
```

```
  //-------------------------------
  // Использование оператора typeid
  //-------------------------------
  bp = &b_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else  cout <<"Операция приведения объекта типа Base к " <<"типу Derived
не выполнилась.\n";

  bp = &d_ob;
  if(typeid(*bp) == typeid(Derived)) {
    dp = (Derived *) bp;
    dp->derivedOnly();
  }
  else  cout <<"Ошибка, приведение типа должно " <<"быть реализовано!\n";

  //-------------------------------------
  // Использование оператора dynamic_cast
  //-------------------------------------
  bp = &b_ob;
  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
  else  cout << "Операция приведения объекта типа Base к " <<" типу
Derived не выполнилась.\n"; bp = &d_ob;

  dp = dynamic_cast<Derived *> (bp);
  if(dp) dp->derivedOnly();
   else    cout << "Ошибка, приведение типа должно " << "быть
реализовано!\n";
  return 0;
}
```

Как видите, использование оператора *dynamic_cast* упрощает логику, необходимую для преобразования указателя на базовый класс в указатель на производный класс. Вот как выглядят результаты выполнения этой программы.

```
Операция приведения объекта типа Base к типу Derived не выполнилась.

Это объект класса Derived. Операция приведения объекта типа Base к типу
Derived не выполнилась. Это объект класса Derived.
```

И еще. Оператор *dynamic_cast* можно также использовать применительно к шаблонным классам.

Оператор const_cast

Оператор const_cast переопределяет модификаторы const и/или volatile.

Оператор *const_cast* используется для явного переопределения модификаторов *const* и/или *volatile*. Новый тип должен совпадать с исходным, за исключением его атрибутов *const* или *volatile*. Чаще всего оператор *const_cast* используется для удаления признака постоянства (атрибута *const*). Его общий формат имеет следующий вид.

```
const_cast<type> (expr)
```

Здесь элемент *type* задает новый тип операции приведения, а элемент *expr* означает выражение, которое приводится к новому типу.

Использование оператора *const_cast* демонстрируется в следующей программе.

```
// Демонстрация использования оператора const_cast.
#include <iostream>
using namespace std;

void f (const int *p)
{
  int *v;
  // Переопределение const-атрибута.
  v = const_cast<int *> (p);
  *v = 100; // теперь объект можно модифицировать
}

int main( )
{
  int x = 99;
  cout << "Значение x до вызова функции f( ): " << x<< endl;

  f (&x);
  cout <<"Значение x после вызова функции f( ): " << x<< endl;

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Значение x до вызова функции f( ): 99
Значение x после функции f( ): 100
```

Как видите, переменная *x* была модифицирована функцией *f()*, хотя параметр, принимаемый ею, задается как *const*-указатель.

Необходимо подчеркнуть, что использование оператора *const_cast* для удаления *const*-атрибута является потенциально опасным средством. Поэтому обращайтесь с ним очень осторожно.

И еще. Удалять *const*-атрибут способен только оператор *const_cast*. Другими словами, ни *dynamic_cast*, ни *static_cast*, ни *reinterpret_cast* нельзя использовать для изменения *const*-атрибута объекта.

Оператор static_cast

Оператор static_cast выполняет операцию неполиморфного приведения типов.

Оператор *static_cast* выполняет операцию неполиморфного приведения типов. Его можно использовать для любого стандартного преобразования. При этом во время работы программы никаких проверок на допустимость не выполняется. Оператор *static_cast* имеет следующий общий формат записи.

```
static_cast<type> (expr)
```

Здесь элемент *type* задает новый тип операции приведения, а элемент *expr* означает выражение, которое приводится к этому новому типу.

Оператор *static_cast*, по сути, является заменителем оригинального оператора приведения типов. Он лишь выполняет неполиморфное преобразование. Например, при выполнении следующей программы переменная типа *float* приводится к типу *int*.

```
// Использование оператора static_cast.
#include <iostream>
using namespace std;

int main( )
{
  int i;
  float f;

  f = 199.22F;
  i = static_cast<int> (f);
  cout << i;

  return 0;
}
```

Оператор reinterpret_cast

Оператор reinterpret_cast выполняет фундаментальное изменение типа.

Оператор *reinterpret_cast* преобразует один тип в принципиально другой. Например, его можно использовать для преобразования указателя в целое значение и целого значения — в указатель. Его также можно использовать для приведения наследственно несовместимых типов указателей. Этот оператор имеет следующий общий формат записи.

```
reinterpret_cast<type> (expr)
```

Здесь элемент *type* задает новый тип операции приведения, а элемент *expr* означает выражение, которое приводится к этому новому типу.

Использование оператора *reinterpret_cast* демонстрируется в следующей программе.

```
// Пример использования оператора reinterpret_cast.
#include <iostream>
using namespace std;

int main( )
{
  int i;
  char *p = "Это короткая строка.";
  i = reinterpret_cast<int> (p); // Приводим указатель к типу int.
  cout << i;
  return 0;
}
```

Здесь оператор *reinterpret_cast* преобразует указатель *p* в целочисленное значение. Данное преобразование представляет фундаментальное изменение типа.

Сравнение обычной операции приведения типов с новыми четырьмя cast-операторами

Кому-то из читателей могло бы показаться, что описанные выше четыре *cast*-оператора полностью заменяют традиционную операцию приведения типов. И тогда у них может возникнуть такой вопрос: *"Стоит ли всегда вместо обычной операции приведения типов использовать более новые средства?"*. Дело в том, что общего правила для всех программистов не существует. Поскольку новые операторы были созданы для повышения безопасности довольно рискованной операции приведения одного типа данных к другому, многие C++-программисты убеждены, что их следует использовать исключительно с этой целью. И здесь трудно что-либо возразить. Другие же программисты считают, что поскольку традиционная операция приведения типов служила им *"верой и правдой"* в течение многих лет, то от нее не стоит так легко отказываться. Например, для выполнения простых и относительно безопасных операций приведения типов (как те, что требуются при вызове функций ввода-вывода *read()* и *write()*, описанных в предыдущей главе) *"старое доброе"* средство вполне приемлемо.

Существует еще одна точка зрения, с которой трудно не согласиться: при выполнении операций приведения полиморфных типов определенно стоит использовать оператор *dynamic_cast*.

Глава 20: Пространства имен и другие темы

В этой главе описаны *пространства имен* и такие эффективные средства, как *explicit*-конструкторы, *указатели на функции*, *static*-члены, *const*-функции-члены, альтернативный синтаксис инициализации членов класса, операторы указания на члены, ключевое слово *asm*, спецификация компоновки и функции преобразования.

Пространства имен

Пространство имен определяет некоторую декларативную область.

Пространства имен мы кратко рассмотрели в главе 2. Они позволяют локализовать имена идентификаторов, чтобы избежать конфликтных ситуаций с ними. В C++-среде программирования используется огромное количество имен переменных, функций и имен классов. До введения пространств имен все эти имена конкурировали за память в глобальном пространстве имен, что и было причиной возникновения многих конфликтов. Например, если бы в вашей программе была определена функция *toupper()*, она могла бы (в зависимости от списка параметров) переопределить стандартную библиотечную функцию *toupper()*, поскольку оба имени должны были бы храниться в глобальном пространстве имен. Конфликты с именами возникали также при использовании одной программой нескольких библиотек сторонних производителей. В этом случае имя, определенное в одной библиотеке, конфликтовало с таким же именем из другой библиотеки. Подобная ситуация особенно неприятна при использовании одноименных классов. Например, если в вашей программе определен класс *VideoMode*, и в библиотеке, используемой вашей программой, определен класс с таким же именем, конфликта не избежать.

Для решения описанной проблемы было создано ключевое слово *namespace*. Поскольку оно локализует видимость объявленных в нем имен, это значит, что пространство имен позволяет использовать одно и то же имя в различных контекстах, не вызывая при этом конфликта имен. Возможно, больше всего от нововведения "повезло" C++-библиотеке стандартных функций. До появления ключевого слова *namespace* вся C++-библиотека была определена в глобальном пространстве имен (которое было, конечно же, единственным). С наступлением *namespace-"эры"* C++-библиотека определяется в собственном пространстве имен, именуемом *std*, которое значительно понизило вероятность возникновения конфликтов имен. В своей программе программист волен создавать собственные пространства имен, чтобы локализовать видимость тех имен, которые, по его мнению, могут стать причиной конфликта. Это особенно важно, если вы занимаетесь созданием библиотек классов или функций.

Понятие пространства имен

Ключевое слово *namespace* позволяет разделить глобальное пространство имен путем

создания некоторой декларативной области. По сути, пространство имен определяет область видимости. Общий формат задания пространства имен таков.

```
namespace name {
  // объявления
}
```

Все, что определено в границах инструкции namespace, находится в области видимости этого пространства имен.

В следующей программе приведен пример использования *namespace*-инструкции. Она локализует имена, используемые для реализации простого класса счета в обратном направлении. В созданном здесь пространстве имен определяется класс *counter*, который реализует счетчик, и переменные *upperbound* и *lowerbound*, содержащие значения верхней и нижней границ, применяемых для всех счетчиков.

```
namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n) {
       if(n <= upperbound) count = n;
     }

     int run() {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}
```

Здесь переменные *upperbound* и *lowerbound*, а также класс *counter* являются частью области видимости, определенной пространством имен *CounterNameSpace*.

В любом пространстве имен к идентификаторам, которые в нем объявлены, можно обращаться напрямую, т.е. без указания этого пространства имен. Например, в функции *run()*, которая находится в пространстве имен *CounterNameSpace*, можно напрямую обращаться к переменной *lowerbound*:

```
if(count > lowerbound) return count--;
```

Но поскольку инструкция *namespace* определяет область видимости, то при обращении к объектам, объявленным в пространстве имен, извне этого пространства необходимо использовать оператор разрешения области видимости. Например, чтобы присвоить значение *10* переменной *upperbound* из кода, который является внешним по отношению к пространству имен *CounterNameSpace*, нужно использовать такую инструкцию.

```
CounterNameSpace::upperbound = 10;
```

Чтобы объявить объект типа *counter* вне пространства имен *CounterNameSpace*, используйте инструкцию, подобную следующей.

```
CounterNameSpace::counter ob;
```

В общем случае, чтобы получить доступ к некоторому члену пространства имен извне этого пространства, необходимо предварить имя этого члена именем пространства и разделить эти имена оператором разрешения области видимости.

Рассмотрим программу, в которой демонстрируется использование пространства имен *CounterNameSpace*.

```
// Демонстрация использования пространства имен.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
    public:
      counter (int n) {
        if(n <= upperbound) count = n;
        else count = upperbound;
      }

      void reset (int n) {
        if(n <= upperbound) count = n;
      }

      int run( ) {
        if(count > lowerbound) return count--;
        else return lowerbound;
      }
  };
}

int main( )
{
  CounterNameSpace::upperbound = 100;
  CounterNameSpace::lowerbound = 0;
  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run( );
    cout << i << " ";
```

```
  }while(i > CounterNameSpace :: lowerbound);
  cout << endl;

  CounterNameSpace::counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  }while(i > CounterNameSpace::lowerbound);
  cout << endl;

  ob2.reset(100);
  CounterNameSpace::lowerbound = 90;

  do {
    i = ob2.run();
    cout << i << " ";
  }while(i > CounterNameSpace::lowerbound);

  return 0;
}
```

Обратите внимание на то, что при объявлении объекта класса *counter* и обращении к переменным *upperbound* и *lowerbound* используется имя пространства имен *CounterNameSpace*. Но после объявления объекта типа *counter* уже нет необходимости в полной квалификации его самого или его членов. Поскольку пространство имен однозначно определено, функцию *run()* объекта *ob1* можно вызывать напрямую, т.е. без указания (в качестве префикса) пространства имен (*ob1.run()*).

Программа может содержать несколько объявлений пространств имен с одинаковыми именами. Это означает, что пространство имен можно разбить на несколько файлов или на несколько частей в рамках одного файла. Вот пример.

```
namespace NS {
  int i;
}
  // . . .
namespace NS {
  int j;
}
```

Здесь пространство имен *NS* разделено на две части. Однако содержимое каждой части относится к одному и тому же пространству имен *NS*.

Любое пространство имен должно быть объявлено вне всех остальных областей видимости. Это означает, что нельзя объявлять пространства имен, которые локализованы, например, в рамках функции. При этом одно пространство имен может быть вложено в другое.

Инструкция using

Инструкция using делает заданное пространство имен "видимым", т.е. действующим.

Если программа включает множество ссылок на члены некоторого пространства имен, то нетрудно представить, что необходимость указывать имя этого пространства имен при каждом к ним обращении, очень скоро утомит вас. Эту проблему позволяет решить инструкция *using*, которая применяется в двух форматах.

```
using namespace имя;
using name::член;
```

В первой форме элемент *имя* задает название пространства имен, к которому вы хотите получить доступ. Все члены, определенные внутри заданного пространства имен, попадают в *"поле видимости"*, т.е. становятся частью текущего пространства имен и их можно затем использовать без квалификации (уточнения пространства имен). Во второй форме делается *"видимым"* только указанный член пространства имен. Например, предполагая, что пространство имен *CounterNameSpace* определено (как показано выше), следующие инструкции *using* и присваивания будут вполне законными.

```
using CounterNameSpace::lowerbound; /* Видимым стал только член lowerbound */

lowerbound = 10; /* Все в порядке, поскольку член lowerbound находится в области видимости. */

using namespace CounterNameSpace; // Все члены видимы.
upperbound = 100; // Все в порядке, поскольку все члены видимы.
```

Использование инструкции *using* демонстрируется в следующей программе (которая представляет собой новый вариант счетчика из предыдущего раздела).

```
// Использование инструкции using.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
      int count;
    public:
      counter (int n) {
        if(n <= upperbound) count = n;
        else count = upperbound;
      }

      void reset(int n) {
        if(n <= upperbound) count = n;
      }

      int run() {
        if(count > lowerbound) return count--;
        else return lowerbound;
      }
  };
}

int main()
{
```

```
i

   /* Используется только член upperbound из пространства имен
CounterNameSpace. */
  using CounterNameSpace::upperbound;

   /* Теперь для установки значения переменной upperbound не нужно
указывать пространство имен. */
  upperbound = 100;

   /* Но при обращении к переменной lowerbound и другим объектам по-
прежнему необходимо указывать пространство имен. */
  CounterNameSpace::lowerbound = 0;
  CounterNameSpace::counter ob1(10);
  int i;

  do {
    i = ob1.run();
    cout << i << " ";
  }while(i > CounterNameSpace::lowerbound);
  cout. << endl;

  /* Теперь используем все пространство имен CounterNameSpace. */
  using namespace CounterNameSpace;

  counter ob2(20);

  do {
    i = ob2.run();
    cout << i << " ";
  }while(i > lowerbound);
  cout << endl;

  ob2.reset(100);
  lowerbound = 90;
  do {
    i = ob2.run();
    cout << i << " ";
  }while(i > lowerbound);

  return 0;
}
```

Эта программа иллюстрирует еще один важный момент. Использование одного пространства имен не переопределяет другое. Если некоторое пространство имен становится *"видимым"*, это значит, что оно просто добавляет свои имена к именам других, уже действующих пространств. Поэтому к концу этой программы к глобальному пространству имен добавились и *std*, и *CounterNameSpace*.

Неименованные пространства имен

Неименованное пространство имен ограничивает идентификаторы рамками файла, в котором они объявлены.

Существует *неименованное* пространство имен специального типа, которое позволяет создавать идентификаторы, уникальные для данного файла. Общий формат его объявления выглядит так.

```
namespace {
  // объявления
}
```

Неименованные пространства имен позволяют устанавливать уникальные идентификаторы, которые известны только в области видимости одного файла. Другими словами, члены файла, который содержит неименованное пространство имен, можно использовать напрямую, без уточняющего префикса. Но вне файла эти идентификаторы неизвестны.

Как упоминалось выше в этой книге, использование модификатора типа *static* также позволяет ограничить область видимости глобального пространства имен файлом, в котором он объявлен. Например, рассмотрим следующие два файла, которые являются частью одной и той же программы.

Файл One	**Файл** Two

`static int k;`
`void f1() {`
`  k = 99; // OK`
`}`
	`extern int k;`
	`void f2() {`
	`  k = 10; // ошибка`
	`}`

Поскольку переменная *k* определена в файле *One*, ее и можно использовать в файле *One*. В файле *Two* переменная *k* определена как внешняя (extern-переменная), а это значит, что ее имя и тип известны, но сама переменная *k* в действительности не определена. Когда эти два файла будут скомпонованы, попытка использовать переменную *k* в файле *Two* приведет к возникновению ошибки, поскольку в нем нет определения для переменной *k*. Тот факт, что *k* объявлена *static*-переменной в файле *One*, означает, что ее область видимости ограничивается этим файлом, и поэтому она недоступна для файла *Two*.

Несмотря на то что использование глобальных *static*-объявлений все еще разрешено в C++, для локализации идентификатора в рамках одного файла лучше использовать неименованное пространство имен. Рассмотрим пример.

Файл One	**Файл** Two

`namespace {`
`  int k;`
`}`
`static int k;`
`void f1() {`
`  k = 99; // OK`
`}`
	`extern int k;`
	`void f2() {`
	`  k = 10; // ошибка`
	`}`

Здесь переменная *k* также ограничена рамками файла *One*. Для новых программ рекомендуется использовать вместо модификатора *static* неименованное пространство имен.

Обычно для большинства коротких программ и программ среднего размера нет необходимости в создании пространств имен. Но, формируя библиотеки многократно используемых функций или классов, имеет смысл заключить свой код (если хотите обеспечить его максимальную переносимость) в собственное пространство имен.

Пространство имен std

Пространство имен std используется библиотекой C++.

Стандарт C++ определяет всю свою библиотеку в собственном пространстве имен, именуемом *std*. Именно по этой причине большинство программ в этой книге включает следующую инструкцию:

следующую инструкцию:

```
using namespace std;
```

При выполнении этой инструкции пространство имен *std* становится текущим, что открывает прямой доступ к именам функций и классов, определенных в этой библиотеке, т.е. при обращении к ним отпадает необходимость в использовании префикса *std::*.

Конечно, при желании можно явным образом квалифицировать каждое библиотечное имя префиксом *std::*. Например, следующая программа не привносит библиотеку в глобальное пространство имен.

```
// Использование явно заданной квалификации std::.
#include <iostream>

int main( )
{
  double val;

  std::cout << "Введите число: ";
    std::cin >> val;

  std::cout << "Вы ввели число ";
  std::cout << val;

  return 0;
}
```

Здесь имена *cout* и *cin* явно дополнены именами своих пространств имен. Итак, чтобы записать данные в стандартный выходной поток, следует использовать не просто имя потока *cout*, а имя с префиксом *std::cout*, а чтобы считать данные из стандартного входного потока, нужно применить "префиксное" имя *std::cin*.

Если ваша программа использует стандартную библиотеку только в ограниченных пределах, то, возможно, ее и не стоит вносить в глобальное пространство имен. Но если ваша программа содержит сотни ссылок на библиотечные имена, то гораздо проще сделать пространство имен *std* текущим, чем полностью квалифицировать каждое имя в отдельности.

Если вы используете только несколько имен из стандартной библиотеки, то, вероятно, имеет смысл использовать инструкцию *using* для каждого из них в отдельности. Преимущество этого подхода состоит в том, что эти имена можно по-прежнему использовать без префикса *std::*, не внося при этом всю библиотеку стандартных функций в глобальное пространство имен. Рассмотрим пример.

```
/* Внесение в глобальное пространство имен лишь нескольких имен.
*/
#include <iostream>

// Получаем доступ к именам потоков cout и cin.
using std::cout;
using std::cin;

int main( )
{
  double val;

  cout << "Введите число: ";
```

```
  cout << "Введите число: ";
    cin >> val;

  cout << "Вы ввели число ";
  cout << val;

  return 0;
}
```

Здесь имена потоков *cin* и *cout* можно использовать напрямую, но остальная часть пространства имен *std* не внесена в область видимости.

Как упоминалось выше, исходная библиотека C++ была определена в глобальном пространстве имен. Если вам придется модернизировать старые C++-программы, то вы должны либо включить в них инструкцию *using namespace std*, либо дополнить каждое обращение к члену библиотеки префиксом *std::*. Это особенно важно, если вам придется заменять старые заголовочные **.h*-файлы современными заголовками. Помните, что старые заголовочные **.h*-файлы помещают свое содержимое в глобальное пространство имен, а современные заголовки — в пространство имен *std*.

Указатели на функции

Указатель на функцию ссылается на входную точку этой функции.

Указатель на функцию— это довольное сложное, но очень мощное средство C++. Несмотря на то что функция не является переменной, она, тем не менее, занимает физическую область памяти, некоторый адрес которой можно присвоить указателю. Адрес, присваиваемый указателю, является входной точкой функции. (Именно этот адрес используется при вызове функции.) Если некоторый указатель ссылается на функцию, то ее (функцию) можно вызвать с помощью этого указателя.

Указатели на функции также позволяют передавать функции в качестве аргументов другим функциям. Адрес функции можно получить, используя имя функции без круглых скобок и аргументов. (Этот процесс подобен получению адреса массива, когда также используется только его имя без индекса.) Если присвоить адрес функции указателю, то эту функцию можно вызвать через указатель. Например, рассмотрим следующую программу. Она содержит две функции, *vline()* и *hline()*, которые рисуют на экране вертикальные и горизонтальные линии заданной длины.

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

void vline(int i), hline(int i);

int main()
{
  void (*p)(int i);

  p = vline; // указатель на функцию vline()
  (*p)(4); // вызов функции vline()
  p = hline; // указатель на функцию hline()
  (*p)(3); // вызов функции hline()

  return 0;
}

void hline(int i)
{
  for( ; i; i--) cout << "-";
```

```
  for( ; i; i--) cout <<  - ;
  cout << "\n";
}

void vline(int i)
{
  for( ; i; i--) cout << "|\n";
}
```

Вот как выглядят результаты выполнения этой программы.

```
I
I
I
I
- - -
```

Рассмотрим эту программу в деталях. В первой строке тела функции main() объявляется переменная *p* как указатель на функцию, которая принимает один целочисленный аргумент и не возвращает никакого значения. Это объявление не определяет, какая функция имеется в виду. Оно лишь создает указатель, который можно использовать для адресации любой функции этого типа. Необходимость круглых скобок, в которые заключен указатель **p*, следует из C++-правил предшествования.

В следующей строке указателю *p* присваивается адрес функции *vline()*. Затем выполняется вызов функции *vline()* с аргументом *4*. После этого указателю *p* присваивается адрес функции *hline()*, и с помощью этого указателя реализуется ее вызов.

В этой программе при вызове функций посредством указателя используется следующий формат:

```
(*p) (4);
```

Однако функцию, адресуемую указателем *p*, можно вызвать с использованием более простого синтаксиса:

```
p (4);
```

Единственная причина, по которой чаще используется первый вариант вызова функции, состоит в том, что всем, кто станет разбирать вашу программу, станет ясно, что здесь реализован вызов функции через указатель *p*, а не вызов функции с именем *p*. Во всем остальном эти варианты эквивалентны.

Несмотря на то что в предыдущем примере указатель на функцию используется только ради иллюстрации, зачастую такое его применение играет очень важную роль. Указатель на функцию позволяет передавать ее адрес другой функции. В качестве показательного примера можно привести функцию *qsort()* из стандартной C++-библиотеки. Функция *qsort()* — это функция быстрой сортировки, основанная на алгоритме *Quicksort*, который упорядочивает содержимое массива. Вот как выглядит ее прототип.

```
void qsort(void * start, size_t length, size_t size, int (*compare)
(const void *, const void *));
```

Функция qsort() — *это функция сортировки из стандартной C++-библиотеки*.

Прототип функции *qsort() "прописан"* в заголовке *<cstdlib>*, в котором также определен тип *size_t* (как тип unsigned int). Чтобы использовать функцию *qsort()*, необходимо передать ей указатель на начало массива объектов, который вы хотите отсортировать (параметр *start*), длину этого массива (параметр *length*), размер в байтах каждого элемента (параметр *size*) и указатель на функцию сравнения элементов массива (параметр **compare*).

Функция сравнения, используемая функциеи *qsort()*, сопоставляя два элемента массива, должна возвратить отрицательное значение, если ее первый аргумент указывает на значение, которое меньше второго, нуль, если эти аргументы равны, и положительное значение, если первый аргумент указывает на значение, которое больше второго.

Чтобы понять, как можно использовать функцию *qsort()*, рассмотрим следующую программу.

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

int comp(const void *a, const void *b);

int main( )
{
  char str[] = "Указатели на функции дают гибкость.";

  qsort(str, strlen(str), 1, comp);

  cout << "Отсортированная строка: " << str;

  return 0;
}

int comp(const void *a, const void *b)
{
  return * (char *) a - * (char *) b;
}
```

Вот как выглядят результаты выполнения этой программы.

```
Отсортированная строка: Уаааабгдезиииикккклнностттуфцью
```

Эта программа сортирует строку *str* в возрастающем порядке. Поскольку функции *qsort()* передается вся необходимая ей информация, включая указатель на функцию сравнения, ее можно использовать для сортировки данных любого типа. Например, следующая программа сортирует массив целых чисел. Для гарантии переносимости при определении размера целочисленного значения в ней используется оператор *sizeof*.

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

int comp(const void *a, const void *b);

int main( )
{
  int num[] = {10, 4, 3, 6, 5, 7, 8};
  int i;

  qsort(num, 7, sizeof(int), comp);
  for(i=0; i<7; i++)
    cout << num[i] << ' ';
```

```
    cout << num[i] << "  ";

  return 0;
}

int comp(const void *a, const void *b)
{
  return * (int *) a - * (int *) b;
}
```

Не стану утверждать, что указатели на функции не так просты для понимания, но практика поможет и *"с ними найти общий язык"*. В отношении указателей на функции необходимо рассмотреть еще один аспект, связанный с перегруженными функциями.

Как найти адрес перегруженной функции

Получить адрес перегруженной функции немного сложнее, чем найти адрес обычной "одиночной" функции. Если же существует несколько версий перегруженной функции, то должен существовать механизм, который бы определял, адрес какой именно версии мы получаем. При получении адреса перегруженной функции именно способ объявления указателя определяет, адрес какой ее версии будет получен. По сути, объявление указателя в этом случае сравнивается с соответствующими объявлениями указателей перегруженных функций. Функция, объявление которой обнаружит совпадение, и будет той функцией, адрес которой мы получили.

В следующем примере программы содержится две версии функции *space()*. Первая версия выводит на экран *count* пробелов, а вторая— *count* символов, переданных в качестве аргумента *ch*. В функции main() объявляются два указателя на функции. Первый задан как указатель на функцию с одним целочисленным параметром, а второй — как указатель на функцию с двумя параметрами.

```
/* Использование указателей на перегруженные функции.
*/
#include <iostream>
using namespace std;

// Вывод на экран count пробелов.
void space(int count)
{
  for( ; count; count--) cout << ' ';
}

// Вывод на экран count символов, переданных в ch.
void space(int count, char ch)
{
  for( ; count; count--) cout << ch;
}

int main( )
{
  /* Создание указателя на void-функцию с одним int-параметром. */
  void (*fp1) (int);

  /* Создание указателя на void-функцию с одним int-параметром и одним
параметром типа char. */
```

```
параметров типа char. */
  void (*fp2)(int, char);

  fp1 = space; // получаем адрес функции space(int)
  fp2 = space; // получаем адрес функции space(int, char)

  fp1(22); // Выводим 22 пробела (этот вызов аналогичен вызову (* fp1) (22)) .
  cout << "|\n";

  fp2(30, 'x'); // Выводим 30 символов "x" (этот вызов аналогичен вызову (*fp2) (30, 'x').
  cout << "|\n";

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
                    I
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX I
```

Как отмечено в комментариях, компилятор способен определить, адрес какой перегруженной функции он получает, на основе того, как объявлены указатели *fp1* и *fp2*.

Итак, когда адрес перегруженной функции присваивается указателю на функцию, то именно это объявление указателя служит основой для определения того, адрес какой функции был присвоен. При этом объявление указателя на функцию должно соответствовать одной (и только одной) из перегруженных функций. В противном случае в программу вносится неоднозначность, которая вызовет ошибку компиляции.

Статические члены класса

Один статический член класса разделяется всеми объектами класса.

Ключевое слово *static* можно применять и к членам класса. Объявляя член класса статическим, мы тем самым уведомляем компилятор о том, что независимо от того, сколько объектов этого класса будет создано, существует только одна копия этого *static*-члена. Другими словами, *static*-член разделяется всеми объектами класса. Все статические данные при первом создании объекта инициализируются нулевыми значениями, если не представлено других значений инициализации.

При объявлении статического члена данных в классе программист не должен его определять. Необходимо обеспечить его глобальное определение вне этого класса. Это реализуется путем повторного объявления этой статической переменной с помощью оператора разрешения области видимости, который позволяет идентифицировать, к какому классу она принадлежит. Только в этом случае для этой статической переменной будет выделена память.

Рассмотрим пример использования *static*-члена класса. Изучите код этой программы и постарайтесь понять, как она работает.

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

class ShareVar {
   static int num;
  public:
   void setnum(int i) { num = i; };
```

```
    void shownum( ) { cout << num << " "; }
};

int ShareVar::num; // определяем static-член num

int main( )
{
  ShareVar a, b;

  a.shownum( ); // выводится 0
  b.shownum( ); // выводится 0
  a.setnum(10); // устанавливаем static-член num равным 10
  a.shownum( ); // выводится 10
  b.shownum( ); // также выводится 10

  return 0;
}
```

Обратите внимание на то, что статический целочисленный член *num* объявлен и в классе *ShareVar*, и определен в качестве глобальной переменной. Как было заявлено выше, необходимость такого двойного объявления вызвана тем, что при объявлении члена *num* в классе *ShareVar* память для него не выделяется. C++ инициализирует переменную *num* значением *0*, поскольку никакой другой инициализации в программе нет. Поэтому в результате двух первых вызовов функции *shownum()* для объектов *a* и *b* отображается значение *0*. Затем объект *a* устанавливает член *num* равным *10*, после чего объекты *a* и *b* снова выводят на экран его значение с помощью функции *shownum()*. Но так как существует только одна копия переменной *num*, разделяемая объектами *a* и *b*, значение *10* будет выведено при вызове функции *shownum()* для обоих объектов.

Узелок на память. *При объявлении члена класса статическим вы обеспечиваете создание только одной его копии, которая будет совместно использоваться всеми объектами этого класса.*

Если *static*-переменная является открытой (т.е. public-переменной), к ней можно обращаться напрямую через имя ее класса, без ссылки на какой-либо конкретный объект. (Безусловно, обращаться можно также и через имя объекта.) Рассмотрим, например, эту версию класса *ShareVar*.

```
class ShareVar {
  public:
    static int num;
    void setnum(int i) { num = i; };
    void shownum( ) { cout << num << " "; }
};
```

В данной версии переменная *num* является public-членом данных. Это позволяет нам обращаться к ней напрямую, как показано в следующий инструкции.

```
ShareVar::num = 100;
```

Здесь значение переменной *num* устанавливается независимо от объекта, а для обращения к ней достаточно использовать имя класса и оператор разрешения области видимости. Более того, эта инструкция законна даже до создания каких-либо объектов типа *ShareVar*! Таким образом, получить или установить значение *static*-члена класса можно до того, как будут созданы какие-либо объекты.

И [illegible] *static*

и хотя вы, возможно, пока не почувствовали неоходимости в *static*-членах класса, по мере программирования на C++ вам придется столкнуться с ситуациями, когда они окажутся весьма полезными, позволив избежать применения глобальных переменных.

Можно также объявить статической и функцию-член, но это — нераспространенная практика. К статической функции-члену могут получить доступ только другие *static*-члены этого класса. (Конечно же, статическая функция-член может получать доступ к нестатическим глобальным данным и функциям.) Статическая функция-член не имеет указателя *this*. Создание виртуальных статических функций-членов не разрешено. Кроме того, их нельзя объявлять с модификаторами *const* или *volatile*. Статическую функцию-член можно вызвать для объекта ее класса или независимо от какого бы то ни было объекта, а для обращения к ней достаточно использовать имя класса и оператор разрешения области видимости.

Применение к функциям-членам модификаторов const и mutable

Константная (const-) функция-член не может модифицировать объект, который ее вызвал.

Функции-члены класса могут быть объявлены с использованием модификатора *const*. Это означает, что с указателем *this* в этом случае необходимо обращаться как с *const*-указателем. Другими словами, *const*-функция не может модифицировать объект, для которого она вызвана. Кроме того, *const*-объект не может вызвать не *const*-функцию-член. Но *const*-функцию-член могут вызывать как *const*-, так и *не const*-объекты.

Чтобы определить функцию как *const*-член класса, используйте формат, представленный в следующем примере.

```
class X {
    int some_var;
  public:
    int f1() const; // const-функция-член
};
```

Как видите, модификатор *const* располагается после объявления списка параметров функции.

Цель объявления функции как *const*-члена — не допустить модификацию объекта, который ее вызывает. Например, рассмотрим следующую программу.

```
/* Демонстрация использования const-функций-членов. Эта программа не скомпилируется.
*/
#include <iostream>
using namespace std;

class Demo {
    int i;
  public:
    int geti() const {
      return i; // все в порядке
    }
    void seti (int x) const { i = x; // ошибка! }
};

int main()
{
  Demo ob;
```

```
  ob.seti(1900);
  cout << ob.geti();

  return 0;
}
```

Эта программа не скомпилируется, поскольку функция *seti()* объявлена как *const*-член. Это означает, что ей не разрешено модифицировать вызывающий объект. Ее попытка изменить содержимое переменной *i* приводит к возникновению ошибки. В отличие от функции *seti()*, функция *geti()* не пытается модифицировать переменную *i*, и потому она совершенно приемлема.

Возможны ситуации, когда нужно, чтобы *const*-функция могла изменить один или несколько членов класса, но никак не могла повлиять на остальные. Это можно реализовать с помощью модификатора *mutable*, который переопределяет атрибут функции *const*. Другими словами, *mutable*-член может быть модифицирован *const*-функцией-членом. Рассмотрим пример.

```
/* Демонстрация использования модификатора mutable.
*/
#include <iostream>
using namespace std;

class Demo {
    mutable int i;
    int j;
  public:
    int geti() const {
      return i; // все в порядке
    }

    void seti(int x) const {
      i = x; // теперь все в порядке
    }

    /* Следующая функция не скомпилируется,
      void setj (int x) const {
        j = x; // Это по-прежнему неверно!
      }
    */
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();

  return 0;
```

```
}
```

Здесь член *i* определен с использованием модификатора *mutable*, поэтому его можно изменить с помощью функции *seti()*. Однако переменная *j* не является *mutable*-членом, поэтому функции *setj()* не разрешено модифицировать его значение.

Использование explicit-конструкторов

Для создания "неконвертирующего" конструктора используйте спецификатор explicit.

В C++ определено ключевое слово *explicit*, которое применяется для обработки специальных ситуаций, когда используются конструкторы определенных типов. Чтобы понять назначение спецификатора *explicit*, рассмотрим следующую программу.

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

class myclass {
   int a;
  public:
   myclass(int x) { a = x; }
   int geta( ) { return a; }
};

int main( )
{
  myclass ob(4);
  cout << ob.geta( );

  return 0;
}
```

Здесь конструктор класса *myclass* принимает один параметр. Обратите внимание на то, как объявлен объект *ob* в функции main(). Значение 4, заданное в круглых скобках после имени *ob*, представляет собой аргумент, который передается параметру *x* конструктора *myclass()*, а параметр *x* в свою очередь используется для инициализации члена *a*. Именно таким способом мы инициализируем члены класса с начала этой книги. Однако существует и альтернативный вариант инициализации. Например, при выполнении следующей инструкции член *a* также получит значение *4*.

```
myclass ob = 4; /* Этот формат инициализации автоматически преобразуется
в формат myclass(4). */
```

Как отмечено в комментарии, этот формат инициализации автоматически преобразуется в вызов конструктора класса *myclass*, а число *4* используется в качестве аргумента. Другими словами, предыдущая инструкция обрабатывается компилятором так, как если бы она была записана:

```
myclass ob(4);
```

В общем случае всегда, когда у вас есть конструктор, который принимает только один аргумент, для инициализации объекта можно использовать любой из форматов: либо *ob(x)*, либо *ob=x*. Дело в том, что при создании конструктора класса с одним аргументом вами неявно создается преобразование из типа аргумента в тип этого класса.

Если вам не нужно, чтобы такое неявное преобразование имело место, можно предотвратить его с помощью спецификатора *explicit*. Ключевое слово *explicit* применяется только к конструкторам. Конструктор, определенный с помощью спецификатора *explicit*, будет задействован только в том случае, если для инициализации членов класса используется обычный синтаксис конструктора. Никаких автоматических преобразований выполнено не будет. Например, объявляя конструктор класса *myclass* с использованием

спецификатора *explicit*, мы тем самым отменяем поддержку автоматического преобразования типов. В этом варианте определения класса функция *myclass()* объявляется как *explicit*-конструктор.

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

class myclass {
   int a;
  public:
   explicit myclass(int x) { a = x; }
   int geta() { return a; }
};
```

Теперь будут разрешены к применению только конструкторы, заданные в таком формате.

```
myclass ob(110);
```

Чем интересно неявное преобразование конструктора

Автоматическое преобразование из типа аргумента конструктора в вызов конструктора само по себе имеет интересные последствия. Рассмотрим, например, следующий код.

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

class myclass {
   int num;
  public:
   myclass(int i) { num = i; }
   int getnum() { return num; }
};

int main()
{
  myclass o(10);

  cout << o.getnum() << endl; // отображает 10

  /* Теперь используем неявное преобразование для присвоения нового значения. */
  o = 1000;

  cout << o.getnum() << endl; // отображает 1000
  return 0;
}
```

Обратите внимание на то, что новое значение присваивается объекту *o* с помощью такой инструкции:

```
o = 1000;
```

Использование данного формата возможно благодаря неявному преобразованию из типа *int* в тип *myclass*, которое создается конструктором *myclass()*. Конечно же, если бы конструктор *myclass()* был объявлен с помощью спецификатора *explicit*, то предыдущая инструкция не могла бы выполниться.

инструкция не могла бы выполниться.

Синтаксис инициализации членов класса

В примерах программ из предыдущих глав члены данных получали начальные значения в конструкторах своих классов. Например, следующая программа содержит класс *myclass*, который включает два члена данных *numA* и *numB*. Эти члены инициализируются в конструкторе *myclass()*.

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

class myclass {
   int numA;
   int numB;
  public:
     /* Инициализируем члены numA и numB в конструкторе myclass(), используя обычный синтаксис. */
   myclass(int x, int y) { numA = x; numA = y;}
   int getNumA( ) { return numA; }
   int getNumB( ) { return numB; }
};

int main( )
{
  myclass ob1(7, 9), ob2 (5, 2);
  cout << "Значения членов данных объекта ob1 равны " << ob1.getNumB( ) << " и " << ob1.getNumA( ) << endl;

  cout << "Значения членов данных объекта ob2 равны " << ob2.getNumB( ) << " и " << ob2.getNumA( ) << endl;

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Значения членов данных объекта ob1 равны 9 и 7
Значения членов данных объекта ob2 равны 2 и 5
```

Присвоение начальных значений членам данных *numA* и *numB* в конструкторе, как это делается в конструкторе *myclass()*, — обычная практика, которая применяется для многих классов. Но этот метод годится не для всех случаев. Например, если бы члены *numA* и *numB* были заданы как *const*-переменные, т.е. таким образом:

```
class myclass {
  const int numA; // const-член
  const int numB; // const-член
};
```

то им нельзя было бы присвоить значения с помощью конструктора класса *myclass*, поскольку *const*-переменные должны быть инициализированы однократно, после чего им уже нельзя придать другие значения. Подобные проблемы возникают при использовании ссылочных членов, которые должны быть инициализированы, и при использовании членов класса, которые не имеют конструкторов по умолчанию. Для решения проблем такого рода в C++ предусмотрена поддержка альтернативного синтаксиса инициализации членов класса, который позволяет присваивать им начальные значения при создании объекта

класса, который позволяет присваивать им начальные значения при создании объекта класса.

Синтаксис инициализации членов класса аналогичен тому, который используется для вызова конструктора базового класса. Вот как выглядит общий формат такой инициализации.

```
constructor(список_аргументов):
  член1(инициализатор),
  член2(инициализатор),
  // ...
  членN (инициализатор)
{
  // тело конструктора
}
```

Члены, подлежащие инициализации, указываются после конструктора класса, и отделяются от имени конструктора и списка его аргументов двоеточием. При этом в одном и том же списке можно смешивать обращения к конструкторам базового класса с инициализацией членов.

Ниже представлена предыдущая программа, но переделанная так, чтобы члены *numA* и *numB* были объявлены с использованием модификатора *const*, и получали свои начальные значения с помощью альтернативного синтаксиса инициализации членов класса.

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

class myclass {
   const int numA; // const-член
   const int numB; // const-член
  public:
   /* Инициализируем члены numA и numB с использованием альтернативного синтаксиса инициализации. */
   myclass(int x, int y) : numA(x), numB(y) { }

   int getNumA() { return numA; }
   int getNumB() { return numB; }
};

int main()
{
  myclass ob1 (7, 9), ob2(5, 2);

  cout << "Значения членов данных объекта ob1 равны " << ob1.getNumB() << " и " << ob1.getNumA()<< endl;

  cout << "Значения членов данных объекта ob2 равны " << ob2.getNumB() << " и " << ob2.getNumA()<< endl;

  return 0;
}
```

Эта программа генерирует такие же результаты, как и ее предыдущая версия. Однако

Эта программа генерирует такие же результаты, как и ее предыдущая версия. Однако обратите внимание на то, как инициализированы члены *numA* и *numB*.

```
myclass(int x, int y) : numA(x), numB(y) { }
```

Здесь член *numA* инициализируется значением, переданным в аргументе *x*, а член *numB* — значением, переданным в аргументе *y*. И хотя члены *numA* и *numB* сейчас определены как *const*-переменные, они могут получить свои начальные значения при создании объекта класса *myclass*, поскольку здесь используется альтернативный синтаксис инициализации членов класса.

Использование ключевого слова asm

С помощью ключевого слова asm в C++-программу встраивается код, написанный на языке ассемблера.

Несмотря на то что C++ — всеобъемлющий и мощный язык программирования, возможны ситуации, обработка которых для него оказывается весьма затруднительной. (Например, в C++ не предусмотрена инструкция, которая могла бы запретить прерывания.) Чтобы справиться с подобными специальными ситуациями, C++ предоставляет средство, которое позволяет войти в код, написанный на языке ассемблера, совершенно игнорируя C++-компилятор. Этим средством и является инструкция *asm*, используя которую можно встроить ассемблерный код непосредственно в C++-программу. Этот код скомпилируется без каких-либо изменений и станет частью кода вашей программы, начиная с места нахождения инструкции *asm*.

Общий формат использования ключевого слова *asm* имеет следующий вид.

```
asm ("код");
```

Здесь элемент *код* означает инструкцию, написанную на языке ассемблера, которая будет встроена в программу. При этом некоторые компиляторы также позволяют использовать и другие форматы записи инструкции asm.

```
asm инструкция;

asm инструкция newline

asm {
  последовательность инструкций
}
```

Здесь элемент *инструкция* означает любую допустимую инструкцию языка ассемблера. Поскольку использование инструкции *asm* зависит от конкретной реализации среды программирования, то за подробностями обратитесь к документации, прилагаемой к вашему компилятору.

На момент написания этой книги в среде Visual C++ (Microsoft) для встраивания кода, написанного на языке ассемблера, предлагалось использовать инструкцию *__asm*. Во всём остальном этот формат аналогичен описанию инструкции *asm*.

Осторожно! *Для использования инструкции asm необходимо обладать доскональными знаниями языка ассемблера. Если вы не считаете себя специалистом по этому языку, то лучше пока избегать использования инструкции asm, поскольку неосторожное ее применение может вызвать тяжелые последствия для вашей системы*.

Спецификация компоновки

Спецификатор компоновки позволяет определить способ компоновки функции.

В C++ можно определить, как функция связывается с вашей программой. По умолчанию функции компонуются как C++-функции. Но, используя спецификацию компоновки, можно обеспечить компоновку функций, написанных на других языках программирования. Общий формат спецификатора компоновки выглядит так:

```
extern "язык" прототип функции
```

```
extern "язык" прототип_функции
```

Здесь элемент *язык* означает нужный язык программирования. Все C++-компиляторы поддерживают как C-, так и C++-компоновку. Некоторые компиляторы также позволяют использовать спецификаторы компоновки для таких языков, как *Fortran*, *Pascal* или *BASIC*. (Эту информацию необходимо уточнить в документации, прилагаемой к вашему компилятору.)

Следующая программа позволяет скомпоновать функцию *myCfunc()* как C-функцию.

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

extern "C" void myCfunc();

int main()
{
  myCfunc();
  return 0;
}

// Эта функция будет скомпонована как C-функция.
void myCfunc() {
  cout << "Эта функция скомпонована как C-функция.\n";
}
```

На заметку. *Ключевое слово extern — необходимая составляющая спецификации компоновки. Более того, спецификация компоновки должна быть глобальной; ее нельзя использовать в теле какой-либо функции.*

Используя следующий формат спецификации компоновки, можно задать не одну, а сразу несколько функций.

```
extern "язык" {
  прототипы_функций
}
```

Спецификации компоновки используются довольно редко, и вам, возможно, никогда не придется их применять. Основное их назначение — позволить применение в C++-программах кода, написанного сторонними организациями на языках, отличных от C++.

Операторы указания на члены ".*" и "->*"

Операторы указания на член позволяют получить доступ к члену класса через указатель на этот член.

В C++ предусмотрена возможность сгенерировать указатель специального типа, который *"ссылается"* не на конкретный экземпляр члена в объекте, а на член класса вообще. Указатель такого типа называется *указателем на член класса* (pointer-to-member). Это — не обычный C++-указатель. Этот специальный указатель обеспечивает только соответствующее смещение в объекте, которое позволяет обнаружить нужный член класса. Поскольку указатели на члены — не настоящие указатели, к ним нельзя применять операторы *"."* и *"->"*. Для получения доступа к члену класса через указатель на член необходимо использовать специальные операторы *".*"* и *"->*"*.

Если идея, изложенная в предыдущем абзаце, вам показалась немного *"туманной"*, то следующий пример поможет ее прояснить. При выполнении этой программы отображается сумма чисел от *1* до *7*. Здесь доступ к членам класса *myclass* (функции *sum_it()* и переменной *sum*) реализуется путем использования указателей на члены.

```
// Пример использования указателей на члены класса.
#include <iostream>
using namespace std;

class myclass {
  public:
    int sum;
    void myclass::sum_it(int x);
};

void myclass::sum_it(int x) {
  int i;
  sum = 0;
  for(i=x; i; i--) sum += i;
}

int main( )
{
  int myclass::*dp; // указатель на int-член класса
  void (myclass::*fp)(int x); // указатель на функцию-член
  myclass c;

  dp = &myclass::sum; // получаем адрес члена данных
  fp = &myclass::sum_it; // получаем адрес функции-члена

  (c.*fp)(7); // вычисляем сумму чисел от 1 до 7
  cout << "Сумма чисел от 1 до 7 равна " << c.*dp;

  return 0;
}
```

Результат выполнения этой программы таков.

```
Сумма чисел от 1 до 7 равна 28
```

В функции main() создается два члена-указателя: *dp* (для указания на переменную sum) и *fp* (для указания на функцию sum_it()). Обратите внимание на синтаксис каждого объявления. Для уточнения класса используется оператор разрешения контекста (оператор разрешения области видимости). Программа также создает объект типа *myclass* с именем *c*.

Затем программа получает адреса переменной *sum* и функции *sum_it()* и присваивает их указателям *dp* и *fp* соответственно. Как упоминалось выше, эти адреса в действительности представляют собой лишь смещения в объекте типа *myclass*, по которым можно найти переменную *sum* и функцию *sum_it()*. Затем программа использует указатель на функцию *fp*, чтобы вызвать функцию *sum_it()* для объекта *c*. Наличие дополнительных круглых скобок объясняется необходимостью корректно применить оператор ".*". Наконец, программа отображает значение суммы чисел, получая доступ к переменной *sum* объекта *c* через указатель *dp*.

При доступе к члену объекта с помощью объекта или ссылки на него необходимо использовать оператор ".*". Но если для этого используется указатель на объект, нужно использовать оператор "->*", как показано в этой версии предыдущей программы.

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

```
class myclass {
  public:
    int sum;
    void myclass::sum_it(int x);
};

void myclass::sum_it(int x) {
  int i;
  sum = 0;
  for(i=x; i; i--) sum += i;
}

int main()
{
  int myclass::*dp; // указатель на int-член класса
  void (myclass::*fp)(int x); // указатель на функцию-член
  myclass *c, d; // член с сейчас -- указатель на объект

  c = &d; // присваиваем указателю с адрес объекта

  dp = &myclass::sum; // получаем адрес члена данных sum
  fp = &myclass::sum_it; // получаем адрес функции sum_it()

   (c->*fp) (7); // Теперь используем оператор для вызова функции sum_it().
  cout << "Сумма чисел от 1 до 7 равна " << c->*dp; // ->*

  return 0;
}
```

В этой версии переменная *c* объявляется как указатель на объект типа *myclass*, а для доступа к члену данных *sum* и функции-члену *sum_it()* используется оператор *"->*"*.

Помните, что операторы указания на члены класса предназначены для специальных случаев, и их не стоит использовать для решения обычных повседневных задач программирования.

Создание функций преобразования

Функция преобразования автоматически преобразует тип класса в другой тип.

Иногда возникает необходимость в одном выражении объединить созданный программистом класс с данными других типов. Несмотря на то что перегруженные операторные функции могут обеспечить использование смешанных типов данных, в некоторых случаях все же можно обойтись простым преобразованием типов. И тогда, чтобы преобразовать класс в тип, совместимый с типом остальной части выражения, можно использовать функцию преобразования типа. Общий формат функции преобразования типа имеет следующий вид.

```
operator type( ) {return value;}
```

Здесь элемент *type* — новый тип, который является целью нашего преобразования, а элемент *value*— значение после преобразования. Функция преобразования должна быть членом класса, для которого она определяется.

Чтобы проиллюстрировать создание функции преобразования, воспользуемся классом *three_d* еще раз. Предположим, что нам нужно иметь средство преобразования объекта типа *three_d* в целочисленное значение, которое можно использовать в целочисленном выражении. Более того, такое преобразование должно происходить с использованием произведения значений трех координат. Для реализации этого мы будем использовать функцию преобразования, которая выглядит следующим образом,

```
operator int( ) { return x * y * z; }
```

Теперь рассмотрим программу, которая иллюстрирует работу функции преобразования.

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

class three_d {
   int x, y, z; // 3-мерные координаты
  public:
   three_d(int a, int b, int c) { x = a; y = b; z = c; }

   three_d operator+(three_d op2);
   friend ostream &operator<<(ostream &stream, three_d &obj);

   operator int( ) {return x * y * z; }
};

/* Отображение координат X, Y, Z - функция вывода данных для класса three_d.
*/
ostream &operator<<(ostream &stream, three_d &obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";

  return stream;
}

three_d three_d::operator+(three_d op2)
{
  three_d temp(0, 0, 0);
  temp.x = x+op2.x;
  temp.y = y+op2.y;
  temp.z = z+op2.z;

  return temp;
}

int main( )
{
  three_d a(1, 2, 3), b(2, 3, 4);
  cout << a << b;
```

```
  cout << b+100; /* Отображает число 124, поскольку здесь выполняется
преобразование объекта класса в значение типа int. */
  cout << "\n";

  a = a + b; // Сложение двух объектов класса three_d выполняется без
преобразования типа.
  cout << a; // Отображает координаты 3, 5, 7

  return 0;
}
```

Эта программа генерирует такие результаты.

```
1, 2, 3
2, 3, 4
124
3, 5, 7
```

Как подтверждают результаты выполнения этой программы, если в таком выражении целочисленного типа, как *cout<<b+100*, используется объект типа *three_d*, к этому объекту применяется функция преобразования. В данном случае функция преобразования возвращает значение *24*, которое затем участвует в операции сложения с числом *100*. Но когда в преобразовании нет необходимости, как при вычислении выражения *a=a+b*, функция преобразования не вызывается.

Если функция преобразования создана, то она будет вызываться везде, где требуется преобразование, включая ситуации, когда объект передается функции в качестве аргумента. Например, если объект класса *three_d* передать стандартной функции *abs()*, также будет вызвана функция, выполняющая преобразование объекта типа *three_d* в значение типа *int*, поскольку функция *abs()* должна принимать аргумент целочисленного типа.

Узелок на память. *Для различных ситуаций можно создавать различные функции преобразования. Например, можно определить функции, которые преобразуют объекты типа three_d в значения типа double или long, при этом созданные функции будут применяться автоматически. Функции преобразования позволяют интегрировать новые типы классов, создаваемые программистом, в C++-среду программирования.*

Глава 21: Введение в стандартную библиотеку шаблонов

Материал этой главы составляет то, что многие считают самым важным добавлением в C++ за последние годы. Речь идет о стандартной библиотеке шаблонов (*Standard Template Library — STL*). Именно включение библиотеки STL в C++ было основным событием, которое обсуждалось в период стандартизации C++. Библиотека STL предоставляет шаблонные классы и функции общего назначения, которые реализуют многие популярные и часто используемые алгоритмы и структуры данных. Например, она включает поддержку векторов, списков, очередей и стеков, а также определяет различные функции, обеспечивающие к ним доступ. Поскольку библиотека STL состоит из шаблонных классов, алгоритмы и структуры данных могут быть применены к данным практически любого типа.

Библиотека STL — *это набор шаблонных классов и функций общего назначения.*

Библиотека *STL* — это результат разработки программного обеспечения, который вобрал в себя одни из самых сложных средств языка C++. Чтобы понимать содержимое библиотеки STL и уметь им пользоваться, необходимо освоить весь материал, изложенный в предыдущих главах. Особенно хорошо нужно ориентироваться в шаблонах. Откровенно говоря, шаблонный синтаксис, который описывает STL, может поначалу испугать, хотя он выглядит сложнее, чем есть на самом деле. Несмотря на то что материал этой главы не труднее остального в этой книге, не следует огорчаться, если что-то на первый взгляд вам покажется непонятным. Немного терпения при рассмотрении примеров — и вскоре вы поймете, что за непривычным синтаксисом скрывается строгая простота STL.

STL — довольно большая библиотека, и все ее средства невозможно изложить в одной главе. Полное описание библиотеки STL со всеми ее нюансами и методами программирования заняло бы целую книгу. Цель представленного здесь обзора — познакомить вас с основными операциями, принципами проектирования и основами STL-программирования. Освоив материал этой главы, вы сможете легко изучить остальную часть библиотеки STL самостоятельно.

В этой главе также описан еще один важный класс C++ — класс *string*. Он предназначен для определения строкового типа данных, который позволяет обрабатывать символьные строки во многом подобно тому, как обрабатываются данные других типов: с помощью операторов. Класс *string* тесно связан с библиотекой *STL*.

Обзор STL

Несмотря на большой размер стандартной библиотеки шаблонов и порой пугающий синтаксис, в действительности ее средства довольно легко использовать, если понять, как она построена и из каких элементов состоит. Поэтому, прежде чем переходить к рассмотрению примеров, познакомимся с основными составляющими STL.

Ядро стандартной библиотеки шаблонов включает три основных элемента: *контейнеры*,

алгоритмы и итераторы. Они работают совместно один с другим, предоставляя тем самым готовые решения различных задач программирования.

Контейнеры — *это объекты, которые содержат другие объекты*.

Контейнеры — это объекты, содержащие другие объекты. Существует несколько различных типов контейнеров. Например, класс *vector* определяет динамический массив, класс *queue* создает двустороннюю очередь, а класс *list* обеспечивает работу с линейным списком. Эти контейнеры называются *последовательными контейнерами* и являются базовыми в STL. Помимо базовых, библиотека STL определяет *ассоциативные контейнеры*, которые позволяют эффективно находить нужные значения на основе заданных ключевых значений (ключей). Например, класс *map* обеспечивает хранение пар "ключ-значение" и предоставляет возможность находить значение по заданному уникальному ключу.

Каждый контейнерный класс определяет набор функций, которые можно применять к данному контейнеру. Например, контейнер списка включает функции, предназначенные для выполнения вставки, удаления и объединения элементов. А стек включает функции, которые позволяют помещать значения в стек и извлекать их из стека.

Алгоритмы обрабатывают содержимое контейнеров.

Алгоритмы обрабатывают содержимое контейнеров. Их возможности включают средства инициализации, сортировки, поиска и преобразования содержимого контейнеров. Многие алгоритмы работают с заданным диапазоном элементов контейнера.

Итераторы подобны указателям.

Итераторы — это объекты, которые в той или иной степени действуют подобно указателям. Они позволяют циклически опрашивать содержимое контейнера практически так же, как это делается с помощью указателя при циклическом опросе элементов массива. Существует пять типов итераторов.

Итераторы	Описание
Произвольного доступа (random access)	Сохраняют и считывают значения; позволяют организовать произвольный доступ к элементам контейнера
Двунаправленные (bidirectional)	Сохраняют и считывают значения; обеспечивают инкрементно-декрементное перемещение
Однонаправленные (forward)	Сохраняют и считывают значения; обеспечивают только инкрементное перемещение
Входные (input)	Считывают, но не записывают значения; обеспечивают только инкрементное перемещение
Выходные (output)	Записывают, но не считывают значения; обеспечивают только инкрементное перемещение

В общем случае итератор, который имеет большие возможности доступа, можно использовать вместо итератора с меньшими возможностями. Например, однонаправленным итератором можно заменить входной итератор.

Итераторы обрабатываются аналогично указателям. Их можно инкрементировать и декрементировать. К ним можно применять оператор разыменования адреса *. Итераторы объявляются с помощью типа *iterator*, определяемого различными контейнерами.

Библиотека STL поддерживает *реверсивные итераторы*, которые являются либо двунаправленными, либо итераторами произвольного доступа, позволяя перемещаться по последовательности в обратном направлении. Следовательно, если реверсивный итератор указывает на конец последовательности, то после инкрементирования он будет указывать на элемент, расположенный перед концом последовательности.

При ссылке на различные типы итераторов в описаниях шаблонов в этой книге будут использованы следующие термины.

Термин	Представляемый итератор
BiIter	Двунаправленный

	Двунаправленный
ForIter	Однонаправленный
InIter	Входной
OutIter	Выходной
RandIter	Произвольного доступа

STL опирается не только на контейнеры, алгоритмы и итераторы, но и на другие стандартные компоненты. Основными из них являются распределители памяти, предикаты и функции сравнения.

Распределитель памяти управляет выделением памяти для контейнера.

Каждый контейнер имеет свой *распределитель памяти* (allocator). Распределители управляют выделением памяти для контейнера. Стандартный распределитель — это объект класса *allocator*, но при необходимости (в специализированных приложениях) можно определять собственные распределители. В большинстве случаев стандартного распределителя вполне достаточно.

Предикат возвращает в качестве результата значение ИСТИНА/ЛОЖЬ.

Некоторые алгоритмы и контейнеры используют специальный тип функции, называемый *предикатом* (predicate). Существует два варианта предикатов: унарный и бинарный. Унарный предикат принимает один аргумент, а бинарный — два. Эти функции возвращают значения *ИСТИНА*/*ЛОЖЬ*, но точные условия, которые заставят их вернуть истинное или ложное значение, определяются программистом. В остальной части этой главы, когда потребуется унарная функция-предикат, на это будет указано с помощью типа *UnPred*. При необходимости использования бинарного предиката будет применяться тип *BinPred*. В бинарном предикате аргументы всегда расположены в порядке *первый*, *второй* относительно функции, которая вызывает этот предикат. Как для унарного, так и для бинарного предикатов аргументы должны содержать значения, тип которых совпадает с типом объектов, хранимых данным контейнером.

Функции сравнения сравнивают два элемента последовательности.

Некоторые алгоритмы и классы используют специальный тип бинарного предиката, который сравнивает два элемента. Функции сравнения возвращают значение *true*, если их первый аргумент меньше второго. Функции сравнения идентифицируются с помощью типа *Comp*.

Помимо заголовков, требуемых различными классами STL, стандартная библиотека C++ включает заголовки *<utility>* и *<functional>*, которые обеспечивают поддержку STL. Например, в заголовке *<utility>* определяется шаблонный класс *pair*, который может хранить пару значений. Мы будем использовать класс *pair* ниже в этой главе.

Шаблоны в заголовке *<functional>* позволяют создавать объекты, которые определяют функцию *operator()*. Эти объекты называются объектами-функциями, и их во многих случаях можно использовать вместо указателей на функции. Существует несколько встроенных объектов-функций, объявленных в заголовке *<functional>*. Приведем здесь некоторые из них.

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Возможно, наиболее широко используется объект-функция *less*, который определяет, при каких условиях один объект меньше другого. Объекты-функции можно использовать вместо реальных указателей на функции в алгоритмах STL, о которых пойдет речь ниже. Используя объекты-функции вместо указателей на функции, библиотека STL в некоторых случаях генерирует более эффективный программный код.

Материал этой главы не предусматривает использования объектов-функций, и мы не будем применять их напрямую. (Подробное описание объектов-функций можно найти в моей книге *Полный справочник по C++*, 4-е издание, М. : Издательский дом "Вильямс".)

Контейнерные классы

Как упоминалось выше, контейнеры представляют собой объекты STL, которые предназначены для хранения данных. Контейнеры, определяемые в STL, представлены в табл. 21.1. В ней также указаны заголовки, которые необходимо включать в программу при использовании каждого контейнера. Но несмотря на то, что класс *string* также является контейнером, позволяющим хранить и обрабатывать символьные строки, он в эту таблицу не включен и рассматривается ниже в этой главе.

Таблица 21.1. Контейнеры, определенные в STL

Контейнер	Описание	Заголовок
`bitset`	Битовое множество	`<bitset>`
`deque`	Дек (двусторонняя очередь, или очередь с двусторонним доступом)	`<deque>`
`list`	Линейный список	`<list>`
`map`	Отображение. Хранит пары "ключ-значение", в которых каждый ключ связан только с одним значением	`<map>`

Окончание табл. 21.1

Контейнер	Описание	Заголовок
`multimap`	Мультиотображение. Хранит пары "ключ-значение", в которых каждый ключ может быть связан с двумя или более значениями	`<map>`
`multiset`	Множество, в котором каждый элемент необязательно уникален (мультимножество)	`<set>`
`priority_queue`	Приоритетная очередь	`<queue>`
`queue`	Очередь	`<queue>`
`set`	Множество, в котором каждый элемент уникален	`<set>`
`stack`	Стек	`<stack>`
`vector`	Динамический массив	`<vector>`

Поскольку имена типов в объявлениях шаблонных классов произвольны, контейнерные классы объявляют *typedef*-версии этих типов, что конкретизирует имена типов. Некоторые из наиболее популярных *typedef*-имен приведены ниже.

`size_type`	Некоторый целый тип, приблизительно аналогичный типу `size_t`
`reference`	Ссылка на элемент
`const_reference`	Константная (`const`-) ссылка на элемент
`iterator`	Итератор
`const_iterator`	Константный (`const`-) итератор
`reverse_iterator`	Реверсивный итератор
`const_reverse_iterator`	Константный реверсивный итератор
`value_type`	Тип значения, хранимого в контейнере (то же самое, что и обобщенный тип `T`)
`allocator_type`	Тип распределителя (памяти)
`key_type`	Тип ключа
`key_compare`	Тип функции, которая сравнивает два ключа

mapped_type	Тип значения, сохраняемого в отображении (то же самое, что и обобщенный тип T)
value_compare	Тип функции, которая сравнивает два значения

Поскольку в одной главе невозможно рассмотреть все контейнеры, в следующих разделах мы представим только три из них: *vector*, *list* и *map*. Если вы поймете, как работают эти три контейнера, у вас не будет проблем с использованием остальных.

Векторы

Векторы представляют собой динамические массивы.

Одним из контейнеров самого широкого назначения является вектор. Класс *vector* поддерживает динамический массив, который при необходимости может увеличивать свой размер. Как вы знаете, в C++ размер массива фиксируется во время компиляции. И хотя это самый эффективный способ реализации массивов, он в то же время является и самым ограничивающим, поскольку размер массива нельзя изменять во время выполнения программы. Эта проблема решается с помощью вектора, который по мере необходимости обеспечивает выделение дополнительного объема памяти. Несмотря на то что вектор — это динамический массив, тем не менее, для доступа к его элементам можно использовать стандартное обозначение индексации массивов.

Вот как выглядит шаблонная спецификация для класса vector:

```
template <class T, class Allocator = allocator<T> > class vector
```

Здесь *T*— тип сохраняемых данных, а элемент *Allocator* означает распределитель памяти, который по умолчанию использует стандартный распределитель. Класс *vector* имеет следующие конструкторы.

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type num, const T &val = T(), const Allocator &a =
Allocator());

vector(const vector<T, Allocator> &ob);

template <class InIter> vector(InIter start, InIter end, const Allocator
&a = Allocator());
```

Первая форма конструктора предназначена для создания пустого вектора. Вторая создает вектор, который содержит *num* элементов со значением *val*, причем значение *val* может быть установлено по умолчанию. Третья форма позволяет создать вектор, который содержит те же элементы, что и заданный вектор *ob*. Четвертая предназначена для создания вектора, который содержит элементы в диапазоне, заданном параметрами-итераторами *start* и *end*.

Ради достижения максимальной гибкости и переносимости любой объект, который предназначен для хранения в векторе, должен определяться конструктором по умолчанию. Кроме того, он должен определять операции *"<"* и *"=="* Некоторые компиляторы могут потребовать определения и других операторов сравнения. (В виду существования различных реализаций для получения точной информации о требованиях, предъявляемых вашим компилятором, следует обратиться к прилагаемой к нему документации.) Все встроенные типы автоматически удовлетворяют этим требованиям.

Несмотря на то что приведенный выше синтаксис шаблона выглядит довольно "массивно", в объявлении вектора нет ничего сложного. Рассмотрим несколько примеров.

```
vector<int> iv; /* Создание вектора нулевой длины для хранения int-
значений. */

vector<char> cv(5); /* Создание 5-элементного вектора для хранения char-
значений. */

vector<char> cv(5, 'x'); /* Инициализация 5-элементного char-вектора. */
```

```
vector<int> iv2(iv); /* Создание int-вектора на основе int-вектора iv.
*/
```

Для класса vector определены следующие операторы сравнения:

```
==, <, <=, !=, > и >=
```

Для вектора также определен оператор индексации *"[]"*, который позволяет получить доступ к элементам вектора с помощью стандартной записи с использованием индексов. Функции-члены, определенные в классе *vector*, перечислены в табл. 21.2. Самыми важными из них являются *size()*, *begin()*, *end()*, *push_back()*, *insert()* и *erase()*. Очень полезна функция *size()*, которая возвращает текущий размер вектора, поскольку она позволяет определить размер вектора во время выполнения программы. Помните, что векторы при необходимости увеличивают свой размер, поэтому нужно иметь возможность определять его величину во время работы программы, а не только во время компиляции.

Таблица 21.2. Функции-члены, определенные в классе `vector`

Функция-член	Описание
`template <class InIter> void assign(InIter start, InIter end);`	Помещает в вектор последовательность, определяемую параметрами `start` и `end`
`void assign(size_type num, const T &val);`	Помещает в вектор `num` элементов со значением `val`
`reference at(size_type i); const_reference at(size_type i) const;`	Возвращает ссылку на элемент, заданный параметром `i`
`reference back(); const_reference back() const;`	Возвращает ссылку на последний элемент в векторе
`iterator begin(); const_iterator begin() const;`	Возвращает итератор для первого элемента в векторе
`size_type capacity() const;`	Возвращает текущую емкость вектора, или количество элементов, которое может храниться в векторе до того, как возникнет необходимость в выделении дополнительной памяти
`void clear();`	Удаляет все элементы из вектора
`bool empty() const;`	Возвращает истинное значение, если используемый вектор пуст, и ложное значение в противном случае
`const_iterator end() const; iterator end();`	Возвращает итератор для конца вектора
`iterator erase(iterator i);`	Удаляет элемент, адресуемый итератором `i`; возвращает итератор для элемента, расположенного после удаленного
`iterator erase(iterator start, iterator end);`	Удаляет элементы в диапазоне, задаваемом параметрами `start` и `end`; возвращает итератор для элемента, расположенного за последним удаленным элементом
`reference front(); const_reference front() const;`	Возвращает ссылку на первый элемент в векторе
`allocator_type get_allocator() const;`	Возвращает распределитель памяти вектора
`iterator insert( iterator i, const T &val = T());`	Вставляет значение `val` непосредственно перед элементом, заданным параметром `i`; возвращает итератор для этого элемента

Окончание табл. 21.2

Функция-член	Описание
`void insert(iterator i, size_type num, const T &val);`	Вставляет `num` копий значения `val` непосредственно перед элементом, заданным параметром `i`
`template <class InIter> void insert( iterator i, InIter start, InIter end);`	Вставляет в вектор последовательность элементов, определяемую параметрами `start` и `end`, непосредственно перед элементом, заданным параметром `i`
`size_type max_size() const;`	Возвращает максимальное число элементов, которое может содержать вектор
`reference operator[]( size_type i) const; const_reference operator[]( size_type i) const;`	Возвращает ссылку на элемент, заданный параметром `i`
`void pop_back();`	Удаляет последний элемент в векторе
`void push_back(const T &val);`	Добавляет в конец вектора элемент, значение которого задано параметром `val`
`reverse_iterator rbegin(); const_reverse_iterator rbegin() const;`	Возвращает реверсивный итератор для конца вектора
`reverse_iterator rend(); const_reverse_iterator rend() const;`	Возвращает реверсивный итератор для начала вектора
`void reserve(size_type num);`	Устанавливает емкость вектора равной значению не менее заданного `num`
`void resize(size_type num, T val = T());`	Устанавливает размер вектора равным значению, заданному параметром `num`. Если вектор для этого нужно удлинить, то в его конец добавляются элементы со значением, заданным параметром `val`
`size_type size() const;`	Возвращает текущее количество элементов в векторе
`void swap( deque<T, Allocator> &ob);`	Выполняет обмен элементами вызывающего вектора и вектора `ob`

Функция *begin()* возвращает итератор, который указывает на начало вектора. Функция *end()* возвращает итератор, который указывает на конец вектора. Как уже разъяснялось, итераторы подобны указателям, и с помощью функций *begin()* и *end()* можно получить итераторы для начала и конца вектора соответственно.

Функция *push_back()* помещает заданное значение в конец вектора. При необходимости длина вектора увеличивается так, чтобы он мог принять новый элемент. С помощью функции *insert()* можно добавлять элементы в середину вектора. Кроме того, вектор можно инициализировать. В любом случае, если вектор содержит элементы, то для доступа к ним и их модификации можно использовать средство индексации массивов. А с помощью функции *erase()* можно удалять элементы из вектора.

Рассмотрим короткий пример, который иллюстрирует базовое поведение вектора.

```
// Демонстрация базового поведения вектора.
#include <iostream>
#include <vector>
using namespace std;
```

```
using namespace std;

int main()
{
  vector<int> v; // создание вектора нулевой длины
  unsigned int i;

  // Отображаем исходный размер вектора v.
  cout << "Размер = " << v.size() << endl;

  /* Помещаем значения в конец вектора, и размер вектора будет по необходимости увеличиваться. */
  for(i=0; i<10; i++) v.push_back(i);

  // Отображаем текущий размер вектора v.
  cout << "Текущее содержимое:\n";
  cout << "Новый размер = " << v.size() << endl;

  // Отображаем содержимое вектора.
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  /* Помещаем в конец вектора новые значения, и размер вектора будет по необходимости увеличиваться. */
  for(i=0; i<10; i++ ) v.push_back(i+10);

  // Отображаем текущий размер вектора v.
  cout << "Новый размер = " << v.size() << endl;

  // Отображаем содержимое вектора.
  cout << "Текущее содержимое:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  // Изменяем содержимое вектора.
  for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

  // Отображаем содержимое вектора.
  cout << "Содержимое удвоено:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Размер = 0
Текущее содержимое:
Новый размер = 10
0 1 2 3 4 5 6 7 8 9
Новый размер = 20
```

```
Новый размер = 20
Текущее содержимое:
0 1 2 3 4  5  6   7  8   9  10 11 12 13 14 15 16 17 18 19
Содержимое удвоено:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Рассмотрим внимательно код этой программы. В функции main() создается вектор *v* для хранения *int*-элементов. Поскольку при его создании не было предусмотрено никакой инициализации, вектор *v* получился пустым, а его емкость равна нулю. Другими словами, мы создали вектор нулевой длины. Это подтверждается вызовом функции-члена *size()*. Затем, используя функцию-член *push_back()*, в конец этого вектора мы помещаем 10 элементов, что заставляет вектор увеличиться в размере, чтобы разместить новые элементы. Теперь размер вектора стал равным 10. Обратите внимание на то, что для отображения содержимого вектора *v* используется стандартная запись индексации массивов. После этого в вектор добавляются еще 10 элементов, и вектор *v* автоматически увеличивается в размере, чтобы и их принять на хранение. Наконец, используя опять-таки стандартную запись индексации массивов, мы изменяем значения элементов вектора *v*.

Обратите также внимание на то, что для управления циклами, используемыми для отображения содержимого вектора *v* и его модификации, в качестве признака их завершения применяется значение размера вектора, получаемое с помощью функции *v.size()*. Одно из преимуществ векторов перед массивами состоит в том, что у нас есть возможность узнать текущий размер вектора, что в определенных ситуациях является очень полезным средством.

Доступ к вектору с помощью итератора

Как вы знаете, массивы и указатели в C++ тесно связаны между собой. К элементам массива можно получить доступ как с помощью индекса, так и с помощью указателя. В библиотеке STL аналогичная связь существует между векторами и итераторами. Это значит, что к членам вектора можно обращаться как с помощью индекса, так и с помощью итератора. Эта возможность демонстрируется в следующей программе.

```
// Доступ к вектору с помощью итератора.
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
  vector<char> v; // создание массива нулевой длины
  int i;

  // Помещаем значения в вектор.
  for(i=0; i<10; i++) v.push_back('A' + i);

  /* Получаем доступ к содержимому вектора с помощью индекса. */
  for(i=0; i<10; i++) cout << v[i] << " ";
  cout << endl;

  /* Получаем доступ к содержимому вектора с помощью итератора. */
  vector<char>:: iterator p = v.begin( );
  while(p != v.end( )) {
    cout << *p << " ";
```

```
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Вот как выглядят результаты выполнения этой программы.

```
A B C D E F G H I J
A B C D E F G H I J
```

В этой программе сначала создается вектор нулевой длины. Затем с помощью функции *push_back()* в конец вектора помещаются символы, в результате чего размер вектора соответствующим образом увеличивается.

Обратите внимание на то, как объявляется итератор *p*. Тип этого итератора определяется контейнерными классами. Поэтому для получения итератора для конкретного контейнера используйте объявление, аналогичное показанному в этом примере: просто укажите для данного итератора имя контейнера. В нашей программе итератор *p* инициализируется таким образом, чтобы он указывал на начало вектора (с помощью функции-члена *begin()*). Итератор, который возвращает эта функция, можно затем использовать для поэлементного доступа к вектору, инкрементируя его соответствующим образом. Этот процесс аналогичен тому, как можно использовать указатель для доступа к элементам массива. Чтобы определить, когда будет достигнут конец вектора, используется функция-член *end()*. Эта функция возвращает итератор, установленный за последним элементом вектора. Поэтому, если значение *p* равно *v.end()*, значит, конец вектора достигнут.

Вставка и удаление элементов из вектора

Помимо занесения новых элементов в конец вектора, у нас есть возможность вставлять элементы в середину вектора, используя функцию *insert()*. Удалять элементы можно с помощью функции *erase()*. Использование функций *insert()* и *erase()* демонстрируется в следующей программе.

```
// Демонстрация вставки элементов в вектор и удаления их из него.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v;
  unsigned int i;

  for(i=0; i<10; i++) v.push_back('A' + i);

  // Отображаем исходное содержимое вектора.
  cout << "Размер = " << v.size() << endl;
  cout << "Исходное содержимое вектора:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  vector<char>:: iterator p = v.begin();
  p += 2; // указатель на 3-й элемент вектора
```

```
  // Вставляем 10 символов 'X' в вектор v.
  v.insert(p, 10, 'X');

  /* Отображаем содержимое вектора после вставки символов. */
  cout << "Размер вектора после вставки = " << v.size()<< endl;
  cout << "Содержимое вектора после вставки:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl << endl;

  // Удаление вставленных элементов.
  p = v.begin();
  p += 2; // указатель на 3-й элемент вектора
  v.erase(p, p+10); // Удаляем 10 элементов подряд.

  /* Отображаем содержимое вектора после удаления символов. */
  cout << "Размер вектора после удаления символов = "<< v.size() << endl;
  cout << "Содержимое вектора после удаления символов:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  return 0;
}
```

При выполнении эта программа генерирует следующие результаты.

```
Размер = 10
Исходное содержимое вектора:
A B C D E F G H I J

Размер вектора после вставки = 20
Содержимое вектора после вставки:
A B X X X X X X X X X C D E F G H I J

Размер вектора после удаления символов = 10
Содержимое вектора после удаления символов:
A B C D E F G H I J
```

Сохранение в векторе объектов класса

В предыдущих примерах векторы служили для хранения значений только встроенных типов, но этим их возможности не ограничиваются. В вектор можно помещать объекты любого типа, включая объекты классов, создаваемых программистом. Рассмотрим пример, в котором вектор используется для хранения объектов класса *three_d*. Обратите внимание на то, что в этом классе определяются конструктор по умолчанию и перегруженные версии операторов *"<"* и *"=="*. Имейте в виду, что, возможно, вам придется определить и другие операторы сравнения. Это зависит от того, как используемый вами компилятор реализует библиотеку STL.

```
// Хранение в векторе объектов класса.
#include <iostream>
#include <vector>
using namespace std;
```

```
class three_d {
   int x, y, z;
  public:
   three_d() { x = y = z = 0; }
   three_d(int a, int b, int c) { x = a; y = b; z = c; }
   three_d &operator+(int a) {
     x += a;
     y += a;
     z += a;
     return *this;
   }

   friend ostream &operator<<(ostream &stream, three_d obj);
   friend bool operator<(three_d a, three_d b);
   friend bool operator==(three_d a, three_d b);
};

/* Отображаем координаты X, Y, Z с помощью оператора вывода для класса
three_d. */
ostream &operator<<(ostream &stream, three_d obj)
{
  stream << obj.x << ", ";
  stream << obj.y << ", ";
  stream << obj.z << "\n";
  return stream;
}

bool operator<(three_d a, three_d b)
{
  return (a.x + a.y + a.z) < (b.x + b.y + b.z);
}

bool operator==(three_d a, three_d b)
{
  return (a.x + a.y + a.z) == (b.x + b.y + b.z);
}

int main()
{
  vector<three_d> v;
  unsigned int i;

  // Добавляем в вектор объекты.
  for(i=0; i<10; i++)
    v.push_back(three_d(i, i+2, i-3));
```

```
  // Отображаем содержимое вектора.
  for(i=0; i<v.size(); i++)
    cout << v[i];
  cout << endl;

  // Модифицируем объекты в векторе.
  for(i=0; i<v.size(); i++) v [i] = v[i] + 10;

  // Отображаем содержимое модифицированного вектора.
  for(i=0; i<v.size(); i++) cout << v[i];

  return 0;
}
```

Эта программа генерирует такие результаты.

```
0, 2, -3
1, 3, -2
2, 4, -1
3, 5, 0
5, 7, 2
6, 8, 3
7, 9, 4
8, 10, 5
9, 11, 6
10, 12, 7
11, 13, 8
12, 14, 9
13, 15, 10
14, 16, 11
15, 17, 12
16, 18, 13
17, 19, 14
18, 20, 15
19, 21, 16
```

Векторы обеспечивают безопасность хранения элементов, обнаруживая при этом чрезвычайную мощь и гибкость их обработки, но уступают массивам в эффективности использования. Поэтому для большинства задач программирования чаще отдается предпочтение обычным массивам. Однако возможны ситуации, когда уникальные особенности векторов оказываются важнее затрат системных ресурсов, связанных с их использованием.

О пользе итераторов

Частично сила библиотеки STL обусловлена тем, что многие ее функции используют итераторы. Этот факт позволяет выполнять операции с двумя контейнерами одновременно. Рассмотрим, например, такой формат векторной функции *insert()*.

```
template <class InIter>
```

```
template <class InIter>
void insert(iterator i, InIter start, InIter end);
```

Эта функция вставляет исходную последовательность, определенную параметрами *start* и *end*, в приемную последовательность, начиная с позиции *i*. При этом нет никаких требований, чтобы итератор *i* относился к тому же вектору, с которым связаны итераторы *start* и *end*. Таким образом, используя эту версию функции *insert()*, можно один вектор вставить в другой. Рассмотрим пример.

```
// Вставляем один вектор в другой.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v, v2;
  unsigned int i;

  for(i=0; i<10; i++) v.push_back('A' + i);

  // Отображаем исходное содержимое вектора.
  cout << "Исходное содержимое вектора:\n";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl << endl;

  // Инициализируем второй вектор.
  char str[] = "-STL – это сила!-";
  for(i=0; str[i]; i++) v2 .push_back (str [i]);

  /* Получаем итераторы для середины вектора v, а также начала и конца
вектора v2. */
  vector<char>::iterator p = v.begin()+5;
  <char>::iterator p2start = v2.begin();
  vector<char>::iterator p2end = v2.end();

  // Вставляем вектор v2 в вектор v.
  v.insert(p, p2start, p2end);

  // Отображаем результат вставки.
  cout << "Содержимое вектора v после вставки:\n";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

При выполнении эта программа генерирует следующие результаты.

```
Исходное содержимое вектора:
A B C D E F G H I J
Содержимое вектора v после вставки:
```

```
A B C D E - S T L -- это сила! - F G H I J
```

Как видите, содержимое вектора *v2* вставлено в середину вектора *v*.

По мере дальнейшего изучения возможностей, предоставляемых STL, вы узнаете, что итераторы являются связующими средствами, которые делают библиотеку единым целым. Они позволяют работать с двумя (и больше) объектами STL одновременно, но особенно полезны при использовании алгоритмов, описанных ниже в этой главе.

Списки

Список — *это контейнер с двунаправленным последовательным доступом к его элементам*.

Класс *list* поддерживает функционирование двунаправленного линейного списка. В отличие от вектора, в котором реализована поддержка произвольного доступа, список позволяет получать к своим элементам только последовательный доступ. Двунаправленность списка означает, что доступ к его элементам возможен в двух направлениях: от начала к концу и от конца к началу.

Шаблонная спецификация класса *list* выглядит следующим образом.

```
template <class T, class Allocator = allocator<T>> class list
```

Здесь *T* — тип данных, сохраняемых в списке, а элемент *Allocator* означает распределитель памяти, который по умолчанию использует стандартный распределитель. В классе *list* определены следующие конструкторы.

```
explicit list(const Allocator &a = Allocator( ) );

explicit list(size_type num, const T &val = T( ), const Allocator &a =
Allocator( ));

list(const list<T, Allocator> &ob);

template <class InIter>list(InIter start, InIter end, const Allocator &a
= Allocator( ));
```

Конструктор, представленный в первой форме, создает пустой список. Вторая форма предназначена для создания списка, который содержит *num* элементов со значением *val*. Третья создает список, который содержит те же элементы, что и объект *ob*. Четвертая создает список, который содержит элементы в диапазоне, заданном параметрами *start* и *end*.

Для класса *list* определены следующие операторы сравнения:

```
==, <, <=, !=, > и >=
```

Функции-члены, определенные в классе *list*, перечислены в табл. 21.3. В конец списка, как и в конец вектора, элементы можно помещать с помощью функции *push_back()*, но с помощью функции *push_front()* можно помещать элементы в начало списка. Элемент можно также вставить и в середину списка, для этого используется функция *insert()*. Один список можно поместить в другой, используя функцию *splice()*. А с помощью функции *merge()* два списка можно объединить и упорядочить результат.

Таблица 21.3. Функции-члены класса `list`

Функция-член	Описание
`template <class InIter> void assign(InIter start, InIter end);`	Помещает в список последовательность, определяемую параметрами `start` и `end`
`void assign(size_type num, const T &val);`	Помещает в список `num` элементов со значением `val`
`reference back();`	Возвращает ссылку на последний элемент в списке

`const_reference back() const;`	
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор для первого элемента в списке
`void clear();`	Удаляет все элементы из списка
`bool empty() const;`	Возвращает истинное значение, если используемый список пуст, и ложное в противном случае
`iterator end();` `const_iterator end() const;`	Возвращает итератор, соответствующий концу списка
`iterator erase(iterator i);`	Удаляет элемент, адресуемый итератором `i`; возвращает итератор, указывающий на элемент, расположенный после удаленного

Продолжение табл. 21.3

Функция-член	Описание

`iterator erase(`
`  iterator start,`
`  iterator end);`
	Удаляет элементы в диапазоне, задаваемом параметрами `start` и `end`; возвращает итератор для элемента, расположенного за последним удаленным элементом

`reference front();` `const_reference front() const;`	Возвращает ссылку на первый элемент в списке
`allocator_type` `  get_allocator() const;`	Возвращает распределитель памяти списка

`iterator insert(`
`  iterator i,`
`  const T &val = T());`
	Вставляет значение `val` непосредственно перед элементом, заданным параметром `i`; возвращает итератор для этого элемента

`void insert(iterator i,`
`  size_type num,`
`  const T &val);`
	Вставляет `num` копий значения `val` непосредственно перед элементом, заданным параметром `i`

`template <class InIter>`
`  void insert(`
`  iterator i,`
`  InIter start,`
`  InIter end);`
	Вставляет в список последовательность, определяемую параметрами `start` и `end`, непосредственно перед элементом, заданным параметром `i`

`size_type max_size() const;`	Возвращает максимальное число элементов, которое может содержать список

`void merge(`
`  list<T, Allocator> &ob);`
`template <class Comp>`
`  void merge(`
`  list<T, Allocator> &ob,`
`  Comp cmpfn);`
	Объединяет упорядоченный список, содержащийся в объекте `ob`, с вызывающим упорядоченным списком. Результат также упорядочивается. После объединения список, содержащийся в `ob`, остается пустым. Во второй форме может быть задана функция сравнения, которая определяет, при каких условиях один элемент меньше другого

`void pop_back();`	Удаляет последний элемент в списке
`void pop_front();`	Удаляет первый элемент в списке
`void push_back(` `  const T &val);`	Добавляет в конец списка элемент со значением, заданным параметром `val`
`void push_front(` `  const T &val);`	Добавляет в начало списка элемент со значением, заданным параметром `val`

`reverse_iterator rbegin();`
`const_reverse_iterator`
`  rbegin() const;`
	Возвращает реверсивный итератор для конца списка

`reverse_iterator rend();` `const_reverse_iterator rend() const;`	Возвращает реверсивный итератор для начала списка
`void remove(const T &val);`	Удаляет из списка элементы со значением *val*
`template <class UnPred> void remove_if(UnPred pr);`	Удаляет элементы, для которых унарный предикат *pr* равен значению `true`

Окончание табл. 21.3

Функция-член	Описание
`void resize(size_type num, T val = T());`	Устанавливает размер списка равным значению, заданному параметром *num*. Если список для этого нужно удлинить, то в конец списка добавляются элементы со значением, заданным параметром *val*
`void reverse();`	Реверсирует список
`size_type size() const;`	Возвращает текущее количество элементов в списке
`void sort();` `template <class Comp> void sort(Comp cmpfn);`	Сортирует список. Вторая форма сортирует список с помощью функции сравнения *cmpfn*, которая позволяет определять, при каких условиях один элемент меньше другого
`void splice(iterator i, list<T, Allocator> &ob);`	Вставляет содержимое списка *ob* в вызывающий список в позиции, указанной итератором *i*. После выполнения этой операции список *ob* остается пустым
`void splice(iterator i, list<T, Allocator> &ob, iterator el);`	Удаляет из списка *ob* элемент, адресуемый итератором *el*, и сохраняет его в вызывающем списке в позиции, адресуемой итератором *i*
`void splice(iterator i, list<T, Allocator> &ob, iterator start, iterator end);`	Удаляет из списка *ob* диапазон, определяемый параметрами *start* и *end*, и сохраняет его в вызывающем списке, начиная с позиции, адресуемой итератором *i*
`void swap(list<T, Allocator> &ob);`	Выполняет обмен элементами вызывающего списка и списка *ob*
`void unique();` `template <class BinPred> void unique(BinPred pr);`	Удаляет из списка элементы-дубликаты. Вторая форма для определения уникальности использует предикат *pr*

Чтобы достичь максимальной гибкости и переносимости для любого объекта, который подлежит хранению в списке, следует определить конструктор по умолчанию и оператор "<" (и желательно другие операторы сравнения). Более точные требования к объекту (как к потенциальному элементу списка) необходимо согласовывать в соответствии с документацией на используемый вами компилятор.

Рассмотрим простой пример списка.

```
// Базовые операции, определенные для списка.
#include <iostream>
#include <list>
using namespace std;

int main()
{
```

```
  list<char> lst; // создание пустого списка
  int i;

  for(i=0; i<10; i++) lst.push_back('A'+i);

  cout << "Размер = " << lst.size() << endl;

  cout << "Содержимое : ";
  list<char>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

Результаты выполнения этой программы таковы:

```
Размер = 10
Содержимое : ABCDEFGHIJ
```

При выполнении эта программа создает список символов. Сначала создается пустой объект списка. Затем в него помещается десять букв (от *A* до *J*). Заполнение списка реализуется путем использования функции *push_back()*, которая помещает каждое новое значение в конец существующего списка. После этого отображается размер списка и его содержимое. Содержимое списка выводится на экран в результате выполнения следующего кода.

```
list<char>::iterator p = lst.begin();
while(p != lst.end()) {
  cout << *p;
  p++;
}
```

Здесь итератор *p* инициализируется таким образом, чтобы он указывал на начало списка. При выполнении очередного прохода цикла итератор *p* инкрементируется, чтобы указывать на следующий элемент списка. Этот цикл завершается, когда итератор *p* указывает на конец списка. Применение подобных циклов — обычная практика при использовании библиотеки STL. Например, аналогичный цикл мы применили для отображения содержимого вектора в предыдущем разделе.

Поскольку списки являются двунаправленными, заполнение их элементами можно производить с обоих концов. Например, при выполнении следующей программы создается два списка, причем элементы одного из них расположены в порядке, обратном по отношению к другому.

```
/* Элементы можно помещать в список как с начала, так и с конца.
*/
#include <iostream>
#include <list>
using namespace std;

int main()
{
```

```
  list<char> lst;
  list<char> revlst;
  int i;

  for(i=0; i<10; i++ ) lst.push_back('A'+i);

  cout << "Размер списка lst = " << lst.size() << endl;
  cout << "Исходное содержимое списка: ";

  list<char>::iterator p;

  /* Удаляем элементы из списка lst и помещаем их в список revlst в
обратном порядке. */

  while(!lst.empty()) {
    p = lst.begin();
    cout << *p;
    revlst.push_front(*p);
    lst.pop_front();
  }
  cout << endl << endl;

  cout << "Размер списка revlst = ";
  cout << revlst.size() << endl;

  cout << "Реверсированное содержимое списка: ";
  p = revlst.begin();
  while(p != revlst.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

Эта программа генерирует такие результаты.

```
Размер списка lst = 10
Исходное содержимое списка: ABCDEFGHIJ

Размер списка revlst = 10
Реверсированное содержимое списка: JIHGFEDCBA
```

В этой программе список реверсируется путем удаления элементов с начала списка *lst* и занесения их в начало списка *revlst*.

Сортировка списка

Список можно отсортировать с помощью функции-члена *sort()*. При выполнении следующей программы создается список случайно выбранных целых чисел, который затем упорядочивается по возрастанию.

```
// Сортировка списка.
#include <iostream>
```

```
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<int> lst;
  int i;

  // Создание списка случайно выбранных целых чисел.
  for(i=0; i<10; i++ )lst.push_back(rand() );

  cout << "Исходное содержимое списка:\n";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // Сортировка списка.
  lst.sort();

  cout << "Отсортированное содержимое списка:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

Вот как может выглядеть один из возможных вариантов выполнения этой программы.

```
Исходное содержимое списка:
41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Отсортированное содержимое списка:
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
```

Объединение одного списка с другим

Один упорядоченный список можно объединить с другим. В результате мы получим упорядоченный список, который включает содержимое двух исходных списков. Новый список остается в вызывающем списке, а второй список становится пустым. В следующем примере выполняется слияние двух списков. Первый список содержит буквы *ACEGI*, а второй— буквы *BDFHJ*. Эти списки затем объединяются, в результате чего образуется упорядоченная последовательность букв *ABCDEFGHIJ*.

```
// Слияние двух списков.
```

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

int main()
{
  list<char> lst1, lst2;
  int i;

  for(i=0; i<10; i+=2) lst1.push_back('A'+i);
  for(i=1; i<11; i+=2) lst2.push_back('A'+i);

  cout << "Содержимое списка lst1: ";
  list<char>::iterator p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  cout << "Содержимое списка lst2: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p;
    p++;
  }
  cout << endl << endl;

  // Теперь сливаем эти два списка.
  lst1.merge(lst2);
  if(lst2.empty())
    cout << "Список lst2 теперь пуст.\n";

  cout << "Содержимое списка lst1 после объединения:\n";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p;
    p++;
  }

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Содержимое списка lst1: ACEGI

Содержимое списка lst2: BDFHJ
```

```
Список lst2 теперь пуст.
Содержимое списка lst1 после объединения:
ABCDEFGHIJ
```

Хранение в списке объектов класса

Рассмотрим пример, в котором список используется для хранения объектов типа *myclass*. Обратите внимание на то, что для объектов типа *myclass* перегружены операторы *"<"*, *">"*, *"!="* и *"=="*. (Для некоторых компиляторов может оказаться излишним определение всех этих операторов или же придется добавить некоторые другие.) В библиотеке STL эти функции используются для определения упорядочения и равенства объектов в контейнере. Несмотря на то что список не является упорядоченным контейнером, необходимо иметь средство сравнения элементов, которое применяется при их поиске, сортировке или объединении.

```
// Хранение в списке объектов класса.
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass {
   int a, b;
   int sum;
  public:
   myclass() { a = b = 0; }
   myclass(int i, int j) {
     a = i;
     b = j;
     sum = a + b;
   }
   int getsum() { return sum; }

   friend bool operator<(const myclass &o1, const myclass &o2);
   friend bool operator>(const myclass &o1, const myclass &o2);
   friend bool operator==(const myclass &o1, const myclass &o2);
   friend bool operator!=(const myclass &o1, const myclass &o2);
};

bool operator<(const myclass &o1, const myclass &o2)
{
  return o1.sum < o2.sum;
}

bool operator>(const myclass &o1, const myclass &o2)
{
  return o1.sum > o2.sum;
}

bool operator==(const myclass &o1, const myclass &o2)
```

```
bool operator==(const myclass &o1, const myclass &o2)
{
  return o1.sum == o2.sum;
}

bool operator!=(const myclass &o1, const myclass &o2)
{
  return o1.sum != o2.sum;
}

int main()
{
  int i;

  // Создание первого списка.
  list<myclass> lst1;
  for(i=0; i <10; i++) lst1.push_back(myclass(i, i));

  cout << "Первый список: ";
  list<myclass>::iterator p = lst1.begin();
    while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // Создание второго списка.
  list<myclass> lst2;
  for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));

  cout << "Второй список: ";
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << p->getsum() << " ";
    p++;
  }
  cout << endl;

  // Теперь объединяем списки lst1 и lst2.
    lst1.merge(lst2);

  // Отображаем объединенный список.
  cout << "Объединенный список: ";
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << p->getsum() << " ";
    p++;
```

```
  }

  return 0;
}
```

Эта программа создает два списка объектов типа *myclass* и отображает их содержимое. Затем выполняется объединение этих двух списков с последующим отображением нового содержимого результирующего списка. Итак, программа генерирует такие результаты.

```
Первый список: 0 2 4 6 8 10 12 14 16 18
Второй список: 0 5 10 15 20 25 30 35 40 45
Объединенный список: 0 0 2 4 5 6 8 10 10 12 14 15 16 18 20 25 30 35 40 45
```

Отображения

Отображение — *это ассоциативный контейнер*.

Класс *map* поддерживает ассоциативный контейнер, в котором уникальным ключам соответствуют определенные значения. По сути, ключ — это просто имя, которое присвоено некоторому значению. После того как значение сохранено в контейнере, к нему можно получить доступ, используя его ключ. Таким образом, в самом широком смысле отображение — это список пар "ключ-значение". Если нам известен ключ, мы можем легко найти значение. Например, мы могли бы определить отображение, в котором в качестве ключа используется имя человека, а в качестве значения — его телефонный номер. Ассоциативные контейнеры становятся все более популярными в программировании.

Как упоминалось выше, отображение может хранить только уникальные ключи. Ключи-дубликаты не разрешены. Чтобы создать отображение, которое бы позволяло хранить неуникапьные ключи, используйте класс *multimap*.

Контейнер *map* имеет следующую шаблонную спецификацию.

```
template <class Key, class T, class Comp = less<Key>, class Allocator =allocator<pair<const Key, T> > >class map
```

Здесь *Key*— тип данных ключей, *T*— тип сохраняемых (отображаемых) значений, а *Comp* — функция, которая сравнивает два ключа. По умолчанию в качестве функции сравнения используется стандартная функция-объект *less*. Элемент *Allocator* означает распределитель памяти, который по умолчанию использует стандартный распределитель *allocator*.

Класс *map* имеет следующие конструкторы.

```
explicit map(const Comp &cmpfn = Comp(), const Allocator &a = Allocator());

map(const map<Key, T, Comp, Allocator> &ob);

template <class InIter> map(InIter start, InIter end, const Comp &cmpfn = Comp(), const Allocator &a = Allocator());
```

Первая форма конструктора создает пустое отображение. Вторая предназначена для создания отображения, которое содержит те же элементы, что и отображение *ob*. Третья создает отображение, которое содержит элементы в диапазоне, заданном итераторами *start* и *end*. Функция, заданная параметром *cmpfn* (если она задана), определяет характер упорядочения отображения.

В общем случае любой объект, используемый в качестве ключа, должен определять конструктор по умолчанию и перегружать оператор *"<"* (а также другие необходимые операторы сравнения). Эти требования для разных компиляторов различны.

Для класса *map* определены следующие операторы сравнения:

```
==  <  <=  !=  >  и  >=
```

--, <, <=, !=, > и >=

Функции-члены, определенные для класса *map*, представлены в табл. 21.4. В их описании под элементом *key_type* понимается тип ключа, а под элементом *value_type* — значение выражения *pair<Key, T>*.

Таблица 21.4. Функции-члены, определенные в классе map

Функция-член	Назначение
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор для первого элемента в отображении
`void clear();`	Удаляет все элементы из отображения
`size_type count(` `  const key_type &k) const;`	Возвращает число вхождений ключа *k* в отображении (1 или 0)
`bool empty() const;`	Возвращает значение `true`, если данное отображение пустое, и значение `false` в противном случае
`iterator end();` `const_iterator end() const;`	Возвращает итератор, указывающий на конец отображения

`pair<iterator, iterator>`
`  equal_range(`
`  const key_type &k);`
`pair<const_iterator,`
`  const_iterator>`
`  equal_range(`
`  const key_type &k) const;`
	Возвращает пару итераторов, которые указывают на первый и последний элементы в отображении, содержащие заданный ключ

`void erase(iterator i);`	Удаляет элемент, на который указывает итератор *i*
`void erase(iterator start,` `  iterator end);`	Удаляет элементы в диапазоне, заданном параметрами *start* и *end*
`size_type erase(` `  const key_type &k);`	Удаляет из отображения элементы, ключи которых имеют значение *k*

`iterator find(`
`  const key_type &k);`
`const_iterator find(`
`  const key_type &k) const;`
	Возвращает итератор, соответствующий заданному ключу. Если такой ключ не обнаружен, возвращает итератор, соответствующий концу отображения

`allocator_type` `  get_allocator() const;`	Возвращает распределитель памяти отображения

`iterator insert(`
`  iterator i,`
`  const value_type &val);`
	Вставляет значение *val* в позицию элемента (или после него), заданного итератором *i*. Возвращает итератор, указывающий на этот элемент

`template <class InIter>`
`  void insert(InIter start,`
`  InIter end);`
	Вставляет элементы заданного диапазона

`pair<iterator, bool>`
`  insert(`
`  const value_type &val);`
	Вставляет значение *val* в вызывающее отображение. Возвращает итератор, указывающий на этот элемент. Элемент вставляется только в том случае, если его еще нет в отображении. Если элемент был вставлен, возвращает значение `pair<iterator, true>`, в противном случае — значение `pair<iterator, false>`

`key_compare key_comp() const;`	Возвращает объект-функцию, которая сравнивает ключи

`iterator lower_bound(`
`  const key_type &k);`
`const_iterator lower_bound(`
	Возвращает итератор для первого элемента в отображении, ключ которого равен значению *k* или больше этого значения

`const key_type &k) const;`	

Окончание табл. 21.4

Функция-член	Назначение
`size_type max_size() const;`	Возвращает максимальное число элементов, которое может содержать данное отображение
`reference operator[](` `const key_type &i);`	Возвращает ссылку на элемент, заданный параметром `i`. Если этого элемента не существует, вставляет его в отображение

`reverse_iterator rbegin();`
`const_reverse_iterator`
`rbegin() const;`
	Возвращает реверсивный итератор, соответствующий концу отображения

`reverse_iterator rend();`
`const_reverse_iterator`
`rend() const;`
	Возвращает реверсивный итератор, соответствующий началу отображения

`size_type size() const;`	Возвращает текущее число элементов в отображении
`void swap(map<Key, T, Comp,` `Allocator> &ob);`	Выполняет обмен элементами вызывающего отображения и отображения `ob`

`iterator upper_bound(`
`const key_type &k);`
`const_iterator upper_bound(`
`const key_type &k) const;`
	Возвращает итератор для первого элемента в отображении, ключ которого больше заданного значения `k`

`value_compare value_comp()` `const;`	Возвращает объект-функцию, которая сравнивает значения

Пары "ключ-значение" хранятся в отображении как объекты класса *pair*, который имеет следующую шаблонную спецификацию.

```
template <class Ktype, class Vtype>
struct pair {
  typedef Ktype first_type; // тип ключа
  typedef Vtype second_type; // тип значения
  Ktype first; // содержит ключ
  Vtype second; // содержит значение

  // Конструкторы
  pair();
  pair (const Ktype &k, const Vtype &v);
  template<class A, class B> pair(const<A, B> &ob);
}
```

Как отмечено в комментариях, член *first* содержит ключ, а член *second* — значение, соответствующее этому ключу.

Создать пару "ключ-значение" можно либо с помощью конструкторов класса *pair*, либо путем вызова функции *make_pair()*, которая создает парный объект на основе типов данных, используемых в качестве параметров. Функция *make_pair()* — это обобщенная функция, прототип которой имеет следующий вид.

```
template <class Ktype, class Vtype>
pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

```
pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

Как видите, функция *make_pair()* возвращает парный объект, состоящий из значений, типы которых заданы параметрами *Ktype* и *Vtype*. Преимущество использования функции *make_pair()* состоит в том, что типы объектов, объединяемых в пару, определяются автоматически компилятором, а не явно задаются программистом.

Возможности использования отображения демонстрируется в следующей программе. В данном случае в отображении сохраняется *10* пар "ключ-значение". Ключом служит символ, а значением — целое число. Пары "ключ-значение" хранятся следующим образом:

```
A 0
B 1
C 2
```

и т.д. После сохранения пар в отображении пользователю предлагается ввести ключ (т.е. букву из диапазона *A-J*), после чего выводится значение, связанное с этим ключом.

```
// Демонстрация использования простого отображения.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // Помещаем пары в отображение.
  for(i = 0; i <10; i++) {
    m.insert(pair<char, int>('A'+i, i));
  }

  char ch;
  cout << "Введите ключ: ";
  cin >> ch;

  map<char, int>::iterator p;

  // Находим значение по заданному ключу.
  p = m.find(ch);
  if(p != m.end())
    cout << p->second;
  else cout << "Такого ключа в отображении нет.\n";

  return 0;
}
```

Обратите внимание на использование шаблонного класса *pair* для построения пар "ключ-значение". Типы данных, задаваемые *pair*-выражением, должны соответствовать типам отображения, в которое вставляются эти пары.

После инициализации отображения ключами и значениями можно выполнять поиск значения по заданному ключу, используя функцию *find()*. Эта функция возвращает итератор, который указывает на нужный элемент или на конец отображения, если заданный ключ не был найден. При обнаружении совпадения значение, связанное с ключом, можно найти в

члене *second* парного объекта типа *pair*.

В предыдущем примере пары "ключ-значение" создавались явным образом с помощью шаблона *pair<char, int>*. И хотя в этом нет ничего неправильного, зачастую проще использовать с этой целью функцию *make_pair()*, которая создает *pair*-объект на основе типов данных, используемых в качестве параметров. Например, эта строка кода также позволит вставить в отображение *m* пары "ключ-значение" (при использовании предыдущей программы):

```
m.insert(make_pair((char) ('A'+i), i));
```

Здесь, как видите, выполняется операция приведения к типу *char*, которая необходима для переопределения автоматического преобразования в тип *int* результата сложения значения *i* с символом *'A'*.

Хранение в отображении объектов класса

Подобно всем другим контейнерам, отображение можно использовать для хранения объектов создаваемых вами типов. Например, следующая программа создает простой словарь на основе отображения слов с их значениями. Но сначала она создает два класса *word* и *meaning*. Поскольку отображение поддерживает отсортированный список ключей, программа также определяет для объектов типа *word* оператор *"<"*. В общем случае оператор *"<"* следует определять для любых классов, которые вы собираетесь использовать в качестве ключей. (Некоторые компиляторы могут потребовать определения и других операторов сравнения.)

```
// Использование отображения для создания словаря.
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class word {
   char str[20];
  public:
   word() { strcpy(str, ""); }
   word(char *s) { strcpy(str, s); }

   char *get() { return str; }
};

bool operator<(word a, word b)
{
  return strcmp(a.get(), b.get()) < 0;
}

class meaning {
   char str[80];
  public:
   meaning() { strcmp(str, " ");}
   meaning(char *s) { strcpy(str, s); }

   char *get() { return str; }
}
```

```
};

int main()
{
  map<word, meaning> dictionary;

  /* Помещаем в отображение объекты классов word и meaning. */
   dictionary.insert( pair<word, meaning> (word("дом"), meaning("Место проживания.")));

    dictionary.insert( pair<word, meaning> (word("клавиатура"), meaning("Устройство ввода данных.")));

   dictionary.insert( pair<word, meaning> (word("программирование"), meaning("Процесс создания программы.")));

  dictionary.insert( pair<word, meaning> (word("STL"), meaning("Standard Template Library")));

  // По заданному слову находим его значение.
  char str[80];
  cout << "Введите слово: ";
    cin >> str;

  map<word, meaning>::iterator p;

  p = dictionary.find(word(str));
  if(p != dictionary.end())
    cout << "Определение: " << p->second.get();
  else cout << "Такого слова в словаре нет.\n";

  return 0;
}
```

Вот один из возможных вариантов выполнения этой программы.

```
Введите слово: дом
Определение: Место проживания.
```

В этой программе каждый элемент отображения представляет собой символьный массив, который содержит строку с завершающим нулем. Ниже в этой главе мы рассмотрим более простой вариант построения этой программы, в которой использован стандартный тип *string*.

Алгоритмы

Алгоритмы обрабатывают данные, содержащиеся в контейнерах. Несмотря на то что каждый контейнер обеспечивает поддержку собственных базовых операций, стандартные алгоритмы позволяют выполнять более расширенные или более сложные действия. Они также позволяют работать с двумя различными типами контейнеров одновременно. Для получения доступа к алгоритмам библиотеки STL необходимо включить в программу заголовок *<algorithm>*.

В библиотеке STL определено множество алгоритмов, которые описаны в табл. 21.5. Все эти алгоритмы представляют собой шаблонные функции. Это означает, что их можно применять к контейнеру любого типа.

Таблица 21.5. Алгоритмы STL

Алгоритм	Назначение
adjacent_find	Выполняет поиск совпадающих смежных элементов внутри последовательности и возвращает итератор для первого найденного совпадения
binary_search	Выполняет двоичный поиск заданного значения внутри упорядоченной последовательности
copy	Копирует последовательность
copy_backward	Аналогичен алгоритму copy, за исключением того, что копирование происходит в обратном порядке, т.е. сначала перемещаются элементы, находящиеся в конце последовательности
count	Возвращает количество элементов с заданным значением в последовательности
count_if	Возвращает количество элементов, которые удовлетворяют заданному предикату
equal	Определяет, одинаковы ли два диапазона
equal_range	Возвращает диапазон, в который можно вставить элемент, не нарушая порядок некоторой последовательности
fill и fill_n	Заполняют диапазон заданным значением
find	В заданном диапазоне выполняет поиск заданного значения и возвращает итератор для первого вхождения найденного элемента
find_end	В заданном диапазоне выполняет поиск заданной последовательности. Возвращает итератор, соответствующий концу искомой последовательности

Продолжение табл. 21.5

Алгоритм	Назначение
find_first_of	Выполняет поиск первого элемента внутри заданной последовательности, который совпадает с любым элементом из заданного диапазона
find_if	В заданном диапазоне выполняет поиск элемента, для которого определенный пользователем унарный предикат возвращает значение true
for_each	Применяет заданную функцию к заданному диапазону элементов
generate и generate_n	Присваивают значения, возвращаемые некоторой функцией-генератором, элементам из заданного диапазона
includes	Устанавливает факт включения всех элементов одной заданной последовательности в другую заданную последовательность
inplace_merge	Объединяет один заданный диапазон с другим. Оба диапазона должны быть отсортированы в порядке возрастания После выполнения алгоритма полученная последовательность сортируется в порядке возрастания
iter_swap	Меняет местами значения, адресуемые итераторами, которые передаются в качестве параметров
lexicographical_compare	Сравнивает одну заданную последовательность с другой в лексикографическом порядке
lower_bound	Выполняет поиск первого элемента в заданной последовательности, значение которого не меньше заданного значения
make_heap	Создает кучу из заданной последовательности

max	Возвращает максимальное из двух значений
max_element	Возвращает итератор для максимального элемента внутри заданного диапазона
merge	Объединяет две упорядоченные последовательности, помещая результат в третью последовательность
min	Возвращает минимальное из двух значений
min_element	Возвращает итератор для минимального элемента внутри заданного диапазона
mismatch	Выполняет поиск первого несовпадения элементов в двух последовательностях и возвращает итераторы для этих двух элементов
next_permutation	Создает следующую перестановку заданной последовательности
nth_element	Упорядочивает заданную последовательность таким образом, чтобы все элементы, значения которых меньше значения E, размещались перед этим элементом, а все элементы, значения которых больше значения E, размещались после него
partial_sort	Сортирует заданный диапазон
partial_sort_copy	Сортирует заданный диапазон, а затем копирует столько элементов, сколько может поместиться в результирующую последовательность
partition	Сортирует заданную последовательность таким образом, чтобы все элементы, для которых заданный предикат возвращает значение `true`, размещались перед элементами, для которых этот предикат возвращает значение `false`

Окончание табл. 21.5

Алгоритм	Назначение
pop_heap	Меняет местами первый и предпоследний элементы заданного диапазона, а затем перестраивает кучу
prev_permutation	Создает предыдущую перестановку последовательности
push_heap	Помещает элемент в конец кучи
random_shuffle	Придает случайный характер заданной последовательности
remove, remove_if, remove_copy и remove_copy_if	Удаляют элементы из заданного диапазона
replace, replace_copy, replace_if и replace_copy_if	Заменяют заданные элементы из диапазона другими элементами
reverse и reverse_copy	Меняет порядок следования элементов в заданном диапазоне на противоположный
rotate и rotate_copy	Выполняет циклический сдвиг влево элементов в заданном диапазоне
search	Выполняет поиск одной последовательности внутри другой
search_n	Внутри некоторой последовательности выполняет поиск заданного числа подобных элементов
set_difference	Создает последовательность, которая содержит разность двух упорядоченных множеств
set_intersection	Создает последовательность, которая содержит пересечение двух упорядоченных множеств
set_union	Создает последовательность, которая содержит объединение двух упорядоченных множеств

sort	Сортирует заданный диапазон
sort_heap	Сортирует кучу в заданном диапазоне
stable_partition	Упорядочивает заданную последовательность таким образом, чтобы все элементы, для которых заданный предикат возвращает значение `true`, размещались перед элементами, для которых этот предикат возвращает значение `false`. Такое разбиение является стабильным, что означает сохранение относительного порядка последовательности
stable_sort	Выполняет устойчивую (стабильную) сортировку заданного диапазона. Это значит, что равные элементы не переставляются
swap	Меняет местами заданные два значения
swap_ranges	Выполняет обмен элементов в заданном диапазоне
transform	Применяет функцию к заданному диапазону элементов и сохраняет результат в новой последовательности
unique и unique_copy	Удаляет повторяющиеся элементы из заданного диапазона
upper_bound	Находит последний элемент в заданной последовательности, который не больше заданного значения

Подсчет элементов

Одна из самых популярных операций, которую можно выполнить для любой последовательности элементов, — подсчитать их количество. Для этого можно использовать один из алгоритмов: *count()* или *count_if()*. Общий формат этих алгоритмов имеет следующий вид.

```
template <class InIter, class T>
ptrdiff_t count(InIter start, InIter end, const T &val);

template <class InIter, class UnPred>
ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

Алгоритм *count()* возвращает количество элементов, равных значению *val*, в последовательности, границы которой заданы параметрами *start* и *end*. Алгоритм *count_if()*, действуя в последовательности, границы которой заданы параметрами *start* и *end*, возвращает количество элементов, для которых унарный предикат *pfn* возвращает значение *true*. Тип *ptrdiff_t* определяется как некоторая разновидность целочисленного типа.

Использование алгоритмов *count()* и *count_if()* демонстрируется в следующей программе.

```
/* Демонстрация использования алгоритмов count и count_if.
*/
#include <iostream>
#include <vector>
#include <algorithm>
#include <cctype>
using namespace std;

/* Это унарный предикат, который определяет, представляет ли данный символ гласный звук.
*/
bool isvowel(char ch)
{
  ch = tolower(ch);
  if(ch=='a' || ch=='e' || ch=='и' || ch=='o' || ch=='y' || ch=='ы' ||
```

```
ch=='я' || ch=='ё' || ch=='ю' || ch=='э')
   return true;

  return false;
}

int main()
{
  char str[] = "STL-программирование — это сила!";
  vector<char> v;
  unsigned int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  cout << "Последовательность: ";
  for(i=0; i<v.size(); i++) cout << v[i];
  cout << endl;

  int n;

  n = count (v.begin(), v.end(), 'м');
  cout << n << " символа м\n";

  n = count_if(v.begin(), v.end(), isvowel);
  cout << n << " символов представляют гласные звуки.\n";

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Последовательность: STL-программирование -- это сила!
2 символа м
11 символов представляют гласные звуки.
```

Программа начинается с создания вектора, который содержит строку *"STL-программирование - это сила!"*. Затем используется алгоритм *count()* для подсчета количества букв *'м'* в этом векторе. После этого вызывается алгоритм *count_if()*, который подсчитывает количество символов, представляющих гласные звуки с использованием в качестве предиката функции *isvowel()*. Обратите внимание на то, как закодирован этот предикат. Все унарные предикаты получают в качестве параметра объект, тип которого совпадает с типом элементов, хранимых в контейнере, для которого и создается этот предикат. Предикат должен возвращать значение *ИСТИНА* или *ЛОЖЬ*.

Удаление и замена элементов

Иногда полезно сгенерировать новую последовательность, которая будет состоять только из определенных элементов исходной последовательности. Одним из алгоритмов, который может справиться с этой задачей, является *remove_copy()*. Его общий формат выглядит так.

```
template <class ForIter, class OutIter, class T>
OutIter remove_copy(InIter start, InIter end, OutIter result, const T &val);
```

Алгоритм *remove_copy()* копирует с извлечением из заданного диапазона элементы, которые равны значению *val*, и помещает результат в последовательность, адресуемую параметром

result. Алгоритм возвращает итератор, указывающий на конец результата. Контейнер-приемник должен быть достаточно большим, чтобы принять результат.

Чтобы в процессе копирования один элемент в последовательности заменить другим, используйте алгоритм *replace_copy()*. Его общий формат выглядит так.

```
template <class ForIter, class OutIter, class T>
OutIter replace_copy(InIter start, InIter end, OutIter result, const T &old, Const T &new);
```

Алгоритм *replace_copy()* копирует элементы из заданного диапазона в последовательность, адресуемую параметром *result*. В процессе копирования происходит замена элементов, которые имеют значение *old*, элементами, имеющими значение *new*. Алгоритм помещает результат в последовательность, адресуемую параметром *result*, и возвращает итератор, указывающий на конец этой последовательности. Контейнер-приемник должен быть достаточно большим, чтобы принять результат.

В следующей программе демонстрируется использование алгоритмов *remove_copy()* и *replace_copy()*. При ее выполнении создается последовательность символов, из которой удаляются все буквы *'m'*. Затем выполняется замена всех букв *'o'* буквами *'X'*.

```
/* Демонстрация использования алгоритмов remove_copy и replace_copy.
*/
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  char str[] = "Это очень простой тест.";
  vector<char> v, v2(30);
  unsigned int i;

  for(i=0; str[i]; i++) v.push_back(str[i]);

  // **** демонстрация алгоритма remove_copy ****
  cout << "Входная последовательность: ";
  for(i=0; i<v.size(); i++)
    cout << v[i];
  cout << endl;

  // Удаляем все буквы 'т'.
  remove_copy(v.begin(), v.end(), v2.begin(), 'Т');
  cout << "После удаления букв 'т' : ";
  for(i=0; v2[i]; i++)
    cout << v2[i];
  cout << endl << endl;

  // **** Демонстрация алгоритма replace_copy ****
  cout << "Входная последовательность: ";
  for(i=0; i<v.size(); i++)
```

```
    cout << v[i];
  cout << endl;

  // Заменяем буквы 'о' буквами 'X'.
  replace_copy(v.begin(), v.end(), v2.begin(), 'о', 'X');
  cout << "После замены букв 'о' буквами 'X': ";
  for(i=0; v2[i]; i++)
    cout << v2[i];
  cout << endl << endl;

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Входная последовательность: Это очень простой тест.
После удаления букв 'т': Эо очень просой ес.

Входная последовательность: Это очень простой тест.
После замены букв 'о' буквами 'X': ЭтX Xчень прXстXй тест.
```

Реверсирование последовательности

В программах часто используется алгоритм *reverse()*, который в диапазоне, заданном параметрами *start* и *end*, меняет порядок следования элементов на противоположный. Его общий формат имеет следующий вид.

```
template <class BiIter>
void reverse(BiIter start, BiIter end);
```

В следующей программе демонстрируется использование этого алгоритма.

```
// Демонстрация использования алгоритма reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<int> v;
  unsigned int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Исходная последовательность: ";
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl;

  reverse (v.begin(), v.end());

  cout << "Реверсированная последовательность: ";
```

```
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Исходная последовательность: 0123456789
Реверсированная последовательность: 9876543210
```

Преобразование последовательности

Одним из самых интересных алгоритмов является *transform()*, поскольку он позволяет модифицировать каждый элемент в диапазоне в соответствии с заданной функцией. Алгоритм *transform()* используется в двух общих форматах.

```
template <class InIter, class OutIter, class Func>
OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);

template <class InIter1, class InIter2, class OutIter, class Func>
OutIter transform(InIter1 start1, InIter1 end1, InIter2 start2, OutIter result, Func binaryfunc);
```

Алгоритм *transform()* применяет функцию к диапазону элементов и сохраняет результат в последовательности, заданной параметром *result*. В первой форме диапазон задается параметрами *start* и *end*. Применяемая для преобразования функция задается параметром *unaryfunc*. Она принимает значение элемента в качестве параметра и должна возвратить преобразованное значение. Во второй форме алгоритма преобразование выполняется с использованием бинарной функции, которая принимает в качестве первого параметра значение предназначенного для преобразования элемента из последовательности, а в качестве второго параметра — элемент из второй последовательности. Обе версии возвращают итератор, указывающий на конец результирующей последовательности.

В следующей программе используется простая функция преобразования *xform()*, которая возводит в квадрат каждый элемент списка. Обратите внимание на то, что результирующая последовательность сохраняется в том же списке, который содержал исходную последовательность.

```
// Пример использования алгоритма transform.
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// Простая функция преобразования.
int xform(int i) {
  return i * i; // квадрат исходного значения
}

int main()
{
  list<int> x1;
  int i;

  // Помещаем значения в список.
  for(i=0; i<10; i++) x1.push_back(i);
```

```
  cout << "Исходный список x1: ";
  list<int>:: iterator p = x1.begin();
  while(p != x1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl;

  // Преобразование списка x1.
  p = transform(x1.begin(), x1.end(), x1.begin(), xform);

  cout << "Преобразованный список x1: ";
  p = x1.begin();
  while(p != x1.end()) {
    cout << *p << " ";
    p++;
  }

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Исходный список x1: 0 1 2 3 4 5 6 7 8 9
Преобразованный список x1: 0 1 4 9 16 25 36 49 64 81
```

Как видите, каждый элемент в списке *x1* теперь возведен в квадрат.

Исследование алгоритмов

Описанные выше алгоритмы представляют собой только малую часть всего содержимого библиотеки STL. И конечно же, вам стоит самим исследовать другие алгоритмы. Заслуживают внимания многие, например *set_union()* и *set_difference()*. Они предназначены для обработки содержимого такого контейнера, как множество. Интересны также алгоритмы *next_permutation()* и *prev_permutation()*. Они создают следующую и предыдущую перестановки элементов заданной последовательности. Время, затраченное на изучение алгоритмов библиотеки STL, — это время, потраченное не зря!

Класс string

Класс string обеспечивает альтернативу для строк с завершающим нулем.

Как вы знаете, C++ не поддерживает встроенный строковый тип. Однако он предоставляет два способа обработки строк. Во-первых, для представления строк можно использовать традиционный символьный массив с завершающим нулем. Строки, создаваемые таким способом (он вам уже знаком), иногда называют *С-строками*. Во-вторых, можно использовать объекты класса *string*, и именно этот способ рассматривается в данном разделе.

В действительности класс *string* представляет собой специализацию более общего шаблонного класса *basic_string*. Существует две специализации типа *basic_string*: тип *string*, который поддерживает 8-битовые символьные строки, и тип *wstring*, который поддерживает строки, образованные двухбайтовыми символами. Чаще всего в обычном программировании используются строковые объекты типа *string*. Для использования строковых классов C++ необходимо включить в программу заголовок <*string*>.

Прежде чем рассматривать класс *string*, важно понять, почему он является частью C++-библиотеки. Стандартные классы не сразу были добавлены в определение языка C++. На самом деле каждому нововведению предшествовали серьезные дискуссии и жаркие споры. При том, что C++ уже содержит поддержку строк в виде массивов с завершающим нулем, включение класса *string* в C++, на первый взгляд, может показаться исключением из этого правила. Но это далеко не так. И вот почему: строки с завершающим нулем нельзя обрабатывать стандартными C++-операторами, и их нельзя использовать в обычных C++-выражениях. Рассмотрим, например, следующий фрагмент кода.

```
char s1 [80], s2[80], s3[80];

s1 = "один"; // так делать нельзя

s2 = "два"; // так делать нельзя

s3 = s1 + s2; // ошибка
```

Как отмечено в комментариях, в C++ невозможно использовать оператор присваивания для придания символьному массиву нового значения (за исключением инструкции инициализации), а также нельзя применять оператор *"+"* для конкатенации двух строк. Эти операции можно выполнить с помощью библиотечных функций.

```
strcpy(s1, "one");

strcpy(s2, "two");

strcpy(s3, s1);

strcat(s3, s2);
```

Поскольку символьный массив с завершающим нулем формально не является самостоятельным типом данных, к нему нельзя применить C++-операторы. Это лишает *"изящества"* даже самые элементарные операции со строками. И именно неспособность обрабатывать строки с завершающим нулем с помощью стандартных C++-операторов привела к разработке стандартного строкового класса. Вспомните: создавая класс в C++, мы определяем новый тип данных, который можно полностью интегрировать в C++-среду. Это, конечно же, означает, что для нового класса можно перегружать операторы. Следовательно, вводя в язык стандартный строковый класс, мы создаем возможность для обработки строк так же, как и данных любого другого типа, а именно посредством операторов.

Однако существует еще одна причина, оправдывающая создание стандартного класса *string*: безопасность. Руками неопытного или неосторожного программиста очень легко обеспечить выход за границы массива, который содержит строку с завершающим нулем. Рассмотрим, например, стандартную функцию копирования *strcpy()*. Эта функция не предусматривает механизм проверки факта нарушения границ массива-приемника. Если исходный массив содержит больше символов, чем может принять массив-приемник, то в результате этой ошибки очень вероятен полный отказ системы. Как будет показано ниже, стандартный класс *string* не допускает возникновения подобных ошибок.

Итак, существует три причины для включения в C++ стандартного класса *string*: непротиворечивость данных (строка теперь определяется самостоятельным типом данных), удобство (программист может использовать стандартные C++-операторы) и безопасность (границы массивов отныне не будут нарушаться). Следует иметь в виду, что все выше перечисленное не означает, что вы должны отказываться от использования обычных строк с завершающим нулем. Они по-прежнему остаются самым эффективным средством реализации строк. Но если скорость не является для вас определяющим фактором, использование нового класса *string* даст вам доступ к безопасному и полностью интегрированному способу обработки строк.

И хотя класс *string* традиционно не воспринимается как часть библиотеки STL, он, тем не менее, представляет собой еще один контейнерный класс, определенный в C++. Это означает, что он поддерживает алгоритмы, описанные в предыдущем разделе. При этом строки имеют дополнительные возможности. Для получения доступа к классу *string* необходимо включить в программу заголовок *<string>*.

Класс *string* очень большой, он содержит множество конструкторов и функций-членов. Кроме того, многие функции-члены имеют несколько перегруженных форм. Поскольку в одной главе невозможно рассмотреть все содержимое класса *string*, мы обратим ваше внимание только на самые популярные его средства. Получив общее представление о работе класса *string*, вы сможете легко разобраться в остальных его возможностях самостоятельно.

Прототипы трех самых распространенных конструкторов класса *string* имеют следующий вид.

```
string( );
string(const char *str);
string (const string &str);
```

Первая форма конструктора создает пустой объект класса *string*. Вторая форма создает *string*-объект из строки с завершающим нулем, адресуемой параметром *str*. Эта форма конструктора обеспечивает преобразование из строки с завершающим нулем в объект типа *string*. Третья создает *string*-объект из другого *string*-объекта.

Для объектов класса *string* определены следующие операторы.

Оператор	Описание
=	Присваивание
+	Конкатенация
+=	Присваивание с конкатенацией
==	Равенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
[]	Индексация
<<	Вывод
>>	Ввод

Эти операторы позволяют использовать объекты типа *string* в обычных выражениях и избавляют программиста от необходимости вызывать такие функции, как *strcpy()* или *strcat()*. В общем случае в выражениях можно смешивать *string*-объекты и строки с завершающим нулем. Например, *string*-объект можно присвоить строке с завершающим нулем.

Оператор *"+"* можно использовать для конкатенации одного *string*-объекта с другим или *string*-объекта со строкой, созданной в *С-стиле* (С-строкой). Другими словами, поддерживаются следующие операции.

```
string-объект + string-объект
string-объект + С-строка
С-строка + string-объект
```

Оператор *"+"* позволяет также добавлять символ в конец строки.

В классе *string* определена константа *npos*, которая равна *-1*. Она представляет размер строки максимально возможной длины.

Строковый класс С++ существенно облегчает обработку строк. Например, используя *string*-объекты, можно применять оператор присваивания для назначения *string*-объекту строки в

объекты, можно применять оператор присваивания для назначения string-объекту строки в кавычках, оператор "+" — для конкатенации строк и операторы сравнения — для сравнения строк. Выполнение этих операций демонстрируется в следующей программе.

```
// Программа демонстрации обработки строк.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
  string str1("Класс string позволяет эффективно ");
  string str2("обрабатывать строки.");
  string str3;

  // Присваивание string-объекта.
  str3 = str1;
  cout << str1 << "\n" << str3 << "\n";

  // Конкатенация двух string-объектов.
  str3 = str1 + str2;
  cout << str3 << "\n";

  // Сравнение string-объектов.
  if(str3 > str1) cout << "str3 > str1\n";
  if(str3 == str1 + str2)
    cout << "str3 == str1+str2\n";

  /* Объекту класса string можно также присвоить обычную строку. */
  str1 = "Это строка с завершающим нулем.\n";
  cout << str1;

  /* Создание string-объекта с помощью другого string-объекта. */
  string str4 (str1);
  cout << str4;

  // Ввод строки.
  cout << "Введите строку: ";
    cin >> str4;
  cout << str4;

  return 0;
}
```

При выполнении эта программа генерирует такие результаты.

```
Класс string позволяет эффективно
Класс string позволяет эффективно
Класс string позволяет эффективно обрабатывать строки.
str3 > str1
str3 == str1+str2
```

```
str3 == str1+str2
Это строка с завершающим нулем.
Это строка с завершающим нулем.
Введите строку: Привет
Привет
```

Обратите внимание на то, как легко теперь выполняется обработка строк. Например, оператор *"+"* используется для конкатенации строк, а оператор *">"* — для сравнения двух строк. Для выполнения этих операций с использованием С-стиля обработки строк, т.е. использования строк с завершающим нулем, пришлось бы применять менее удобные средства, а именно вызывать функции *strcat()* и *strcmp()*. Поскольку С++-объекты типа *string* можно свободно смешивать с *С-строками*, их (*string*-объекты) можно использовать в любой программе не только безо всякого ущерба для эффективности, но даже с заметным выигрышем.

Важно также отметить, что в предыдущей программе размер строк не задается. Объекты типа *string* автоматически получают размер, нужный для хранения заданной строки. Таким образом, при выполнении операций присваивания или конкатенации строк строка-приемник увеличится по длине настолько, насколько это необходимо для хранения нового содержимого строки. При обработке *string*-объектов невозможно выйти за границы строки. Именно этот динамический аспект *string*-объектов выгодно отличает их от строк с завершающим нулем (которые часто страдают от нарушения границ).

Обзор функций-членов класса string

Если самые простые операции со строками можно реализовать с помощью операторов, то при выполнении более сложных не обойтись без функций-членов класса *string*. Класс *string* содержит слишком много функций-членов, мы же рассмотрим здесь только самые употребительные из них.

Важно! *Поскольку класс string— контейнер, он поддерживает такие обычные контейнерные функции, как begin(), end() и size().*

Основные манипуляции над строками

Чтобы присвоить одну строку другой, используйте функцию *assign()*. Вот как выглядят два возможных формата ее реализации.

```
string &assign(const string &strob, size_type start, size_type num);

string &assign(const char *str, size_type num);
```

Первый формат позволяет присвоить вызывающему объекту *num* символов из строки, заданной параметром *strob*, начиная с индекса *start*. При использовании второго формата вызывающему объекту присваиваются первые *num* символов строки с завершающим нулем, заданной параметром *str*. В каждом случае возвращается ссылка на вызывающий объект. Конечно, гораздо проще для присвоения одной полной строки другой использовать оператор *"="*. О функции-члене *assign()* вспоминают, в основном, тогда, когда нужно присвоить только часть строки.

С помощью функции-члена *append()* можно часть одной строки присоединить в конец другой. Два возможных формата ее реализации имеют следующий вид.

```
string &append(const string &strob, size_type start, size_type num);

string &append(const char *str, size_type num);
```

Здесь при использовании первого формата *num* символов из строки, заданной параметром *strob*, начиная с индекса *start*, будет присоединено в конец вызывающего объекта. Второй формат позволяет присоединить в конец вызывающего объекта первые *num* символов строки с завершающим нулем, заданной параметром *str*. В каждом случае возвращается ссылка на вызывающий объект. Конечно, гораздо проще для присоединения одной полной строки в конец другой использовать оператор. Функция же *append()* применяется тогда

строки в конец другой использовать оператор. Функция же *append()* применяется тогда, когда необходимо присоединить в конец вызывающего объекта только часть строки.

Вставку или замену символов в строке можно выполнять с помощью функций-членов *insert()* и *replace()*. Вот как выглядят прототипы их наиболее употребительных форматов.

```
string &insert(size_type start, const string &strob);

string &insert(size_type start, const string &strob, size_type insStart,
size_type num);

string &replace(size_type start, size_type num, const string &strob);

string &replace(size_type start, size_type orgNum, const string &strob,
size_type replaceStart, size_type replaceNum);
```

Первый формат функции *insert()* позволяет вставить строку, заданную параметром *strob*, в позицию вызывающей строки, заданную параметром *start*. Второй формат функции *insert()* предназначен для вставки *num* символов из строки, заданной параметром *strob*, начиная с индекса *insStart*, в позицию вызывающей строки, заданную параметром *start*.

Первый формат функции *replace()* служит для замены num символов в вызывающей строке, начиная с индекса *start*, строкой, заданной параметром *strob*. Второй формат позволяет заменить orgNum символов в вызывающей строке, начиная с индекса *start*, *replaceNum* символами строки, заданной параметром *strob*, начиная с индекса *replaceStart*. В каждом случае возвращается ссылка на вызывающий объект.

Удалить символы из строки можно с помощью функции *erase()*. Один из ее форматов выглядит так:

```
string &erase(size_type start = 0, size_type num = npos);
```

Эта функция удаляет *num* символов из вызывающей строки, начиная с индекса *start*. Функция возвращает ссылку на вызывающий объект.

Использование функций *insert()*, *erase()* и *replace()* демонстрируется в следующей программе.

```
// Демонстрация использования функций insert(), erase() и replace().
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1("Это простой тест.");
  string str2("ABCDEFG");

  cout << "Исходные строки:\n";
  cout << "str1: " << str1 << endl;
  cout << "str2: " << str2 << "\n\n";

  // Демонстрируем использование функции insert().
  cout << "Вставляем строку str2 в строку str1:\n";
  str1.insert(5, str2);
  cout << str1 << "\n\n";

  // Демонстрируем использование функции erase().
```

```
  cout << "Удаляем 7 символов из строки str1:\n";
  str1.erase(5, 7);
  cout << str1 <<"\n\n";

  // Демонстрируем использование функции replace().
  cout << "Заменяем 2 символа в str1 строкой str2:\n";
  str1.replace(5, 2, str2);
  cout << str1 << endl;

  return 0;
}
```

Результаты выполнения этой программы таковы.

```
Исходные строки:
str1: Это простой тест.
str2: ABCDEFG

Вставляем строку str2 в строку str1:
Это пABCDEFGростой тест.

Удаляем 7 символов из строки str1:
Это простой тест.

Заменяем 2 символа в str1 строкой str2:
Это пABCDEFGстой тест.
```

Поиск в строке

В классе *string* предусмотрено несколько функций-членов, которые осуществляют поиск. Это, например, такие функции, как *find()* и *rfind()*. Рассмотрим прототипы самых употребительных версий этих функций.

```
size_type find(const string &strob, size_type start=0) const;

size_type rfind(const string &strob, size_type start=npos) const;
```

Функция *find()*, начиная с позиции *start*, просматривает вызывающую строку на предмет поиска первого вхождения строки, заданной параметром *strob*. Если поиск успешен, функция *find()* возвращает индекс, по которому в вызывающей строке было обнаружено совпадение. Если совпадения не обнаружено, возвращается значение *npos*. Функция *rfind()* выполняет то же действие, но с конца. Начиная с позиции *start*, она просматривает вызывающую строку в обратном направлении на предмет поиска первого вхождения строки, заданной параметром *strob* (т.е. она находит в вызывающей строке последнее вхождение строки, заданной параметром *strob*). Если поиск прошел удачно, функция *rfind()* возвращает индекс, по которому в вызывающей строке было обнаружено совпадение. Если совпадения не обнаружено, возвращается значение *npos*.

Рассмотрим короткий пример использования функции *find()*.

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

int main()
{
  int i;
```

```
  string s1 ="Класс string облегчает обработку строк.";
  string s2;

  i = s1.find("string");
  if(i != string::npos) {
    cout << "Совпадение обнаружено в позиции " << i<< endl;
    cout << "Остаток строки таков: ";
    s2.assign (s1, i, s1.size());
    cout << s2;
  }

  return 0;
}
```

Программа генерирует такие результаты.

```
Совпадение обнаружено в позиции 6
Остаток строки таков: string облегчает обработку строк.
```

Сравнение строк

Чтобы сравнить полное содержимое одного *string*-объекта с другим, обычно используются описанные выше перегруженные операторы отношений. Но если нужно сравнить часть одной строки с другой, вам придется использовать функцию-член *compare()*.

```
int compare(size_type start, size_type num, const string &strob) const;
```

Функция *compare()* сравнивает с вызывающей строкой *num* символов строки, заданной параметром *strob*, начиная с индекса *start*. Если вызывающая строка меньше строки *strob*, функция *compare()* возвратит отрицательное значение. Если вызывающая строка больше строки *strob*, она возвратит положительное значение. Если строка *strob* равна вызывающей строке, функция *compare()* возвратит нуль.

Получение строки с завершающим нулем

Несмотря на неоспоримую полезность объектов типа *string*, возможны ситуации, когда вам придется получать из такого объекта символьный массив с завершающим нулем, т.е. его версию С-строки. Например, вы могли бы использовать *string*-объект для создания имени файла. Но, открывая файл, вам нужно задать указатель на стандартную строку с завершающим нулем. Для решения этой проблемы и используется функция-член *c_str()*. Вот как выглядит ее прототип:

```
const char *c_str( ) const;
```

Эта функция возвращает указатель на *С-версию* строки (т.е. на строку с завершающим нулевым символом), которая содержится в вызывающем объекте типа *string*. Полученная строка с завершающим нулем изменению не подлежит. Кроме того, после выполнения других операций над этим *string*-объектом допустимость применения полученной *С-строки* не гарантируется.

Хранение строк в других контейнерах

Поскольку класс *string* определяет тип данных, можно создать контейнеры, которые будут содержать объекты типа *string*. Рассмотрим, например, более удачный вариант программы-словаря, которая была показана выше.

```
/* Использование отображения string-объектов для создания словаря.
*/
#include <iostream>
```

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

int main( )
{
  map<string, string> dictionary;

  dictionary.insert( pair<string, string> ("дом", "Место проживания."));

   dictionary.insert(  pair<string,  string>  ("клавиатура",  "Устройство ввода данных."));

  dictionary.insert( pair<string, string> ("программирование", "Процесс создания программы."));

   dictionary.insert(  pair<string,  string>  ("STL",  "Standard  Template Library"));

  string s;
  cout << "Введите слово: ";
    cin >> s;

  map<string, string>::iterator p;
  p = dictionary.find(s);
  if(p != dictionary.end( ))
    cout << "Определение: " << p->second;
  else cout << "Такого слова в словаре нет.\n";

  return 0;
}
```

И еще об STL

Библиотека STL — важная составляющая языка C++. Многие задачи программирования можно описать, используя терминологию STL. Эта библиотека великолепно сочетает силу своих средств с гибкостью их применения. Несмотря на то что ее синтаксис немного сложноват, он быстро осваивается. Ни один уважающий себя C++-программист не может пренебречь возможностями библиотеки STL, поскольку она — не только настоящее, но и будущее C++-программирования.

Глава 22: Препроцессор C++

Заключительная глава книги посвящена описанию препроцессора C++. Препроцессор C++ — это часть компилятора, которая подвергает вашу программу различным текстовым преобразованиям до реальной трансляции исходного кода в объектный. Программист может давать препроцессору команды, называемые *директивами препроцессора* (preprocessor directives), которые, не являясь формальной частью языка C++, способны расширить область действия его среды программирования.

Препроцессор C++ включает следующие директивы.

#define	#error	#include
#if	#else	#elif
#endif	#ifdef	#ifndef
#undef	#line	#pragma

#undef	#line	#pragma

Как видите, все директивы препроцессора начинаются с символа *'#'*.Теперь рассмотрим каждую из них в отдельности.

На заметку. *Препроцессор C++ — прямой потомок препроцессора C, и некоторые его средства оказались избыточными после введения в C++ новых элементов. Однако он по-прежнему является важной частью C++-среды программирования*.

Директива #define

Директива *#define* определяет имя макроса.

Директива *#define* используется для определения идентификатора и символьной последовательности, которая будет подставлена вместо идентификатора везде, где он встречается в исходном коде программы. Этот идентификатор называется макроименем, а процесс замены — *макроподстановкой* (реализацией макрорасширения). Общий формат использования этой директивы имеет следующий вид.

```
#define макроимя последовательность_символов
```

Обратите внимание на то, что здесь нет точки с запятой. Заданная *последовательность_символов* завершается только символом конца строки. Между элементами *макроимя* (имя_макроса) и *последовательность_символов* может быть любое количество пробелов.

Итак, после включения этой директивы каждое вхождение текстового фрагмента, определенное как *макроимя*, заменяется заданным элементом *последовательность_символов*. Например, если вы хотите использовать слово *UP* в качестве значения *1* и слово *DOWN* в качестве значения *0*, объявите такие директивы *#define*.

```
#define UP 1
#define DOWN 0
```

Данные директивы вынудят компилятор подставлять *1* или *0* каждый раз, когда в файле исходного кода встретится слово *UP* или *DOWN* соответственно. Например, при выполнении инструкции:

```
cout << UP << ' ' << DOWN << ' ' << UP + UP;
```

На экран будет выведено следующее:

```
1 0 2
```

После определения имени макроса его можно использовать как часть определения других макроимен. Например, следующий код определяет имена *ONE*, *TWO* и *THREE* и соответствующие им значения.

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

Важно понимать, что *макроподстановка* — это просто замена идентификатора соответствующей строкой. Следовательно, если вам нужно определить стандартное сообщение, используйте код, подобный этому.

```
#define GETFILE "Введите имя файла"
  // ...
```

Препроцессор заменит строкой *"Введите имя файла"* каждое вхождение идентификатора *GETFILE*. Для компилятора эта cout-инструкция

```
cout << GETFILE;
```

в действительности выглядит так.

```
cout << "Введите имя файла";
```

Никакой текстовой замены не произойдет, если идентификатор находится в строке, заключенной в кавычки. Например, при выполнении следующего кода

```
#define GETFILE "Введите имя файла"
  // ...
cout << "GETFILE - это макроимя\n";
```

на экране будет отображена эта информация

```
GETFILE - это макроимя,
```

а не эта:

```
Введите имя файла - это макроимя
```

Если текстовая последовательность не помещается на строке, ее можно продолжить на следующей, поставив обратную косую черту в конце строки, как показано в этом примере.

```
#define LONG_STRING "Это очень длинная последовательность,\
  которая используется в качестве примера."
```

Среди C++-программистов принято использовать для макроимен прописные буквы. Это соглашение позволяет с первого взгляда понять, что здесь используется макроподстановка. Кроме того, лучше всего поместить все директивы *#define* в начало файла или включить в отдельный файл, чтобы не искать их потом по всей программе.

Макроподстановки часто используются для определения *"магических чисел"* программы. Например, у вас есть программа, которая определяет некоторый массив, и ряд функций, которые получают доступ к нему. Вместо *"жесткого"* кодирования размера массива с помощью константы лучше определить имя, которое бы представляло размер, а затем использовать это имя везде, где должен стоять размер массива. Тогда, если этот размер придется изменить, вам достаточно будет внести только одно изменение, а затем перекомпилировать программу. Рассмотрим пример.

```
#define MAX_SIZE 100
  // ...
float balance[MAX_SIZE];
double index[MAX_SIZE];
int num_emp[MAX_SIZE];
```

Важно! *Важно помнить, что в C++ предусмотрен еще один способ определения констант, который заключается в использовании спецификатора const. Однако многие программисты "пришли" в C++ из C-среды, где для этих целей обычно использовалась директива #define. Поэтому вам еще часто придется с ней сталкиваться в C++-коде.*

Макроопределения, действующие как функции

Директива *#define* имеет еще одно назначение: макроимя может использоваться с аргументами. При каждом вхождении макроимени связанные с ним аргументы заменяются реальными аргументами, указанными в коде программы. Такие макроопределения действуют подобно функциям. Рассмотрим пример.

```
/* Использование "функциональных" макроопределений. */
#include <iostream>
using namespace std;

#define MIN(a, b) (((a)<(b)) ? a : b)

int main()
{
  int x, y;
```

```
  int x, y;

  x = 10;
  y = 20;

  cout << "Минимум равен: " << MIN(x, y);

  return 0;
}
```

При компиляции этой программы выражение, определенное идентификатором *MIN (a, b)*, будет заменено, но *x* и *y* будут рассматриваться как операнды. Это значит, что cout-инструкция после компиляции будет выглядеть так.

```
cout << "Минимум равен: " << (((x)<(y)) ? x : y);
```

По сути, такое макроопределение представляет собой способ определить функцию, которая вместо вызова позволяет раскрыть свой код в строке.

Макроопределения, действующие как функции, — *это макроопределения, которые принимают аргументы*.

Кажущиеся избыточными круглые скобки, в которые заключено макроопределение *MIN*, необходимы, чтобы гарантировать правильное восприятие компилятором заменяемого выражения. На самом деле дополнительные круглые скобки должны применяться практически ко всем макроопределениям, действующим подобно функциям. Нужно всегда очень внимательно относиться к определению таких макросов; в противном случае возможно получение неожиданных результатов. Рассмотрим, например, эту короткую программу, которая использует макрос для определения четности значения.

```
// Эта программа дает неверный ответ.
#include <iostream>
using namespace std;

#define EVEN(a) a%2==0 ? 1 : 0
int main( )
{
  if(EVEN(9+1)) cout << "четное число";
  else cout << "нечетное число ";
  return 0;
}
```

Эта программа не будет работать корректно, поскольку не обеспечена правильная подстановка значений. При компиляции выражение *EVEN(9+1)* будет заменено следующим образом.

```
9+1%2==0 ? 1 : 0
```

Напомню, что оператор *"%"* имеет более высокий приоритет, чем оператор *"+"*. Это значит, что сначала выполнится операция деления по модулю (*%*) для числа *1*, а затем ее результат будет сложен с числом *9*, что (конечно же) не равно *0*. Чтобы исправить ошибку, достаточно заключить в круглые скобки аргумент *a* в макроопределении *EVEN*, как показано в следующей (исправленной) версии той же программы.

```
// Эта программа работает корректно.
#include <iostream>
using namespace std;
```

```
#define EVEN(a) (a)%2==0 ? 1 : 0

int main()
{
  if(EVEN(9+1)) cout << "четное число";
  else cout << "нечетное число";

  return 0;
}
```

Теперь сумма *9+1* вычисляется до выполнения операции деления по модулю. В общем случае лучше всегда заключать параметры макроопределения в круглые скобки, чтобы избежать непредвиденных результатов, подобных описанному выше.

Использование макроопределений вместо настоящих функций имеет одно существенное достоинство: поскольку код макроопределения расширяется в строке, и нет никаких затрат системных ресурсов на вызов функции, скорость работы вашей программы будет выше по сравнению с применением обычной функции. Но повышение скорости является платой за увеличение размера программы (из-за дублирования кода функции).

Важно! *Несмотря на то что макроопределения все еще встречаются в C++-коде, макросы, действующие подобно функциям, можно заменить спецификатором inline, который справляется с той же ролью лучше и безопаснее. (Вспомните: спецификатор inline обеспечивает вместо вызова функции расширение ее тела в строке.) Кроме того, inline-функции не требуют дополнительных круглых скобок, без которых не могут обойтись макроопределения. Однако макросы, действующие подобно функциям, все еще остаются частью C++-программ, поскольку многие C/C++-программисты продолжают использовать их по привычке.*

Директива #error

Директива #error отображает сообщение об ошибке.

Директива *#error* дает указание компилятору остановить компиляцию. Она используется в основном для отладки. Общий формат ее записи таков.

```
#error сообщение
```

Обратите внимание на то, что элемент *сообщение* не заключен в двойные кавычки. При встрече с директивой *#error* отображается заданное *сообщение* и другая информация (она зависит от конкретной реализации рабочей среды), после чего компиляция прекращается. Чтобы узнать, какую информацию отображает в этом случае компилятор, достаточно провести эксперимент.

Директива #include

Директива *#include* включает заголовок или другой исходный файл.

Директива препроцессора *#include* обязывает компилятор включить либо стандартный заголовок, либо другой исходный файл, имя которого указано в директиве *#include*. Имя стандартных заголовков заключается в угловые скобки, как показано в примерах, приведенных в этой книге. Например, эта директива

```
#include <vector>
```

Включает стандартный заголовок для векторов.

При включении другого исходного файла его имя может быть указано в двойных кавычках или угловых скобках. Например, следующие две директивы обязывают C++ прочитать и скомпилировать файл с именем *sample.h*:

```
#include <sample.h>
#include "sample.h"
```

Если имя файла заключено в угловые скобки, то поиск файла будет осуществляться в одном или нескольких специальных каталогах, определенных конкретной реализацией.

Если же имя файла заключено в кавычки, поиск файла выполняется, как правило, в текущем каталоге (что также определено конкретной реализацией). Во многих случаях это означает поиск текущего рабочего каталога. Если заданный файл не найден, поиск повторяется с использованием первого способа (как если бы имя файла было заключено в угловые скобки). Чтобы ознакомиться с подробностями, связанными с различной обработкой директивы *#include* в случае использования угловых скобок и двойных кавычек, обратитесь к руководству пользователя, прилагаемому к вашему компилятору. Инструкции *#include* могут быть вложенными в другие включаемые файлы.

Директивы условной компиляции

Существуют директивы, которые позволяют избирательно компилировать части исходного кода. Этот процесс, именуемый *условной компиляцией*, широко используется коммерческими фирмами по разработке программного обеспечения, которые создают и поддерживают множество различных версий одной программы.

Директивы #if, #else, #elif и #endif

Директивы #if, #ifdef, #ifndef, #else, #elif и #endif — *это директивы условной компиляции*.

Главная идея состоит в том, что если выражение, стоящее после директивы *#if* оказывается истинным, то будет скомпилирован код, расположенный между нею и директивой *#endif* в противном случае данный код будет опущен. Директива *#endif* используется для обозначения конца блока *#if*.

Общая форма записи директивы *#if* выглядит так.

```
#if константное_выражение
  последовательность инструкций
#endif
```

Если *константное_выражение* является истинным, код, расположенный непосредственно за этой директивой, будет скомпилирован. Рассмотрим пример.

```
// Простой пример использования директивы #if.
#include <iostream>
using namespace std;

#define MAX 100
int main( )
{
  #if MAX>10
    cout << "Требуется дополнительная память\n";
  #endif

  // ...
  return 0;
}
```

При выполнении эта программа отобразит сообщение *Требуется дополнительная память* на экране, поскольку, как определено в программе, значение константы *MAX* больше 10. Этот пример иллюстрирует важный момент: Выражение, которое стоит после директивы *#if*, вычисляется во время компиляции. Следовательно, оно должно содержать только идентификаторы, которые были предварительно определены, или константы. Использование же переменных здесь исключено.

Поведение директивы *#else* во многом подобно поведению инструкции *else*, которая является частью языка C++: она определяет альтернативу на случай невыполнения

является частью языка C++. Она определяет альтернативу на случай невыполнения директивы *#if*. Чтобы показать, как работает директива *#else*, воспользуемся предыдущим примером, немного его расширив.

```
// Пример использования директив #if/#else.
#include <iostream>
using namespace std;

#define MAX 6
int main( )
{
  #if MAX>10
    cout << "Требуется дополнительная память.\n");
  #else
    cout << "Достаточно имеющейся памяти.\n";
  #endif

  // . . .
  return 0;
}
```

В этой программе для имени *MAX* определено значение, которое меньше 10, поэтому *#if*-ветвь кода не скомпилируется, но зато скомпилируется альтернативная *#else*-ветвь. В результате отобразится сообщение *Достаточно имеющейся памяти*..

Обратите внимание на то, что директива *#else* используется для индикации одновременно как конца *#if*-блока, так и начала *#else*-блока. В этом есть логическая необходимость, поскольку только одна директива *#endif* может быть связана с директивой *#if*.

Директива *#elif* эквивалентна связке инструкций *else-if* и используется для формирования многозвенной схемы *if-else-if*, представляющей несколько вариантов компиляции. После директивы *#elif* должно стоять константное выражение. Если это выражение истинно, следующий блок кода скомпилируется, и никакие другие *#elif*-выражения не будут тестироваться или компилироваться. В противном случае будет проверено следующее по очереди *#elif*-выражение. Вот как выглядит общий формат использования директивы *#elif*.

```
#if выражение
  последовательность инструкций
#elif выражение 1
  последовательность инструкций
#elif выражение 2
  последовательность инструкций
#elif выражение 3
  последовательность инструкций
// . . .
#elif выражение N
  последовательность инструкций
#endif
```

Например, в этом фрагменте кода используется идентификатор *COMPILED_BY*, который позволяет определить, кем компилируется программа.

```
#define JOHN 0
#define BOB 1
#define TOM 2

#define COMPILED_BY JOHN

#if COMPILED_BY == JOHN
  char who[] = "John";
#elif COMPILED_BY == BOB
  char who[] = "Bob";
#else
  char who[] = "Tom";
#endif
```

Директивы *#if* и *#elif* могут быть вложенными. В этом случае директива *#endif*, *#else* или *#elif* связывается с ближайшей директивой *#if* или *#elif*. Например, следующий фрагмент кода совершенно допустим.

```
#if COMPILED_BY == BOB
  #if DEBUG == FULL
    int port = 198;
  #elif DEBUG == PARTIAL
    int port = 200;
  #endif
#else
   cout << "Боб должен скомпилировать код" << "для отладки вывода данных.\n";
#endif
```

Директивы #ifdef и #ifndef

Директивы *#ifdef* и *#ifndef* предлагают еще два варианта условной компиляции, которые можно выразить как *"если определено"* и *"если не определено"* соответственно.

Общий формат использования директивы *#ifdef* таков.

```
#ifdef макроимя
  последовательность инструкций
#endif
```

Если *макроимя* предварительно определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.

Общий формат использования директивы *#ifndef* таков.

```
#ifndef макроимя
  последовательность инструкций
#endif
```

Если *макроимя* не определено с помощью какой-нибудь директивы *#define*, то *последовательность инструкций*, расположенная между директивами *#ifdef* и *#endif*, будет скомпилирована.

Как директива *#ifdef*, так и директива *#ifndef* может иметь директиву *#else* или *#elif*. Рассмотрим пример

Рассмотрим пример.

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

#define TOM

int main( )
{
  #ifdef TOM
    cout << "Программист Том.\n";
  #else
    cout << "Программист неизвестен.\n";
  #endif

  #ifndef RALPH
    cout << "Имя RALPH не определено.\n";
  #endif

  return 0;
}
```

При выполнении эта программа отображает следующее.

```
Программист Том.
Имя RALPH не определено.
```

Но если бы идентификатор *TOM* был не определен, то результат выполнения этой программы выглядел бы так.

```
Программист неизвестен.
Имя RALPH не определено.
```

И еще. Директивы *#ifdef* и *#ifndef* можно вкладывать точно так же, как и директивы *#if*.

Директива #undef

Директива *#undef* используется для удаления предыдущего определения некоторого макроимени. Ее общий формат таков.

```
#undef макроимя
```

Рассмотрим пример.

```
#define TIMEOUT 100
#define WAIT 0

  // . . .

#undef TIMEOUT
#undef WAIT
```

Здесь имена *TIMEOUT* и *WAIT* определены до тех пор, пока не выполнится директива *#undef*.

Основное назначение директивы *#undef* — разрешить локализацию макроимен для тех частей кода, в которых они нужны.

Использование оператора defined

Помимо директивы *#ifdef* существует еще один способ выяснить, определено ли в программе некоторое макроимя. Для этого можно использовать директиву *#if* в сочетании с

программе некоторое макроимя. Для этого можно использовать директиву *#if* в сочетании с оператором времени компиляции *defined*. Например, чтобы узнать, определено ли макроимя *MYFILE*, можно использовать одну из следующих команд препроцессорной обработки.

```
#if defined MYFILE
```

или

```
#ifdef MYFILE
```

При необходимости, чтобы реверсировать условие проверки, можно предварить оператор *defined* символом *"!"*. Например, следующий фрагмент кода скомпилируется только в том случае, если макроимя *DEBUG* не определено.

```
#if !defined DEBUG
  cout << "Окончательная версия!\n";
#endif
```

О роли препроцессора

Препроцессор C++ — прямой потомок препроцессора языка C, причем без каких-либо усовершенствований. Однако его роль в C++ намного меньше роли, которую играет препроцессор в C. Дело в том, что многие задачи, выполняемые препроцессором в C, реализованы в C++ в виде элементов языка. Страуструп тем самым выразил свое намерение сделать функции препроцессора ненужными, чтобы в конце концов от него можно было бы совсем освободить язык.

На данном этапе препроцессор уже частично избыточен. Например, два наиболее употребительных свойства директивы *#define* были заменены инструкциями C++. В частности, ее способность создавать константное значение и определять макроопределение, действующее подобно функциям, сейчас совершенно избыточна. В C++ есть более эффективные средства для выполнения этих задач. Для создания константы достаточно определить *const*-переменную. А с созданием встраиваемой (подставляемой) функции вполне справляется спецификатор *inline*. Оба эти средства лучше работают, чем соответствующие механизмы директивы *#define*.

Приведем еще один пример замены элементов препроцессора элементами языка. Он связан с использованием однострочного комментария. Одна из причин его создания — разрешить "превращение" кода в комментарий. Как вы знаете, комментарий, использующий */*...*/*-стиль, не может быть вложенным. Это означает, что фрагменты кода, содержащие */*...*/*-комментарии, одним махом "превратить в комментарий" нельзя. Но это можно сделать с *//*-комментариями, окружив их */*...*/*-символами комментария. Возможность "превращения" кода в комментарий делает использование таких директив условной компиляции, как *#ifdef*, частично избыточным.

Директива #line

Директива #line изменяет содержимое псевдопеременных _ _LINE_ _ и _ _FILE_ _.

Директива *#line* используется для изменения содержимого псевдопеременных *_ _LINE_ _* и *_ _FILE_ _*, которые являются зарезервированными идентификаторами (макроименами). Псевдопеременная *_ _LINE_ _* содержит номер скомпилированной строки, а псевдопеременная *_ _FILE_ _*— имя компилируемого файла. Базовая форма записи этой команды имеет следующий вид.

```
#line номер "имя_файла"
```

Здесь *номер* — это любое положительное целое число, а *имя_файла* — любой допустимый идентификатор файла. Значение элемента *номер* становится номером текущей исходной строки, а значение элемента *имя_файла*— именем исходного файла. *Имя_файла*— элемент необязательный. Директива *#line* используется, главным образом, в целях отладки и в специальных приложениях.

Например, следующая программа обязывает начинать счет строк с числа 200. Инструкция

cout отображает номер 202, поскольку это — третья строка в программе после директивной инструкции *#line 200*.

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

#line 200 // Устанавливаем счетчик строк равным 200.
int main( ) // Эта строка сейчас имеет номер 200.
{// Номер этой строки равен 201.
  cout << _ _LINE_ _;// Здесь выводится номер 202.
  return 0;
}
```

Директива #pragma

Директива #pragma зависит от конкретной реализации компилятора.

Работа директивы *#pragma* зависит от конкретной реализации компилятора. Она позволяет выдавать компилятору различные инструкции, предусмотренные создателем компилятора. Общий формат его использования таков.

```
#pragma имя
```

Здесь элемент *имя* представляет имя желаемой *#pragma*-инструкции. Если указанное имя не распознается компилятором, директива *#pragma* попросту игнорируется без сообщения об ошибке.

Важно! *Для получения подробной информации о возможных вариантах использования директивы #pragma стоит обратиться к системной документации по используемому вами компилятору. Вы можете найти для себя очень полезную информацию. Обычно #pragma-инструкции позволяют определить, какие предупреждающие сообщения выдает компилятор, как генерируется код и какие библиотеки компонуются с вашими программами*.

Операторы препроцессора "#" и "##"

В C++ предусмотрена поддержка двух операторов препроцессора: *"#"* и *"##"*. Эти операторы используются совместно с директивой *#define*. Оператор *"#"* преобразует следующий за ним аргумент в строку, заключенную в кавычки. Рассмотрим, например, следующую программу.

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

#define mkstr(s) # s
int main( )
{
  cout << mkstr(Я в восторге от C++);

  return 0;
}
```

Препроцессор C++ преобразует строку

```
cout << mkstr(Я в восторге от C++);
```

в строку

```
cout << "Я в восторге от C++";
```

Оператор используется для конкатенации двух лексем. Рассмотрим пример.

```
#include <iostream>
```

```
using namespace std;

#define concat(a, b) a ## b
int main( )
{
  int xy = 10;

  cout << concat(x, y);

  return 0;
}
```

Препроцессор преобразует строку

```
cout << concat (x, y);
```

в строку

```
cout << xy;
```

Если эти операторы вам кажутся странными, помните, что они не являются операторами "повседневного спроса" и редко используются в программах. Их основное назначение — позволить препроцессору обрабатывать некоторые специальные ситуации.

Зарезервированные макроимена

В языке C++ определено шесть встроенных макроимен.

```
_ _LINE_ _
_ _FILE_ _
_ _DATE_ _
_ _TIME_ _
_ _STDC_ _
_ _cplusplus
```

Рассмотрим каждое из них в отдельности.

Макросы *_ _LINE_ _* и *_ _FILE_ _* описаны при рассмотрении директивы *#line* выше в этой главе. Они содержат номер текущей строки и имя файла компилируемой программы.

Макрос *_ _DATE_ _* представляет собой строку в формате *месяц/день/год*, которая означает дату трансляции исходного файла в объектный код.

Время трансляции исходного файла в объектный код содержится в виде строки в макросе *_ _TIME_ _*. Формат этой строки следующий: *часы.минуты.секунды*.

Точное назначение макроса *_ _STDC_ _* зависит от конкретной реализации компилятора. Как правило, если макрос *_ _STDC_ _* определен, то компилятор примет только стандартный C/C++-код, который не содержит никаких нестандартных расширений.

Компилятор, соответствующий ANSI/ISO-стандарту C++, определяет макрос *_ _cplusplus* как значение, содержащее по крайней мере шесть цифр. "Нестандартные" компиляторы должны использовать значение, содержащее пять (или даже меньше) цифр.

Мысли "под занавес"

Мы преодолели немалый путь: длиной в целую книгу. Если вы внимательно изучили все приведенные здесь примеры, то можете смело назвать себя программистом на C++. Подобно многим другим наукам, программирование лучше всего осваивать на практике, поэтому теперь вам нужно писать побольше программ. Полезно также разобраться в C++-программах, написанных другими (причем разными) профессиональными программистами. При этом важно обращать внимание на то, как программа оформлена и

реализована. Постараитесь наити в них как достоинства, так и недостатки. Это расширит диапазон ваших представлений о программировании. Подумайте также над тем, как можно улучшить существующий код, применив контейнеры и алгоритмы библиотеки STL. Эти средства, как правило, позволяют улучшить читабельность и поддержку больших программ. Наконец, просто больше экспериментируйте! Дайте волю своей фантазии и вскоре вы почувствуете себя настоящим C++-программистом!

Для продолжения теоретического освоения C++ предлагаю обратиться к моей книге *Полный справочник по C++*, М. : Издательский дом "Вильямс". Она содержит подробное описание элементов языка C++ и библиотек.

Приложение А: С-ориентированная система ввода-вывода

Это приложение содержит краткое описание С-системы ввода-вывода. Несмотря на то что вы предполагаете использовать C++-систему ввода-вывода, есть ряд причин, по которым вам все-таки следует понимать основы С-ориентированной системы ввода-вывода. Во-первых, если вам придется работать с С-кодом (особенно, если возникнет необходимость его перевода в C++-код), то вам нужно знать, как работает С-система ввода-вывода. Во-вторых, часто в одной и той же программе используются как С-, так и C++-операции ввода-вывода. Это особенно характерно для очень больших программ, отдельные части которых писались разными программистами в течение довольно длительного периода времени. В-третьих, большое количество существующих С-программ продолжают находиться в эксплуатации и нуждаются в поддержке. Наконец, многие книги и периодические издания содержат программы, написанные на С. Чтобы понимать эти С-программы, необходимо понимать основы функционирования С-системы ввода-вывода.

Узелок на память. *Для C++-программ необходимо использовать объектно-ориентированную C++-систему ввода-вывода*.

В этом приложении описаны наиболее употребительные С-ориентированные функции ввода-вывода. Однако стандартная С-библиотека содержит такое огромное количество функций ввода-вывода, что мы не в силах рассмотреть их здесь в полном объеме. Если же вам придется серьезно погрузиться в С-программирование, то рекомендую обратиться к справочной литературе.

Система ввода-вывода языка С требует включать в программы заголовочный файл *stdio.h* (ему соответствует заголовок *<cstdio>*, отвечающий новому стилю). Каждая С-программа должна использовать заголовочный файл *stdio.h*, поскольку язык С не поддерживает C++-стиль включения заголовков. C++-программа может работать с использованием любого из этих двух вариантов. Заголовок *<cstdio>* помещает свое содержимое в пространство имен *std*, а заголовочный файл *stdio.h*— в глобальное пространство имен, что соответствует С-ориентации. В этом приложении в качестве примеров приведены С-программы, поэтому они используют С-стиль включения заголовочного файла *stdio.h* и не требуют установки пространства имен.

И еще. Как отмечалось в главе 1, стандарт языка С был обновлен в 1999 году и получил название стандарта С99. В то время в С-систему ввода-вывода было внесено несколько усовершенствований. Но поскольку C++ опирается на стандарт С89, то он не поддерживает средств, которые были добавлены в стандарт С99. (Более того, на момент написания этой книги ни один из широко доступных компиляторов C++ не поддерживал стандарт С99. Да и ни одна из широко распространяемых программ не использовала средства стандарта С99.) Поэтому здесь не описываются средства, внесенные в С-систему ввода-вывода стандартом С99. Если же вас интересует язык С, включая полное описание его системы ввода-вывода и средств, добавленных стандартом С99, я рекомендую обратиться к моей книге *Полный справочник по С*, М.: Издательский дом "Вильямс".

Использование потоков в С-системе ввода-вывода

Подобно C++-системе ввода-вывода, С-ориентированная система ввода-вывода опирается на понятие потока. В начале работы программы автоматически открываются три заранее определенных текстовых потока: *stdin*, *stdout* и *stderr*. Их называют стандартными *потоками ввода данных* (входной поток), *вывода данных* (выходной поток) и *ошибок* соответственно

ввода данных (входной поток), *вывода данных* (выходной поток) и *ошибок* соответственно. (Некоторые компиляторы открывают также и другие потоки, которые зависят от конкретной реализации системы.) Эти потоки представляют собой *C-версии* потоков *cin*, *cout* и *cerr* соответственно. По умолчанию они связаны с соответствующим системным устройством.

Поток	Устройство
stdin	клавиатура
stdout	экран
stderr	экран

Помните, что большинство операционных систем, включая Windows, позволяют перенаправлять потоки ввода-вывода, поэтому функции чтения и записи данных можно перенаправлять на другие устройства. Никогда не пытайтесь явно открывать или закрывать эти потоки.

Каждый поток, связанный с файлом, имеет структуру управления файлом типа *FILE*. Эта структура определена в заголовочном файле *stdio.h*. Вы не должны модифицировать содержимое этого блока управления файлом.

Функции printf() и scanf()

Двумя самыми популярными C-функциями ввода-вывода являются *printf()* и *scanf()*. Функция *printf()* записывает данные в стандартное устройство вывода (консоль), а функция *scanf()*, ее дополнение, считывает данные с клавиатуры. Поскольку язык C не поддерживает перегрузку операторов и не использует операторы *"<<"* и *">>"* в качестве операторов ввода-вывода, то для консольного ввода-вывода используются именно функции *printf()* и *scanf()*. Обе они могут обрабатывать данные любых встроенных типов, включая символы, строки и числа. Но поскольку эти функции не являются объектно-ориентированными, их нельзя использовать непосредственно для ввода-вывода объектов классов, создаваемых программистом.

Функция printf()

Функция *printf()* имеет такой прототип:

```
int printf(const char *fmt_string, ...);
```

Первый аргумент, *fmt_string*, определяет способ отображения всех последующих аргументов. Этот аргумент часто называют *строкой форматирования*. Она состоит из элементов двух типов: текста и спецификаторов формата. К элементам первого типа относятся символы (текст), которые выводятся на экран. Элементы второго типа (спецификаторы формата) содержат команды форматирования, которые определяют способ отображения аргументов. Команда форматирования начинается с символа процента, за которым следует код формата. Спецификаторы формата перечислены в табл. A.1. Количество аргументов должно в точности совпадать с количеством команд форматирования, причем совпадение обязательно и в порядке их следования. Например, при вызове следующей функции *printf()*.

```
printf ("Привет %c %d %s", 'c', 10, "всем!");
```

На экране будет отображено: *"Привет с 10 всем! "*.

Функция *printf()* возвращает число реально выведенных символов. Отрицательное значение возврата свидетельствует об ошибке.

Команды формата могут иметь модификаторы, которые задают ширину поля, точность (количество десятичных разрядов) и признак выравнивания по левому краю. Целое значение, расположенное между знаком *%* и *командой форматирования*, выполняет роль *спецификатора минимальной ширины поля*. Наличие этого спецификатора приведет к тому, что результат будет заполнен пробелами или нулями, чтобы гарантированно обеспечить для выводимого значения заданную минимальную длину. Если выводимое значение (строка или число) больше этого минимума, оно будет выведено полностью, несмотря на превышение минимума. По умолчанию в качестве заполнителя используется пробел. Для

превышение минимума. По умолчанию в качестве заполнителя используется пробел. Для заполнения нулями нужно поместить *0* перед спецификатором ширины поля. Например, строка форматирования *%05d* дополнит выводимое число нулями (их будет меньше пяти), чтобы общая длина была равной пяти символам.

Таблица А.1. Спецификаторы формата функции `printf()`

Код	Формат
%c	Символ
%d	Десятичное целое со знаком
%i	Десятичное целое со знаком
%e	Экспоненциальное представление (строчная буква e)
%E	Экспоненциальное представление (прописная буква E)
%f	Значение с плавающей точкой
%g	Использует более короткий из двух форматов: %e или %f (если %e, использует строчную букву e)
%G	Использует более короткий из двух форматов: %E или %F (если %E, использует прописную букву E)
%o	Восьмеричное целое без знака
%s	Строка символов
%u	Десятичное целое без знака
%x	Шестнадцатеричное целое без знака (строчные буквы)
%X	Шестнадцатеричное целое без знака (прописные буквы)
%p	Указатель
%n	Соответствующий аргумент должен быть указателем на целое. Данный спецификатор сохраняет в этом целом число символов, выведенных в выходной поток к текущему моменту (до обнаружения спецификатора %n)
%%	Выводит символ %

Точное значение *модификатора точности* зависит от кода формата, к которому он применяется. Чтобы добавить модификатор точности, поставьте за спецификатором ширины поля десятичную точку, а после нее — значение спецификации точности. Для форматов *a*, *A*, *e*, *E*, *f* и *F* модификатор точности определяет число выводимых десятичных знаков. Например, строка форматирования *%10.4f* обеспечит вывод числа, ширина которого составит не меньше десяти символов, с четырьмя десятичными знаками. Применительно к целым или строкам, число, следующее за точкой, задает максимальную длину поля. Например, строка форматирования *%5.7s* отобразит строку длиной не менее пяти, но не более семи символов. Если выводимая строка окажется длиннее максимальной длины поля, конечные символы будут отсечены.

По умолчанию все выводимые значения выравниваются *по правому краю:* если ширина поля больше выводимого значения, оно будет выровнено по правому краю поля. Чтобы установить выравнивание по левому краю, поставьте знак *"минус"* сразу после знака *%*. Например, строка форматирования *%-10.2f* обеспечит выравнивание вещественного числа (с двумя десятичными знаками в 10-символьном поле) по левому краю. Рассмотрим программу, в которой демонстрируется использование спецификаторов ширины поля и выравнивания по левому краю.

```
#include <stdio.h>
```

```
int main( )
{
  printf("|%11.6f|\n", 123.23);
  printf ("|%-11.6f|\n", 123.23);
  printf("|%11.6s |\n", "Привет всем");
  printf("|%-11.6s |\n", "Привет всем");
  return 0;
}
```

При выполнении эта программа отображает такие результаты.

```
| 123.230000 |
| 123.230000 |
|      Привет|
| Привет      |
```

Существуют два модификатора команд форматирования, которые позволяют функции *printf()* отображать короткие (*short*) и длинные (*long*) целые. Эти модификаторы могут применяться к спецификаторам типа *d*, *i*, *o*, *u*, *x* и *X*. Модификатор *l* уведомляет функцию *printf()* о длинном формате значения. Например, строка *%ld* означает, что должно быть выведено длинное целое. Модификатор *h* указывает на применение короткого формата. Следовательно, строка *%hu* означает, что выводимое целочисленное значение имеет тип *short unsigned*.

Чтобы обозначить, что соответствующий аргумент указывает на длинное целое, к спецификатору *n* можно применить модификатор *l*. Для указания на короткое целое примените к спецификатору *n* модификатор *h*.

Если вы используете современный компилятор, который поддерживает добавленные в 1995 году средства работы с символами широкого формата (двухбайтовыми символами), то можете задействовать модификатор *l* применительно к спецификатору *c*, чтобы уведомить об использовании двухбайтовых символов. Кроме того, модификатор *l* можно использовать с командой формата *s* для вывода строки двухбайтовых символов.

Модификатор *l* можно также поставить перед командами форматирования вещественных чисел *e*, *E*, *f*, *F*, *g* и *G*. В этом случае он уведомит о выводе значения типа *long double*.

Функция scanf()

Функция *scanf()* — это С-функция общего назначения ввода данных с консольного устройства. Она может считывать данные всех встроенных типов и автоматически преобразует числа в соответствующий внутренний формат. Ее поведение во многом обратно поведению функции *printf()*. Общий формат функции *scanf()* таков.

```
int scanf (const char *fmt_string, ...);
```

Управляющая строка, задаваемая параметром *fmt_string*, состоит из символов трех категорий:

- спецификаторов формата;
- "пробельных" символов (пробелы, символы табуляции и пустой строки);
- символов, отличных от "пробельных".

Функция *scanf()* возвращает количество введенных полей, а при возникновении ошибки — значение *EOF* (оно определено в заголовке *stdio.h*).

Спецификаторы формата — им предшествует знак процента (*%*) — сообщают, какого типа данное будет считано следующим. Например, спецификатор *%s* прочитает строку, а *%d* — целое значение. Эти коды приведены в табл. А.2.

Таблица А.2. Спецификаторы формата функции `scanf()`

Код	Значение
`%c`	Считывает один символ
`%d`	Считывает десятичное целое
`%i`	Считывает целое в любом формате (десятичное, восьмеричное, шестнадцатеричное)
`%e`	Считывает вещественное число
`%f`	Считывает вещественное число
`%F`	Аналогично коду `%f` (только C99)
`%g`	Считывает вещественное число
`%o`	Считывает восьмеричное число
`%s`	Считывает строку
`%x`	Считывает шестнадцатеричное число

Окончание табл. А.2

Код	Значение
`%p`	Считывает указатель
`%n`	Принимает целое значение, равное количеству символов, считанных до сих пор
`%u`	Считывает десятичное целое без знака
`%[ ]`	Просматривает набор символов
`%%`	Считывает знак процента

Пробельные символы в строке форматирования заставляют функцию *scanf()* пропустить один или несколько пробельных символов во входном потоке. Под пробельным символом подразумевается пробел, символ табуляции или символ новой строки. По сути, один пробельный символ в управляющей строке заставит функцию *scanf()* считывать, но не сохранять любое количество (пусть даже нулевое) пробельных символов до первого непробельного.

"Непробельный" символ в строке форматирования заставляет функцию *scanf()* прочитать и отбросить соответствующий символ. Например, при использовании строки форматирования *%d*, *%d* функция *scanf()* сначала прочитает целое значение, затем прочитает и отбросит запятую и наконец прочитает еще одно целое. Если заданный символ не обнаружится, работа функции *scanf()* будет завершена.

Все переменные, используемые для приема значений с помощью функции *scanf()*, должны передаваться посредством их адресов. Это значит, что все аргументы должны быть указателями на переменные. (C не поддерживает ссылки или ссылочные параметры.) Передача указателей позволяет функции *scanf()* изменять содержимое любого аргумента. Например, если нужно считать целочисленное значение в переменную *count*, используйте следующий вызов функции *scanf()*.

```
scanf("%d", &count);
```

Строки обычно считываются в символьные массивы, а имя массива (без индекса) является адресом первого элемента в этом массиве. Поэтому, чтобы считать строку в символьный массив *address*, используйте такой код.

```
char address[80];
scanf("%s", address);
```

В этом случае параметр *address* уже является указателем, и поэтому его не нужно предварять оператором *"&"*.

Элементы входного потока, считываемые функцией *scanf()*, должны быть разделены пробелами, символами табуляции или новой строки. Такие символы, как запятая, точка с запятой и тому подобное, не распознаются в качестве разделителей. Это означает, что инструкция:

```
scanf("%d%d", &r, &c);
```

Примет значения, введенные как *10 20*, но наотрез откажется от "блюда", поданного в виде *10,20*.

Подобно *printf()*, в функции *scanf()* спецификаторы формата по порядку сопоставляются с переменными, перечисленными в списке аргументов.

Символ стоящий *"*"* после знака *"%"* и перед кодом формата, прочитает данные заданного типа, но запретит их присваивание переменной. Следовательно, инструкция

```
scanf("%d%*c%d", &x, &y);
```

при вводе данных в виде *10/20* поместит значение *10* в переменную *x*, отбросит знак деления и присвоит значение *20* переменной *y*.

Команды форматирования могут содержать модификатор максимальной длины поля. Он представляет собой целое число, располагаемое между знаком *"%"* и кодом формата, которое ограничивает количество символов, считываемых для любого поля. Например, если вы хотите прочитать в переменную *str* не более 20 символов, используйте следующую инструкцию.

```
scanf ("%20s", str);
```

Если входной поток содержит более 20 символов, то при последующем выполнении операции ввода считывание начнется с того места, в котором "остановился" предыдущий вызов функции *scanf()*. Например, если (при использовании данного примера) вводится такая строка символов

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ,
```

то в переменную *str* будут приняты только первые 20 символов (до буквы *'T'*), поскольку команда форматирования здесь содержит модификатор максимальной длины поля.

Это означает, что остальные символы, *"UVWXYZ"*, не будут использованы вообще. В случае другого вызова функции *scanf()*.

```
scanf("%s", str);
```

символы *"UVWXYZ"* поместились бы в переменной *str*. При обнаружении *"пробельного"* символа ввод данных для поля может завершиться до достижения максимальной длины поля. В этом случае функция *scanf()* переходит к считыванию следующего поля.

Несмотря на то что пробелы, символы табуляции и новой строки используются в качестве разделителей полей, при считывании одиночного символа они читаются подобно любому другому символу. Например, если входной поток состоит из символов *x y*, то инструкция

```
scanf ("%c%c%c", &a, &b, &c);
```

поместит символ *x* в переменную *a*, *пробел* — в переменную *b* и символ *y* — в переменную *c*.

Функцию *scanf()* можно также использовать в качестве набора сканируемых символов (*scanset*). В этом случае определяется набор символов, которые могут быть считаны функцией *scanf()* и присвоены соответствующему массиву символов. Функция *scanf()* продолжает считывать символы и помещать их в соответствующий символьный массив до тех пор, пока не встретится символ, отсутствующий в заданном наборе. После этого она переходит к следующему полю (если такое имеется).

Для определения такого набора необходимо заключить символы, подлежащие сканированию, в квадратные скобки. Открывающая квадратная скобка должна следовать сразу за знаком процента. Например, следующий набор сканируемых символов говорит о том, что необходимо прочитать только символы *X*, *Y* и *Z*.

```
%[XYZ]
```

Соответствующая набору переменная должна быть указателем на массив символов. При возврате из функции *scanf()* этот массив будет содержать строку с завершающим нулем, состоящую из считанных символов. Например, следующая программа использует набор сканируемых символов для считывания цифр в массив *s1*. Если будет введен символ, отличный от цифры, массив *s1* завершится нулевым символом, а остальные символы будут считываться в массив *s2* до тех пор, пока не будет введен следующий *"пробельный"* символ.

```
/* Простой пример использования набора сканируемых символов.
*/
#include <stdio.h>

int main( )
{
  char s1 [80], s2 [80];

  printf("Введите числа, а затем несколько букв:\n");
  scanf("%[0123456789]%s", s1, s2);
  printf("%s %s", s1, s2);

  return 0;
}
```

Многие компиляторы позволяют с помощью дефиса задать в наборе сканируемых символов диапазон. Например, при выполнении следующей инструкции функция *scanf()* будет принимать символы от *A* до *Z*.

```
% [A-Z]
```

При этом в наборе сканируемых символов можно задать даже несколько диапазонов. Например, эта программа считывает сначала цифры, а затем буквы.

```
/* Пример использования в наборе сканируемых символов нескольких
диапазонов.
*/
#include <stdio.h>

int main( )
{
  char s1[80], s2 [80];

  printf("Введите числа, а затем несколько букв:\n");
  scanf("%[0-9]%[a-zA-Z]", s1, s2);
  printf ("%s %s", s1, s2);

  return 0;
}
```

Если первый символ в наборе сканируемых символов является знаком вставки (^), то получаем обратный эффект: вводимые данные будут считываться до первого символа из заданного набора символов, т.е. знак вставки заставляет функцию *scanf()* принимать любые символы, которые *не определены* в наборе. В следующей модификации предыдущей программы знак вставки используется для запрещения считывания символов, тип которых указан в наборе сканируемых символов:

```
/* Пример использования набора сканируемых символов для запрещения
считывания указанных в нем символов
```

```
считывания указанных в нем символов.
*/
#include <stdio.h>

int main()
{
  char s1[80], s2[80];

  printf("Введите не цифры, а затем не буквы:\n");
  scanf("%[^0-9]%[^a-zA-Z]", s1, s2);
  printf ("%s %s", s1, s2);

  return 0;
}
```

Важно помнить, что набор сканируемых символов различает прописные и строчные буквы. Следовательно, если вы хотите сканировать как прописные, так и строчные буквы, задайте их отдельно.

Некоторые спецификаторы формата могут использовать такие модификаторы, которые точно указывают тип переменной, принимающей данные. Чтобы прочитать длинное целое, поставьте перед спецификатором формата модификатор *l*, а чтобы прочитать короткое целое — модификатор *h*. Эти модификаторы можно использовать с кодами формата *d*, *i*, *o*, *u*, *x* и *n*.

По умолчанию спецификаторы *f*, *e* и *g* заставляют функцию *scanf()* присваивать данные переменным типа *float*. Если поставить перед одним из этих спецификаторов формата модификатор *l*, функция *scanf()* присвоит прочитанное данное переменной типа *double*. Использование же модификатора *L* означает, что переменная, принимающая значение, имеет тип *long double*.

Модификатор *l* можно применить и к спецификатору *c*, чтобы обозначить указатель на двухбайтовый символ с типом данных *wchar_t* (если ваш компилятор соответствует стандарту C++). Модификатор *l* можно также использовать с кодом формата *s*, чтобы обозначить указатель на строку двухбайтовых символов. Кроме того, модификатор *l* можно использовать для модификации набора сканируемых двухбайтовых символов.

C-система обработки файлов

Несмотря на то что файловая система в C отличается от той, что используется в C++, между ними есть много общего. C-система обработки файлов состоит из нескольких взаимосвязанных функций. Наиболее популярные из них перечислены в табл. А.3.

Общий поток, который "цементирует" C-систему ввода-вывода, представляет собой *файловый указатель* (file pointer). Файловый указатель — это указатель на информацию о файле, которая включает его имя, статус и текущую позицию. По сути, файловый указатель идентифицирует конкретный дисковый файл и используется потоком, чтобы сообщить всем C-функциям ввода-вывода, где они должны выполнять операции. Файловый указатель — это переменная-указатель типа *FILE*, который определен в заголовке *stdio.h*.

Таблица А.3. C-функции обработки файлов

Функция	Назначение
`fopen()`	Открывает поток
`fclose()`	Закрывает поток
`fputc()`	Записывает символ в поток
`fgetc()`	Считывает символ из потока
`fwrite()`	Записывает блок данных в поток

fwrite()	Записывает блок данных в поток
fread()	Считывает блок данных из потока

Окончание табл. А.3

Функция	Назначение
fseek()	Устанавливает индикатор позиции файла на заданный байт в потоке
fprintf()	Делает для потока то, что функция printf() делает для консоли
fscanf()	Делает для потока то, что функция scanf() делает для консоли ()
feof()	Возвращает значение true, если достигнут конец файла
ferror()	Возвращает значение true, если возникла ошибка
rewind()	Устанавливает индикатор позиции файла в начало файла
remove()	Удаляет файл

Функция fopen()

Функция *fopen()* выполняет три задачи.

1. Открывает поток.

2. Связывает файл с потоком.

3. Возвращает указатель типа *FILE* на этот поток.

Чаще всего под файлом подразумевается дисковый файл. Функция *fopen()* имеет такой прототип.

```
FILE *fopen(const char *filename, const char *mode);
```

Здесь параметр *filename* указывает на имя открываемого файла, а параметр *mode*— на строку, содержащую нужный статус (режим) открытия файла. Возможные значения параметра *mode* показаны в приведенной табл. А.4. Параметр *filename* должен представлять строку символов, составляющих имя файла, которое допустимо в данной операционной системе. Эта строка может включать спецификацию пути, если действующая среда поддерживает такую возможность.

Таблица А.4. Допустимые значения параметра *mode*

mode	Назначение
"r"	Открывает текстовый файл для чтения
"w"	Создает текстовый файл для записи
"a"	Открывает текстовый файл для записи в конец файла
"rb"	Открывает двоичный файл для чтения
"wb"	Создает двоичный файл для записи
"ab"	Открывает двоичный файл для записи в конец файла
"r+"	Открывает текстовый файл для чтения и записи
"w+"	Создает текстовый файл для чтения и записи
"a+"	Открывает текстовый файл для чтения и записи в конец файла
"r+b"	Открывает двоичный файл для чтения и записи
"w+b"	Создает двоичный файл для чтения и записи
"a+b"	Открывает двоичный файл для чтения и записи в конец файла

Если функция *fopen()* успешно открыла заданный файл, она возвращает указатель *FILE*. Этот указатель идентифицирует файл и используется большинством других файловых системных функций. Он не должен подвергаться модификации кодом программы. Если файл не удается открыть, возвращается нулевой указатель.

Как видно из табл. А.4, файл можно открывать либо в текстовом, либо в двоичном режиме. При открытии в текстовом режиме выполняются преобразования некоторых последовательностей символов. Например, символы новой строки преобразуются в последовательности символов *"возврат каретки"/"перевод строки"*. В двоичном режиме подобные преобразования не выполняются.

Если вам нужно открыть файл *test* для записи, используйте следующую инструкцию.

```
fp = fopen("test", "w");
```

Здесь переменная *fp* имеет тип *FILE**. Однако зачастую для открытия файла используется такой код.

```
if((fp = fopen ("test", "w"))==NULL) {
  printf ("Не удается открыть файл.");
  exit(1);
}
```

При таком методе выявляется любая ошибка, связанная с открытием файла (например, при использовании защищенного от записи или заполненного диска), и только после этого можно предпринимать попытку записи в заданный файл. *NULL* — это макроимя, определенное в заголовке *stdio.h*.

Если вы используете функцию *fopen()*, чтобы открыть файл исключительно для выполнения операций вывода (записи), любой уже существующий файл с заданным именем будет стерт, и вместо него будет создан новый. Если файл с таким именем не существует, он будет создан. Если вам нужно добавлять данные в конец файла, используйте режим *"a"*. Если окажется, что указанный файл не существует, он будет создан. Чтобы открыть файл для выполнения операций чтения, необходимо наличие этого файла. В противном случае функция возвратит значение ошибки. Наконец, если файл открывается для выполнения операций чтения-записи, то в случае его существования он не будет удален; но если его нет, он будет создан.

Функция fputc()

Функция *fputc()* используется для вывода символов в поток, предварительно открытый для записи с помощью функции *fopen()*. Ее прототип имеет следующий вид.

```
int fputc(int ch, FILE *fp);
```

Здесь параметр *fp* — файловый указатель, возвращаемый функцией *fopen()*, а параметр *ch* — выводимый символ. Файловый указатель сообщает функции *fputc()*, в какой дисковый файл необходимо записать символ. Несмотря на то что параметр *ch* имеет тип *int*, в нем используется только младший байт.

При успешном выполнении операции вывода функция *fputc()* возвращает записанный в файл символ, в противном случае — значение *EOF*.

Функция fgetc()

Функция *fgetc()* используется для считывания символов из потока, открытого в режиме чтения с помощью функции *fopen()*. Ее прототип имеет следующий вид.

```
int fgetc(FILE *fp);
```

Здесь параметр *fp* — файловый указатель, возвращаемый функцией *fopen()*. Несмотря на то что функция *fgetc()* возвращает значение типа *int*, его старший байт равен нулю.

При возникновении ошибки или достижении конца файла функция *fgetc()* возвращает значение *EOF*. Следовательно, для того, чтобы считать все содержимое текстового файла (до самого конца), можно использовать следующий код.

```
ch = fgetc(fp);
while(ch!=EOF) {
  ch = fgetc(fp);
```

```
  ch = fgetc(fp);
}
```

Функция feof()

Файловая система в С может также обрабатывать двоичные данные. Если файл открыт в режиме ввода двоичных данных, то не исключено, что может быть считано целое число, равное значению *EOF*. В этом случае при использовании такого кода проверки достижения конца файла, как *ch != EOF*, будет создана ситуация, эквивалентная получению сигнала о достижении конца файла, хотя в действительности физический конец файла может быть еще не достигнут. Чтобы решить эту проблему, в языке С предусмотрена функция *feof()*, которая используется для определения факта достижения конца файла при считывании двоичных данных. Ее прототип имеет такой вид.

```
int feof(FILE *fp);
```

Здесь параметр *fp* идентифицирует файл. Функция *feof()* возвращает ненулевое значение, если конец файла был-таки достигнут; в противном случае — нуль. Таким образом, при выполнении следующей инструкции будет считано все содержимое двоичного файла.

```
while(!feof(fp)) ch = fgetc(fp);
```

Безусловно, этот метод применим и к текстовым файлам.

Функция fclose()

Функция *fclose()* закрывает поток, который был открыт в результате обращения к функции *fopen()*. Она записывает в файл все данные, еще оставшиеся в дисковом буфере, и закрывает файл на уровне операционной системы. При вызове функции *fclose()* освобождается блок управления файлом, связанный с потоком, что делает его доступным для повторного использования. Вероятно, вам известно о существовании ограничения операционной системы на количество файлов, которые можно держать открытыми в любой момент времени, поэтому, прежде чем открывать следующий файл, рекомендуется закрыть все файлы, уже ненужные для работы.

Функция *fclose()* имеет следующий прототип,

```
int fclose(FILE *fp);
```

Здесь параметр *fp* — файловый указатель, возвращаемый функцией *fopen()*. При успешном выполнении функция *fclose()* возвращает нуль; в противном случае возвращается значение *EOF*. Попытка закрыть уже закрытый файл расценивается как ошибка. При удалении носителя данных до закрытия файла будет сгенерирована ошибка, как и в случае недостатка свободного пространства на диске.

Использование функций fopen(), fgetc(), fputc() и fclose()

Функции *fopen()*, *fgetc()*, *fputc()* и *fclose()* составляют минимальный набор операций с файлами. Их использование демонстрируется в следующей программе, которая выполняет копирование файла. Обратите внимание на то, что файлы открываются в двоичном режиме и что для проверки достижения конца файла используется функция *feof()*.

```
/* Эта программа копирует содержимое одного файла в другой.
*/
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("Вы забыли указать имя файла.\n");
```

```
    printf("Вы забыли ввести имя файла.\n");
    return 1;
  }
  if((in=fopen(argv[1], "rb")) == NULL) {
    printf("Не удается открыть исходный файл.\n");
    return 1;
  }
  if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Не удается открыть файл-приемник.\n");
    return 1;
  }

  /* Код копирования содержимого файла. */
  while(!feof (in)) {
    ch = fgetc(in);
    if(!feof(in)) fputc (ch, out);
  }

  fclose(in);
  fclose(out);

  return 0;
}
```

Функции ferror() и rewind()

Функция *ferror()* используется для определения факта возникновения ошибки при выполнении операции с файлом. Ее прототип имеет такой вид.

```
int ferror(FILE *fp);
```

Здесь параметр *fp* — действительный файловый указатель. Функция *ferror()* возвращает значение *true*, если при выполнении последней файловой операции произошла ошибка; в противном случае — значение *false*. Поскольку возникновение ошибки возможно при выполнении любой операции с файлом, функцию *ferror()* необходимо вызывать сразу после каждой функции обработки файлов; в противном случае информацию об ошибке можно попросту потерять.

Функция *rewind()* перемещает индикатор позиции файла в начало файла, заданного в качестве аргумента. Ее прототип выглядит так.

```
void rewind(FILE *fp);
```

Здесь параметр *fp* — действительный файловый указатель.

Функции fread() и fwrite()

В файловой системе языка С предусмотрено две функции, *fread()* и *fwrite()*, которые позволяют считывать и записывать блоки данных. Эти функции подобны C++-функциям *read()* и *write()*. Их прототипы имеют следующий вид.

```
size_t fread(void *buffer, size_t num_bytes, size_t count, FILE *fp);

size_t fwrite(const void *buffer, size_t num_bytes, size_t count, FILE
*fp);
```

При вызове функции *fread()* параметр *buffer* представляет собой указатель на область

памяти, которая предназначена для приема данных, считываемых из файла. Функция считывает *count* объектов длиной *num_bytes* из потока, адресуемого файловым указателем *fp*. Функция *fread()* возвращает количество считанных объектов, которое может оказаться меньше заданного значения *count*, если при выполнении этой операции возникла ошибка или был достигнут конец файла.

При вызове функции *fwrite()* параметр *buffer* представляет собой указатель на информацию, которая подлежит записи в файл. Эта функция записывает *count* объектов длиной *num_bytes* в поток, адресуемый файловым указателем *fp*. Функция *fwrite()* возвращает количество записанных объектов, которое будет равно значению *count*, если при выполнении этой операции не было ошибки.

Если при вызове функций *fread()* и *fwrite()* файл был открыт для выполнения двоичной операции, то они могут считывать или записывать данные любого типа. Например, следующая программа записывает в дисковый файл значение типа *float*.

```
/* Запись в дисковый файл значения с плавающей точкой.
*/
#include <stdio.h>
int main( )
{
  FILE *fp;
  float f = 12.23F;
  if((fp=fopen("test", "wb"))==NULL) {
    printf("Не удается открыть файл.\n");
    return 1;
  }
  fwrite(&f, sizeof(float), 1, fp);
  fclose(fp);
  return 0;
}
```

Как показано в этой программе, роль буфера может выполнять (и при том довольно часто) одна переменная.

С помощью функций *fread()* и *fwrite()* часто выполняется считывание и запись содержимого массивов или структур. Например, следующая программа, используя одну только функцию *fwrite()*, записывает содержимое массива значений с плавающей точкой *balance* в файл с именем *balance*. Затем с помощью одной лишь функции *fread()* программа считывает элементы этого массива и отображает их на экране.

```
#include <stdio.h>

int main( )
{
  register int i;
  FILE *fp;
  float balance[100];

  /* Открываем файл для записи. */
  if((fp=fopen("balance", "w"))==NULL) {
    printf("Не удается открыть файл.\n");
    return 1;
```

```
    return 1;
  }

  for(i=0; i<100; i++) balance[i] = (float) i;

  /* Одним "махом" сохраняем весь массив balance. */
  fwrite(balance, sizeof balance, 1, fp);
  fclose(fp);

  /* Обнуляем массив. */
  for(i=0; i<100; i++) balance[i] = 0.0;

  /* Открываем файл для чтения. */
  if((fp=fopen("balance", "r"))==NULL) {
    printf("Не удается открыть файл.\n");
    return 1;
  }

  /* Одним "махом" считываем весь массив balance. */
  fread(balance, sizeof balance, 1, fp);

  /* Отображаем содержимое массива. */
  for(i=0; i<100; i++) printf("%f ", balance[i]);
  fclose(fp);

  return 0;
}
```

Использовать функции *fread()* и *fwrite()* для считывания и записи блоков данных более эффективно, чем многократно вызывать функции *fgetc()* и *fputc()*.

Функция fseek() и выполнение ввода-вывода с произвольным доступом

С-система ввода-вывода позволяет выполнять операции считывания и записи данных с произвольным доступом. Для этого служит функция *fseek()*, которая устанавливает нужным образом индикатор позиции файла. Ее прототип таков.

```
int fseek(FILE *fp, long numbytes, int origin);
```

Здесь параметр *fp* — файловый указатель, возвращаемый функцией *fopen()*, параметр *numbytes* — количество байтов относительно исходного положения, заданного параметром *origin*. Параметр *origin* может принимать одно из следующих макроимен (определенных в заголовке *stdio.h*).

Имя	Назначение
SEEK_SET	Поиск с начала файла
SEEK_CUR	Поиск с текущей позиции
SEEK_END	Поиск с конца файла

Следовательно, чтобы переместить индикатор позиции в файле на *numbytes* байтов относительно его начала, в качестве параметра *origin* необходимо использовать значение *SEEK_SET*. Для перемещения относительно текущей позиции используйте значение *SEEK_CUR*, а для смещения с конца файла — значение *SEEK_END*.

Нулевое значение результата функции свидетельствует об успешном выполнении функции

fseek(), а ненулевое— о возникновении сбоя. Как правило, функцию *fseek()* не рекомендуется использовать для файлов, открытых в текстовом режиме, поскольку преобразование символов может привести к ошибочным перемещениям индикатора позиции в файле. Поэтому лучше использовать эту функцию для файлов, открытых в двоичном режиме. Например, если вам нужно считать 234-й байт в файле *test*, выполните следующий код.

```
int func1( )
{
  FILE *fp;

  if((fp=fopen("test", "rb")) == NULL) {
    printf("Не удается открыть файл.\n");
    exit (1);
  }

  fseek(fp, 234L, SEEK_SET);
  return getc(fp); /* Считывание одного символа, расположенного на 234-й позиции. */
}
```

Функции fprintf() и fscanf()

Помимо рассмотренных выше основных функций ввода-вывода, С-система ввода-вывода включает функции *fprintf()* и *fscanf()*. Поведение этих функций аналогично поведению функций *printf()* и *scanf()*, за исключением того, что они работают с файлами. Именно поэтому эти функции обычно используются в С-программах. Прототипы функций *fprintf()* и *fscanf()* выглядят так.

```
int fprintf(FILE * fp, const char *fmt_string, ...);

int fscanf(FILE * fp, const char *fmt_string, ...);
```

Здесь параметр *fp* — файловый указатель, возвращаемый функцией *fopen()*. Функции *fprintf()* и *fscanf()* работают подобно функциям *printf()* и *scanf()* соответственно, за исключением того, что их действие направлено на файл, определенный параметром *fp*.

Удаление файлов

Функция *remove()* удаляет заданный файл. Ее прототип выглядит так.

```
int remove(const char *filename);
```

Она возвращает нуль при успешном удалении файла и ненулевое значение в противном случае.

Приложение Б: Использование устаревшего C++-компилятора

Программы, приведенные в этой книге, полностью соответствуют стандарту ANSI/ISO для C++ и могут быть скомпилированы практически любым современным C++-компилятором, включая Visual C++ (Microsoft) и C++ Builder (Borland). Следовательно, при использовании современного компилятора у вас не должно быть проблем с компиляцией программ из этой книги. (В этом случае вам вообще не понадобится информация, представленная в этом приложении.)

Но если вы используете компилятор, созданный несколько лет назад, то при попытке скомпилировать наши примеры он может выдать целый список ошибок, не распознав ряд новых C++-средств. И в этом случае не стоит волноваться. Для того чтобы эти программы заработали со старыми компиляторами, нужно внести в них небольшие изменения. Чаще всего старые и новые C++-программы отличаются использованием двух средств: заголовков и пространств имен. Вот об этом и пойдет речь в этом приложении.

Как упоминалось в главе 2, инструкция *#include* включает в программу заданный заголовок. Для более ранних версий C++ под заголовками понимались файлы с расширением *.h*. Например, в старой C++-программе для включения заголовка *iostream* была бы использована следующая инструкция.

```
#include <iostream.h>
```

В этом случае в программу был бы включен заголовочный файл *iostream.h*. Таким образом, включая в старую C++-программу заголовок, необходимо задавать имя файла с расширением *.h*.

В новых C++-программах в соответствии со стандартом ANSI/ISO для C++ используются заголовки другого типа. Современные заголовки определяют не имена файлов, а стандартные идентификаторы, которые могут совпадать с таковыми, но не всегда. Современные C++-заголовки представляют собой абстракцию, которая попросту гарантирует включение в программу требуемой информации.

Поскольку современные заголовки необязательно являются именами файлов, они не должны иметь расширение *.h*. Они определяют имя заголовка, заключенного в угловые скобки. Вот, например, как выглядят два современных заголовка, подлерживаемых стандартом C++.

```
<iostream>

<fstream>
```

Чтобы преобразовать эти *"новые"* заголовки в *"старые"* заголовочные файлы, достаточно добавить расширение *.h*.

Включая современный заголовок в программу, необходимо помнить, что его содержимое относится к пространству имен *std*. Как упоминалось выше, пространство имен — это просто декларативная область. Ее назначение — локализовать имена идентификаторов во избежание коллизий с именами. Старые версии C++ помещают имена библиотечных функций в глобальное пространство имен, а не в пространство имен *std*, используемое современными компиляторами. Таким образом, работая со старым компилятором, не нужно использовать эту инструкцию:

```
using namespace std;
```

В действительности большинство старых компиляторов вообще не воспримут инструкцию *using namespace*.

Два простых изменения

Если ваш компилятор не поддерживает пространства имен и новые заголовки, он выдаст одно или несколько сообщений об ошибках при попытке скомпилировать первые несколько строк программ, приведенных в этой книге. В этом случае в эти программы необходимо внести только два простых изменения: использовать заголовок старого типа и удалить *namespac*e-инструкцию. Например, замените эти инструкции

```
#include <iostream>

using namespace std;
```

такой.

```
#include <iostream.h>
```

Это изменение преобразует *"новую"* программу в *"старую"*. Поскольку при использовании *"старого"* заголовка в глобальное пространство имен считывается все содержимое заголовочного файла, необходимость в использовании *namespaee*-инструкции отпадает. После внесения этих изменений, программу можно скомпилировать с помощью старого компилятора.

Иногда приходится вносить и другие изменения. C++ наследует ряд заголовков из языка C. Язык C не поддерживает современный стиль использования C++-заголовков, используя вместо них заголовочные *.h*-файлы. Для разрешения обратной совместимости стандарт C++

вместо них заголовочные *.h*-файлы. Для разрешения обратной совместимости стандарт C++ по-прежнему поддерживает заголовочные C-файлы. Однако стандарт C++ также определяет современные заголовки, которые можно использовать вместо заголовочных C-файлов. В C++-версиях стандартных C-заголовков к имени C-файла просто добавляется префикс *'c'* и опускается расширение *.h*. Например, C++-заголовком для файла *math.h* служит заголовок *<cmath>*, а для файла *string.h*— заголовок *<cstring>*. Несмотря на то что в C++-программу разрешено включать заголовочный C-файл, против такого подхода у разработчиков стандарта есть существенные возражения (другими словами, это не рекомендовано). Поэтому в настоящей книге используются современные C++-заголовки во всех инструкциях #include. Если ваш компилятор не поддерживает C++-заголовки для C-заголовков, просто замените "старые" заголовочные файлы.

Приложение B: *.NET-расширения для C++

Разработанная компанией Microsoft интегрированная оболочка *.NET Framework* определяет среду, которая предназначена для поддержки разработки и выполнения сильно распределенных приложений, основанных на использовании компонентных объектов. Она позволяет "мирно сосуществовать" различным языкам программирования и обеспечивает безопасность, переносимость программ и общую модель программирования для платформы Windows. Несмотря на относительную новизну оболочки .NET Framework, по всей вероятности, в ближайшем будущем в этой среде будут работать многие C++-программисты.

Интегрированная оболочка *.NET Framework* предоставляет управляемую среду, которая следит за выполнением программы. Программа, предназначенная для помещения в оболочку *.NET Framework*, не компилируется с целью получения объектного кода. Вместо этого она переводится на промежуточный язык *MSIL* (Microsoft Intermediate Language), а затем выполняется под управлением универсального средства *CLR* (Common Language Runtime). Управляемое выполнение— это механизм, который поддерживает ключевые преимущества, предлагаемые оболочкой *.NET Framework*.

Чтобы воспользоваться преимуществами управляемого выполнения, необходимо применять для C++-программ специальный набор нестандартных ключевых слов и директив препроцессора, которые были определены разработчиками компании Microsoft. Важно понимать, что этот дополнительный набор не включен в стандарт C++ (ANSI/ISO Standard C++). Поэтому код, в котором используются эти ключевые слова, нельзя переносить в другие среды выполнения.

Описание оболочки *.NET Framework* и методов C++-программирования, необходимых для ее использования, выходит за рамки этой книги. Однако здесь приводится краткий обзор *.NET*-расширения языка C++ ради тех программистов, которые работают в *.NET*-среде.

Ключевые слова .NET-среды

Для поддержки *.NET*-среды управляемого выполнения C++-программ Microsoft вводит в язык C++ следующие ключевые слова.

__abstract	__box	__delegate
__event	__finally	__gc
__identifier	__interface	__nogc
__pin	__property	__sealed
__try_cast	__typeof	__value

Краткое описание каждого из этих ключевых слов приведено в следующих разделах.

__abstract

Ключевое слово __*abstract* используется в сочетании со словом __*gc* при определении абстрактного управляемого класса. Объект __*abstract*-класса создать нельзя. Для класса, определенного с использованием ключевого слова __*abstract*, необязательно включение в него чисто виртуальной функции.

.

__box

Ключевое слов __*box* заключает в специальную оболочку значение внутри объекта. Такая "упаковка" позволяет использовать тип этого значения в коде, который требует, чтобы данный объект был выведен из класса *System::Object*, базового класса для всех *.NET*-объектов.

__delegate

Ключевое слово __*delegate* определяет объект-делегат, который инкапсулирует указатель на функцию внутри управляемого класса (т.е. класса, модифицированного ключевым словом __*gc*).

__event

Ключевое слово __*event* определяет функцию, которая представляет некоторое событие. Для такой функции задается только прототип.

__finally

Ключевое слово __*finally* — это дополнение к стандартному C++-механизму обработки исключительных ситуаций. Оно используется для определения блока кода, который должен выполняться после выхода из блоков *try/catch*. При этом не имеет значения, какие условия приводят к завершению *try/catch*-блока. Блок __*finally* должен быть выполнен в любом случае.

__gc

Ключевое слово __*gc* определяет управляемый класс. Обозначение *"gc"* представляет собой сокращение от словосочетания *"garbage collection"* (т.е. "сборка мусора") и означает, что объекты этого класса автоматически подвергаются процессу утилизации памяти, освобождаемой во время работы программы, когда они больше не нужны. В объекте отпадает необходимость в случае, когда на него не существует ни одной ссылки. Объекты __*gc*-класса должны создаваться с помощью оператора *new*. Массивы, указатели и интерфейсы также можно определять с использованием ключевого слова __*gc*.

__identifier

Ключевое слово __*identifier* позволяет любому другому ключевому слову языка C++ использоваться в качестве идентификатора. Эта возможность не предназначена для широкого применения и введена для решения специальных задач.

__interface

Ключевое слово __*interface* определяет класс, который должен действовать как интерфейс. В любом интерфейсе ни одна из функций не должна включать тело, т.е. все функции интерфейса являются неявно заданными чисто виртуальными функциями. Таким образом, интерфейс представляет собой абстрактный класс, в котором не реализована ни одна из его функций.

__nogc

Ключевое слово __*nogc* определяет неуправляемый класс. Поскольку такой (неуправляемый) тип класса создается по умолчанию, ключевое слово __*nogc* используется редко.

__pin

Ключевое слово __*pin* используется для определения указателя, который фиксирует местоположение в памяти объекта, на который он указывает. Таким образом, "закрепленный" объект не будет перемещаться в памяти в процессе сборки мусора. Как следствие, сборщик мусора не в состоянии сделать недействительным указатель, модифицированный с помощью ключевого слова __*pin*.

__property

Ключевое слово __*property* определяет свойство, являющееся функцией-членом, которая позволяет установить или получить значение некоторой переменной (члена данных класса). Свойства предоставляют удобное средство управления доступом к закрытым (private) или защищенным (protected) данным

защищенным (protected) данным.

__sealed

Ключевое слово __*sealed* предохраняет модифицируемый им класс от наследования другими классами. Это ключевое слово можно также использовать для информирования о том, что виртуальная функция не может быть переопределена.

__try_cast

С помощью ключевого слова __*try_cast* можно попытаться преобразовать тип выражения. Если предпринятая попытка окажется неудачной, будет сгенерировано исключение типа *System::InvalidCastException*.

__typeof

Ключевое слово __*typeof* позволяет получить объект, который инкапсулирует информацию о данном типе. Этот объект представляет собой экземпляр класса *System::Type*.

__value

Ключевое слово __*value* определяет класс, который представляет собой обозначение типа. Любое обозначение типа содержит собственные значения. И этим тип __*value* отличается от типа __*gc*, который должен выделять память для объекта с помощью оператора *new*. Обозначения типа, не представляют интерес для "сборщика мусора".

Расширения препроцессора

Для поддержки *.NET*-среды компания Microsoft определяет директиву препроцессора *#using*, которая используется для импортирования метаданных в программу. Метаданные содержат информацию о типе и членах класса в форме, которая не зависит от конкретного языка программирования. Таким образом, метаданные обеспечивают поддержку смешанного использования языков программирования. Все управляемые C++-программы должны импортировать библиотеку *<mscorlib.dll>*, которая содержит необходимые метаданные для оболочки *.NET Framework*.

Компания Microsoft определяет две *pragma*-инструкции (используемые с директивой препроцессора #pragma), которые имеют отношение к оболочке *.NET Framework*. Первая (*managed*) определяет управляемый код. Вторая (*unmanaged*) определяет неуправляемый (собственный, т.е. присущий данной среде) код. Эти инструкции могут быть использованы внутри программы для селективного создания управляемого и неуправляемого кода.

Атрибут attribute

Компания Microsoft определяет атрибут *attribute*, который используется для объявления другого атрибута.

Компиляция управляемых C++-программ

На момент написания этой книги единственный доступный компилятор, который мог обрабатывать программы, ориентированные на работу в среде *.NET Framework*, поставлялся компанией Microsoft (Visual Studio .NET). Чтобы скомпилировать управляемую программу, необходимо использовать команду */clr*, которая передаст вашу программу "в руки" универсального средства *Common Language Runtime*.

Предметный указатель

-Символы-

#define, директива, 570

#elif, директива, 576

#endif, директива, 575

#error, директива, 574

#if, директива, 575

#ifdef, директива, 577

#ifndef, директива, 577
#include, директива, 574; 602
#pragma, директива, 580
#undef, директива, 578
#using, 609
.NET Framework, 606
__abstract, 606
__box, 607
__cplusplus, макрос, 582
__DATE__, макрос, 582
__delegate, 607
__event, 607
__FILE__, макрос, 580; 582
__finally, 607
__gc, 607
__identifier, 607
__interface, 608
__LINE__, макрос, 580; 582
__nogc, 608

__pin, 608
__property, 608
__sealed, 608
__STDC__, макрос, 582
__TIME__, макрос, 582
__try_cast, 608
__typeof, 609
__value, 609
-A-
abort(), 417,419
abs(), 167; 191
Allocator, 524
American National Standards Institute, 18
ANSI, 18
asm, 514
assign(), 563
atof(), 164
attribute, 609
auto, спецификатор, 149; 206
-B-
bad(), 471

bad(), 474
bad_cast, исключение, 484
BASIC, 24
basic ios, класс, 440
basic_iostream, класс, 440
basic_istream, класс, 440
basic ostream, класс, 440
basic_streambuf, класс, 440
BCPL, 23
before(), 475
begin(), 529
Binding
  early, 393
  late, 393
bool, 56
boolalpha, флаг, 448
break, 95
-C-
C#, 29
C++ Builder, 27; 33
C89, 23
C99, 23
Call-by-reference, 178
Call-by-value, 178
Cast, 75
catch, 416
cerr, 440
char, 56; 61
cin, 440
class, 266
clear(), 471
clock(), 213
clog, 440
close(), 458
CLR, 606
Common Language Runtime, 606; 609
compare(), 566
const, спецификатор типа, 202; 508
const_cast, оператор, 488
continue, 94
count(), алгоритм, 554
count_if(), алгоритм, 554

cout, 440
-D-
Daylight Saving Time, 251
dec, флаг, 448
delete, 230
double, 40; 56
do-while, 93
dynamic_cast, оператор, 483
-E-
Early binding, 393
end(), 529
enum, 214
eof(), 463
erase(), 529
exit(), 418; 419
EXIT_FAILURE, константа, 419
EXIT_SUCCESS, константа, 419
explicit, 510
extern, 206; 516
-F-
fabs(), 191
fail(), 471
false, константа, 57
fclose(), 595
feof(), 595
ferror(), 597
fgetc(), 595
fill(), 451
find(), 565
fixed, флаг, 448
flags(), 449
Flat model, 141
float, 56
flush(), 467
fmtflags, перечисление, 447
fopen(), 593
for, цикл, 49; 82
FORTRAN, 24
fprintf(), 600
fputc(), 594
fread(), 597

fread(), 597
free(), 233
friend, 294

fscanf(), 600
fseek(), 599
Function overloading, 190
fwrite(), 597
-G-
gcount(), 463
Generated function, 398
get(), 460; 465
getline(), 466
gets(), 107
good(), 471
goto, 97
GUI, 18; 34
-H-
Heap, 229
hex, флаг, 448
-I-
IDE (Integrated Development Environment), 33
if, 48; 78
if-else-if, 81
Inline function, 283
inline, модификатор, 284; 574
insert(), 529; 537
Instantiating, 398
int, 38; 56; 61
Integral promotion, 74
Integrated Development Environment, 33
internal, флаг, 448
International Standards Organization, 18
ios, класс, 447
ios_base, класс, 440
iostate, перечисление, 470
isalpha(), 114
ISO, 18
-J-
Java, 29
-K-

kbhit(), 139
-L-
labs(), 191
Late binding, 393
left, флаг, 448
list, класс, 536
long double, 61; 62
long int, 61; 62
long, модификатор, 60
-M-
main(), 162
make_pair(), 546
malloc(), 233
managed, 609
Manipulator, 447
map, класс, 545
merge(), 537
MFC, 387
Microsoft Foundation Classes, 387
Microsoft Intermediate Language, 606
Modula-2, 23; 41
MSIL, 606
Multiple indirection, 141
mutable, 509
-N-
name(), 475
namespace, 494
Namespace, 35
new, оператор, 230; 430
nothrow, 431
npos, 561
NULL, 594
-O-
Object Oriented Programming, 264
oct, флаг, 448
OOP, 264
open(), 456
openmode, перечисление, 457
operator, 320
Operator, 68
overload, 193

-P-
pair, класс, 546
Pascal, 23; 41
peek(), 467
Plain Old Data, 281
POD-struct, 281
Pointer-to-member, 517
precision(), 451
Predicate, 524
Preprocessor, 570
printf(), 585
private, 281; 355
protected, 357
public, 267; 355
push_back(), 529; 537
put(), 460
putback(), 467
-Q-
qsort(), 503
Quicksort, алгоритм, 503
-R-
rand(), 138; 478

rdstate(), 470
read(), 461
Reference parameter, 181
register, спецификатор, 211
reinterpret_cast, оператор, 490
remove(), 600
return, инструкция, 166
rewind(), 597
rfind(), 565
right, флаг, 448
RTTI, 474
-S-
scanf(), 588
Scanset, 590
scientific, флаг, 448
seekg(), 468; 470
seekp(), 468; 470
setf(), 448

setf(), 448
short int, 61
short, модификатор, 60
showbase, флаг, 448
showflags(), 450
showpoint, флаг, 448
showpos, флаг, 448
signed char, 61
signed int, 61
signed long int, 61; 62
signed short int, 61
signed, модификатор, 60
Simula67, 26
sizeof, 227; 263
skipws, флаг, 448
splice(), 537
Standard C++, 27
Standard Template Library, 26; 54; 522
static, модификатор, 208; 210; 506
static_cast, оператор, 489
std, пространство имен, 35; 438
stderr, поток, 585
stdin, поток, 585
stdout, поток, 585
STL, 26; 54; 522
strcat(), 109
strcmp(), 110
strcpy(), 109; 171
Stream, 439
streamsize, тип, 451
string, класс, 559
strlen(), 111; 161
struct, 238
switch, 87
-T-
tellg(), 470
tellp(), 470
template, 396; 405
template<>, 401; 413
terminate(), 417
this,317; 508
throw, 416

throw, 415
throw-выражение, 427
time_t, тип даты, 251
tm, структура, 251
tolower(), 113
toupper(), 135
true, константа, 57
try, 416
Type promotion, 74
type_info, класс, 474
typeid, 474
typename, 396
-U-
unexpected(), 427
union, 258
unitbuf, флаг, 448
unmanaged, 609
unsetf(), 448; 449
unsigned char, 61
unsigned int, 61
unsigned long int, 61; 62
unsigned short int, 61
unsigned, модификатор, 60
uppercase, флаг, 448
using, 35; 497
virtual, 375; 381
Visual Basic, 23
Visual C++, 27; 33
void, 43; 47; 56
void-функции, 169
volatile, спецификатор типа, 204
-W-
wcerr, 440
wchar_t, 56
wcin, 440
wclog, 440
wcout, 440
while, 91
width(), 451; 452

write(), 461

-A-
Абстрактный класс, 393
Алгоритм
adjacent_find(), 551
binary_search(), 551
copy(), 551
copy_backward(), 551
count(), 551
count_if(), 551
equal(), 551
equal_range(), 551
fill(), 551
fill_n(), 551
find(), 551; 552
find_end(), 551
find_first_of(), 552
for_each(), 552
generate(), 552
generate_n(), 552
includes(), 552
inplace_merge(), 552
iter_swap(), 552
lexicographical_compare(), 552
lower_bound(), 552
make_heap(), 552
max(), 552
max_element(), 552
merge(), 552
min(), 552
min_element(), 552
mismatch(), 552
next_permutation(), 552
nth_element(), 552
partial_sort(), 552
partial_sort_copy(), 552
partition(), 552
pop_heap(), 553
prev_permutation(), 553
push_heap(), 553
Quicksort, 105
random_shuffle(), 553

remove(), 553
remove_copy(), 553; 555
remove_copy_if(), 553
remove_if(), 553
replace(), 553
replace_copy(), 553; 555
replace_copy_if(), 553
replace_if(), 553
reverse(), 553; 557
rotate(), 553
search(), 553
search_n(), 553
set_difference(), 553
set_intersection(), 553
set_symmetric_difference(), 553
set_union(), 553
sort(), 553
sort_heap(), 553
stable_partition(), 553
stable_sort(), 553
swap(), 553
swap_ranges(), 553
transform(), 553; 557
unique(), 553
upper_bond(), 553
Алгоритмы, 523; 551
командной строки, 162
по умолчанию, 193
функции main(),45; 162
Ассемблер, 23; 514
Атрибут
attribute, 609
-Б-
Библиотека
<mscorlib.dll>, 609
STL, 522
Битовое множество, 525
Битовые поля, 256
Блок кода, 24; 51; 148
-В-
Вектор, 527

Виртуальное наследование, 375
Виртуальные функции, 381
Выражение, 73
  условное, 79
-Г-
Глобальные переменные, 59
-Д-
Дек, 525
Декремент, 69
Деструктор, 273
Динамическая идентификация типов, 474
Динамическая инициализация, 300

Динамический массив, 526
Директива препроцессора, 570
  #define, 570
  #elif, 576
  #endif, 575
  #error, 574
  #if, 575
  #ifdef, 577
  #ifhdef, 577
  #include, 574; 602
  #line, 580
  #pragma, 580
  #undef, 578
  #using, 609
Дополнительный код, 62
-З-
Заголовки, 172
Заголовок
  <algorithm>, 551
  <bitset>, 525
  <cctype>, 113
  <cstdio>, 584
  <cstdlib>, 44; 419; 504
  <cstring>, 109
  <ctime>, 213; 251; 298
  <deque>, 525
  <fstream>, 456
  <functional>, 525

<iomanip>, 453
<iostream>, 35; 438; 440; 466
<list>, 525
<map>, 525
<new>, 430; 436
<queue>, 526
<set>, 526
<stack>, 526
<string>, 559
<typeinfo>, 474
<utility>, 525
<vector>, 526
stdio.h, 588
Заголовочный файл
<iostream.h>, 438
stdio.h, 584
-И-
Идентификатор, 53
Индекс, 102
Инициализация
динамическая, 300
массивов, 115
переменных, 66
Инкапсуляция, 27
Инкремент, 69
Инструкция
continue, 94
do-while, 93
for, 49
goto, 97
if, 48; 78
return, 45; 166
switch, 87
while, 91
Исключение, 230; 416
bad_alloc, 430
bad_cast, 484
bad_typeid, 477
System::InvalidCastException, 608
Исключительная ситуация, 416
Итераторы, 523

входные, 523
выходные, 523
двунаправленные, 523
однонаправленные, 523
произвольного доступа, 523
реверсивные, 524
-К-
Класс, 266
allocator, 524
basic_ios, 440
basic_iostream, 440
basic_istream, 440
basic_ostream, 440
basic_streambuf, 440
fstream, 456
ifstream, 456
ios, 447; 457
ios_base, 440
list, 536
map, 545
ofstream, 456
pair, 546
string, 559
type_info, 474
vector, 527
абстрактный, 393
базовый, 352
полиморфный, 381; 475
производный, 352
шаблонный
pair, 525
Классы
контейнерные, 525
bitset, 525
deque, 525
list, 525
map, 525
multimap, 526
multiset, 526
priority_queue, 526

priority_queue, 526
queue, 526
set, 526
stack, 526
vector, 526
обобщенные, 404
Ключевые слова C++, 53
Комментарий, 34
Компилятор
C++ Builder, 33
Visual C++, 33
Константа, 63
CLOCKS_PER_SEC, 298
EXIT_FAILURE, 419
EXIT_SUCCESS, 419
npos, 561
Конструктор, 272; 511
копии, 305; 311
параметризованный, 275
Контейнерные классы, 525
Контейнеры, 523
ассоциативные, 523; 545
векторы, 526
последовательные, 523
Куча, 229; 552; 553
Кэш, 212
-Л-
Лексема, 132
Линейный список, 525
Литерал, 63
восьмеричный, 64
строковый, 65; 106
шестнадцатеричный, 64
Локальные переменные, 57
-М-
Макроимя, 570; 582
Макроподстановка, 570
Макрос
__cplusplus, 582
__DATE__, 582
__FILE__, 582
__LINE__, 582

__LINE__, 582
__STDC__, 582
__TIME__, 582
SEEK_CUR, 599
SEEK_END, 599
SEEK_SET, 599
Манипулятор, 452
boolalpha, 452
dec, 452
endl, 452
ends, 452
fixed, 452
flush, 452
hex, 452
internal, 452
left, 452
noboolalpha, 452
noshowbase, 452
noshowpoint, 452
noshowpos, 452
noskipws, 452
nounitbuf, 453
nouppercase, 453
oct, 453
resetiosflags(), 453
right, 453
scientific, 453
setbase(), 453
setfill(), 453
setiosflags(), 453; 454
setprecision(), 453
setw(), 453
showbase, 453
showpoint, 453
showpos, 453
skipws, 453
unitbuf, 453
uppercase, 453
ws, 453; 454
Манипуляторные функции, 454
Массив, 102; 131
двумерный, 114

двумерный, 114
инициализация, 115
многомерный, 115
одномерный, 102
объектов, 286
строк, 119

указателей, 137
Метаданные, 609
Метка, 98
Многоуровневая непрямая адресация, 141
Множество, 526
битовое, 525
Модели памяти, 140
Модификатор
const, 488; 508
inline, 284
long, 60
mutable, 509
short, 60
signed, 60
static, 208; 210
unsigned, 60
volatile, 488
максимальной длины поля, 590
точности, 587
Модификаторы типов, 60
Мультиотображение, 526
-Н-
Набор сканируемых символов, 590
Наследование, 29; 351
виртуальное, 375
-О-
Обобщенные
классы, 404
функции, 396
Объединения, 258
анонимные, 262
Объект, 28
Объект-функция, 525
less, 525

Объявление
доступа, 370
класса, 360
опережающее, 297
переменных, 57
ООП, 25; 264
Оператор
!=, 475
&, 125
*, 125
==, 475
const_cast, 488
defined, 579
delete, 230
dynamic_cast, 483
new, 230; 430
reinterpret_cast, 490
sizeof, 227; 263
static_cast, 489
typeid, 474; 480; 486
XOR, 279; 221
ввода, 441
вывода, 441
декремента, 50
деления по модулю, 68
дополнения до 1, 221
И, поразрядный, 219
ИЛИ, поразрядный, 220
индексации, 340
инкремента, 50; 323
исключающее ИЛИ, 219; 221
НЕ, 221
присваивания, 38; 336
разрешения контекста, 297; 374
разрешения области видимости, 268; 297;374
разыменования адреса *, 523
Операторы, 68
арифметические, 68
декремента, 69
инкремента, 69
логические, 71

отношений, 71
поразрядные, 218
приведения типов, 483
присваивания,
составные, 225
сдвига, 222
Операция
приведения типов, 75
Опережающее объявление, 297
Отображение, 525; 545
Очередь, 526
приоритетная, 526
-П-
Параметры, 44

ссылочные, 181
формальные, 154
Перегрузка
конструкторов, 298
операторов, 319
ввода-вывода, 441
шаблона функции, 401
функций, 190
Переменные, 38
глобальные, 59; 154
инициализация, 66
локальные, 57
Перечисление, 214
fmtflags, 447
iostate, 470
openmode, 457
Позднее связывание, 393
Поле
сборное
ios::basefield, 448
ios::adjustfield, 448
ios::floatfield, 448
Полиморфизм, 28; 377
Полиморфный класс, 381; 475
Порожденная функция, 398
Поток, 439

cerr, 440
cin, 440
clog, 440
cout, 440
stderr, 585
stdin, 585
stdout, 585
wcerr, 440
wcin, 440
wclog, 440
wcout, 440
двоичный, 439
стандартный
ввода, 585
вывода, 585
ошибок, 585
текстовый, 439
Предикат, 524
Приоритетная очередь, 526
Пространство имен, 494
std, 35; 438; 500
неименованное, 499
Прототип функции, 43; 171
Псевдопеременные
__FILE__, 580
__LINE__, 580
-P-
Раннее связывание, 393
Распределитель памяти, 524
Расширение типа, 74
Реализация, 398
Рекурсия, 173
Ритчи, Дэнис, 23
-C-
Связывание
позднее, 393
раннее, 393
Специализация
класса
явная, 413
функции, 398

явная, 399
Спецификатор
explicit, 510
inline, 574
private, 355
protected, 357
public, 355
компоновки, 515
минимальной ширины поля, 586
Спецификатор класса памяти
auto, 206
extern, 206
register, 211
Спецификатор типа
const, 202
volatile, 204
Список, 536
сортировка, 541
Ссылки
на объекты, 291
на производные типы, 381
независимые, 188
Стандарт
C89, 584
C99, 584
Стандарт C, 23
Стандартная библиотека C++, 54
Стандартная библиотека шаблонов, 54

Стек, 152; 526
Страуструп, Бьерн, 25; 70
Строка, 36; 106
Строковый литерал, 106
Структура, 238
-T-
Таблица строк, 136
Тег, 255
Тип
basic_string, 559
BinPred, 524
bool. 57: 74

char, 56
clock_t, 298
double, 56
float, 56
int, 56
iterator, 523
nothrow_t, 436
off_type, 468
pos_type, 470
ptrdiff_t, 554
size_t, 432; 504
streamsize, 451
string, 559
UnPred, 524
void, 57
wchar_t, 56
wstring, 559
-У-
Указатели, 123
на объекты, 289
на производные типы, 378
на функции, 502
на член класса, 517
Управляющие последовательности, 65
Условное выражение, 79
-Ф-
Фабрика объектов, 478
Файл, 439
Файловый указатель, 592
Факториал числа, 174
Флаг
boolalpha, 448
dec, 448
fixed, 448
hex, 448
internal, 448
left, 448
oct, 448
right, 448
scientific, 448
showbase, 448

showbase, 448
showpoint, 448
showpos, 448
skipws, 448
unitbuf, 448
uppercase, 448
Флаг знака, 62
Формальные параметры, 58
Функции, 24; 147; 294
виртуальные, 381
встраиваемые, 283
манипуляторные, 454
обобщенные, 396
перегрузка, 190
сравнения, 524
Функция, 36; 41
abort(), 417; 419
abs(), 43; 167; 191; 403
asctime(), 252
assign(), 563
atof(), 164
bad(), 471
before(), 475
begin(), 529
clear(), 471
clock(), 213
close(), 458
compare(), 566
end(), 529
eof(), 463
erase(), 529; 532
exit(), 418; 419
fabs(), 191
fail(), 471
fclose(), 595
feof(), 595
ferror(), 597
fgetc(), 595
fill(), 451
find(), 565
flags(), 449
flush(), 467

fopen(), 593
fprintf), 600

fputc(), 594
fread(), 597
free(), 233
fscanf(), 600
fseek(), 599
fwrite(), 597
gcount(), 463
get(), 460; 465
getline(), 466
gets(), 107
good(), 471
insert(), 529; 532; 537
isalpha(), 114
kbhit(), 139
labs(), 191
localtime(), 251; 252
main(), 46; 162
make_pair(), 546
malloc(), 233; 431
merge(), 537
name(), 475
open(), 456
operator, 320
operator(), 525
peek(), 467
precision(), 451
printf(), 585
push_back(), 529; 537
push_front(), 537
put(), 460
putback(), 467
qsort(), 503
rand(), 138; 478
rdstate(), 470
read(), 461
remove(), 600
rewind(), 597

rmnd(), 565
scanf(), 588
seekg(), 468; 470
seekp(), 468; 470
setf(), 448
showflags(), 450
splice(), 537
strcat(), 109
strcmp(), 110
strcpy(), 109; 171
strlen(), 111; 161
tellg(), 470
tellp(), 470
terminate(), 417
time(), 251
tolower(), 113
toupper(), 113; 135
unexpected(), 427
unsetf(), 448
width(), 451; 452
write(), 461
операторная, 320
параметризованная, 45
порожденная, 398
преобразования, 519
шаблонная, 398
-Ц-
Цикл
do-while, 93
for, 49; 82
while, 91
бесконечный, 87
вложенный, 97
-Ш-
Шаблон, 396
Шаблонная функция, 398