



Шаг за шагом

Шилдт

C++

САМОУЧИТЕЛЬ

Шаг за шагом



ГЕРБЕРТ ШИЛДТ

Mc
Graw
Hill

Osborne

ЭКОМ

Mc
Graw
Hill

OSBORNE

ЭКОМ

C++ для начинающих

Шаг за шагом

C++

A Beginner's Guide

Step by Step



Osborne/McGraw-Hill

Герберт Шилдт

C++ для начинающих

Шаг за шагом

Москва

ЭКМ

2013

ББК 32.97

УДК 681.3

Ш58

Шилдт Герберт

Ш58 С++ для начинающих. Серия «Шаг за шагом»/ Шилдт Г.; пер. с англ. – М.: ЭКОМ Паблишерз, 2013. – 640 с.: ил.

Книга известного американского специалиста и популяризатора языков программирования, посвященная основам языка С++. Начиная с таких базовых понятий языка, как типы данных, массивы, строки, указатели и функции, книга охватывает также важнейшие элементы объектно-ориентированного программирования – классы и объекты, наследование, виртуальные функции, потоки вводавывода, исключения и шаблоны. Каждый раздел сопровождается простыми и наглядными примерами, позволяющими получить практические навыки современного программирования.

Книга предназначена для приступающих к изучению языка С++ – одного из самых универсальных и распространенных на сегодня языков программирования.

ISBN: 978-5-9790-0127-2

ISBN: 0-07-219467-7 (англ.)

© ЭКОМ Паблишерз, 2009

© 2005 by McGrawHill Companies

All rights reserved.

Printed in the United States of America.

Об авторе

Герберт Шилдт принадлежит к числу ведущих авторов мирового уровня в области программирования. Он является крупным специалистом по языкам C, C++, Java и C#, а также по программированию в системе Windows. Его книги по программированию были переведены на все основные иностранные языки и разошлись по миру в количестве более 3 миллионов экземпляров. Г. Шилдт — автор многочисленных бестселлеров, к числу которых относятся книги C++: *The Complete Reference*, C#: *The Complete Reference*, Java 2: *The Complete Reference*, C#: *A Beginner's Guide*, Java 2: *A Beginner's Guide*, а также C: *The Complete Reference*. Иллинойский Университет присвоил Г. Шилдту степень магистра по вычислительной технике. Связаться с ним можно в его консультационном отделе по телефону (217) 586-4683.

От переводчика

Эта книга представляет собой превосходно написанное пособие для приступающих к изучению языка C++ — одного из наиболее широко распространенных и универсальных языков программирования. Книга по C++ издано несметное количество. Зайдите в любой магазин технической книги — на полках всегда стоит несколько десятков книг такого рода, как переводных, так и написанных отечественными авторами. Среди них есть книги по самым разнообразным специальным аспектам программирования — использованию СОМ-технологий, распределенному программированию, разработке драйверов устройств и др. Есть и много пособий для начинающих. Чем среди этого изобилия выделяется настоящая книга?

- Книга написана чрезвычайно простым, доступным языком.

От читателя не требуется особых знаний по вычислительной технике или навыков программирования. От самого начала книги и до ее последних страниц новый материал вводится последовательно небольшими порциями и объясняется так просто и понятно, что книга легко читается, а излагаемые в ней понятия хорошо усваиваются. К наиболее сложным вопросам автор обращается несколько раз, терпеливо разъясняя суть дела.

- Книга написана интересно.

Очень многие книги по вычислительной технике характерны тем, что, приступая к их изучению, засыпаешь на первой же странице. Возможно, это связано с боязнью авторов отойти от скрупулезно точных формулировок и определений, которые всегда имеют несколько суточный характер. Книга Г.Шилдта не относится к этой категории; она написана так, что производит впечатление увлеченной беседы двух заинтересованных лиц — автора и читателя. Автор не боится объяснять сложные концепции современного программирования простым человеческим языком — не всегда исчерпывающе точным, но живым, образным и доходчивым.

- Все рассматриваемые темы и даже небольшие по объему разделы сопровождаются простыми, но весьма наглядными программными примерами.

Многие авторы помещают в свои книги по программированию сложные, развернутые примеры с массой не относящихся к делу деталей, среди которых иной раз трудно вычленить рассматриваемое средство языка. В книге Г.Шилдта все примеры исключительно просты и редко занимают больше одной-полтора страниц текста. По этой причине они, естественно, носят несколько формальный характер, однако их простота и наглядность с лихвой компенсируют этот недостаток. Читатель, выполняя приведенные в книге примеры на своем компьютере, может легко внести в них собственные изменения и дополнения, чтобы лучше понять иллюстрируемый в примере материал. Кстати,

автор подробно объясняет методику самостоятельной работы над примерами в разных операционных средах, а в приложении останавливается на особенностях использования относительно старых версий компиляторов, которые могут оказаться в распоряжении читателя.

- Будучи пособием для начинающих, книга, тем не менее, отнюдь не ограничивается изложением начальных сведений.

Наоборот, около половины книги посвящено наиболее сложному, но и наиболее ценному средству языка C++ — объектно-ориентированному программированию. Разумеется, объем и направленность книги заставили автора ограничиться рассмотрением лишь наиболее важных, принципиальных аспектов этого метода. Однако для начинающих этого вполне достаточно; желающим изучить объектно-ориентированное программирование в больших деталях автор рекомендует свои же книги с более глубоким изложением этого материала, но и более сложные в чтении.

Книгу Г.Шилдта можно адресовать самому широкому кругу читателей: школьникам, студентам младших курсов, инженерам, осознавшим необходимость научиться программировать на современном языке. Все они, изучив книгу, по достоинству оценят как силу и красоту языка C++, так и умение автора простыми словами излагать сложные вопросы.

К.Г. Финогенов

Предисловие

C++ является исключительно удобным языком для разработки высокопроизводительного программного обеспечения. Новые принципы синтаксиса и стиля, определенные в этом языке, повлияли на все последующие языки программирования. Например, и Java, и C# произошли от C++. Более того, C++ является языком, принятым международным программистским сообществом, и почти все профессиональные программисты по меньшей мере знакомы с этим языком. Овладевая C++, вы приобретаете твердый фундамент для дальнейшего углубленного изучения любых аспектов современного программирования.

Цель настоящей книги — познакомить вас с основами языка C++. Книга постепенно вводит вас в мир языка, используя при этом многочисленные примеры, вопросы для самопроверки и законченные программные проекты; при этом от вас не требуется иметь предварительный опыт программирования. Книга начинается с изложения таких базовых вопросов, как процедуры компиляции и запуска программ на C++; затем обсуждаются ключевые слова, возможности и конструкции, составляющие язык C++. Закончив изучение книги, вы получите твердое представление о существовании языка C++.

С самого начала следует подчеркнуть, что эта книга должна служить лишь отправным пунктом в изучении программирования. C++ является весьма сложным языком с огромными возможностями, и практическая работа на C++ не ограничивается использованием ключевых слов, операторов и синтаксических правил, составляющих язык. Успешное программирование на C++ требует использования богатого набора классов и библиотечных функций, способствующих разработке прикладных программ. Хотя некоторые элементы библиотек и рассматриваются в этой книге, ее ограниченный объем не позволил уделить этому вопросу должное внимание. Чтобы стать первоклассным программистом на C++, вы должны обязательно освоить библиотеки этого языка. Завершив чтение этой книги, вы приобретете знания, на базе которых сможете серьезно заняться изучением библиотек и всех остальных аспектов C++.

Как организована эта книга

Книга представляет собой последовательный учебник, в котором каждый следующий раздел опирается на предыдущий. Книга состоит из 12 модулей, в каждом из которых обсуждается тот или иной аспект

C++. При этом в книгу включены некоторые специальные элементы, способствующие закреплению изучаемого вами материала.

Цели

Каждый модуль начинается с перечисления поставленных в нем целей, говорящих о том, что именно вы будете изучать. В каждом модуле отмечены разделы, посвященные достижению этих целей.

Вопросы для самопроверки

Каждый модуль завершается перечнем вопросов для самопроверки, которые позволят вам оценить ваши знания. Ответы на эти вопросы приведены в Приложении А.

Минутная тренировка

В конце каждого тематического раздела приводится короткий список вопросов, отвечая на которые, вы сможете удостовериться в понимании ключевых тем данного раздела. Ответы на эти вопросы приводятся на той же странице.

Спросим у эксперта

По всей книге разбросаны вставки “Спросим у эксперта”. В них в виде вопросов и ответов приводятся дополнительные сведения или интересные комментарии к изучаемой теме.

Проекты

Каждый модуль содержит один или несколько программных проектов, показывающих, как применяется на практике изученный вами материал. Эти проекты представляют собой реальные примеры, которые вы сможете использовать как отправные точки для своих собственных программ.

Не требуется предварительного опыта программирования

Изучение книги не предполагает наличия у вас предварительного опыта программирования. Таким образом, вы можете приступить к

чтению книги, даже если вы никогда ранее не программировали. Конечно, реально в наше время большинство читателей будет иметь хотя бы небольшой опыт программирования на том или ином языке. Для большинства это будет опыт написания программ на Java или C#. Как вы увидите, C++ является прародителем обоих этих языков. Таким образом, если вы уже знакомы с Java или C#, изучение C++ не составит для вас большого труда.

Требуемое программное обеспечение

Для того, чтобы компилировать и запускать программы из этой книги, вам понадобится современный компилятор для C++, например, одна из последних версий Microsoft Visual C++ или Borland C++ Builder.

Не забудьте: все программы имеются во Всемирной паутине

Имейте в виду, что все примеры и программные проекты, включенные в эту книгу, можно бесплатно получить в Интернете по адресу www.osborne.com.

Для дальнейшего изучения

Книга C++ для начинающих открывает вам путь к целой серии книг Г. Шилдта по программированию. Ниже перечислены некоторые книги, которые могут представить для вас особый интерес.

Для более основательного знакомства с языком C++, познакомьтесь со следующими книгами:

C++: The Complete Reference

Имеется русский перевод: Г. Шилдт. Полный справочник по C++. — М. и др.: Вильямс, 2004. — 748 с.: ил.

C++ From the Ground Up

Имеется русский перевод: Г. Шилдт. C++ базовый курс. — М. и др.: Вильямс, 2005. — 748 с.: ил.

Teach Yourself C++

Имеется русский перевод: Г. Шилдт. Самоучитель C++. — СПб.: BHV-Санкт-Петербург, 1998. — 683 с.

STL Programming From the Ground Up

C++ Programmer's Reference

Для изучения программирования на языке Java мы рекомендуем такие книги:

Java 2: A Beginner's Guide

Имеется русский перевод: Г. Шилдт. Java 2. — СПб. и др.: BHV-Санкт-Петербург, 1998.

Java 2: The Complete Reference

Имеется русский перевод: П. Ноутон, Г. Шилдт. Полный справочник по Java 2. — СПб. и др.: БЧИ-Петербург, 2000. — 1050 с.: ил.

Java 2 Programmer's Reference

Для изучения программирования на языке C# Г. Шилдт предлагает вам следующее:

C#: A Beginner's Guide

Имеется русский перевод: Г. Шилдт. C#: учебный курс. — СПб. и др.: Питер, 2002. — 508 с.:ил.

C#: The Complete Reference

Имеется русский перевод: Г. Шилдт. Полный справочник по C#. — М. и др.: Вильямс, 2004. — 748 с.:ил.

Для изучения программирования в системе Windows мы рекомендуем целую группу книг Г. Шилдта:

*Windows 98 Programming From the Ground Up**Windows 2000 Programming From the Ground Up**MFC Programming From the Ground Up*

Имеется русский перевод: Г. Шилдт. MFC: Основы программирования. — Киев: BHV, 1997. — 556 с.:ил.

The Windows Programming Annotated Archives

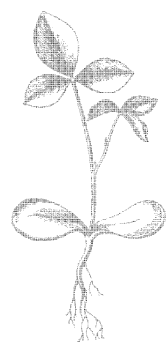
Если вы хотите ознакомиться с языком C, который создал базу всего современного программирования, вас могут заинтересовать следующие книги:

C: The Complete Reference

Имеется русский перевод: Г. Шилдт. Полный справочник по C. — М. и др.: Вильямс, 2002. — 699 с.:ил.

Teach Yourself C

Если вам нужно получить быстрые и точные ответы, обращайтесь за помощью к Герберту Шилдту, признанному специалисту по программированию.



Модуль 1 ОСНОВЫ C++

ЦЕЛИ, ДОСТИГАЕМЫЕ В ЭТОМ МОДУЛЕ

- 1.1 Ознакомиться с историей создания C++
- 1.2 Рассмотреть соотношение C++ и языков C, C# и Java
- 1.3 Узнать о трех принципах объектно-ориентированного программирования
- 1.4 Научиться подготавливать, компилировать и запускать программы на C++
- 1.5 Освоить использование переменных
- 1.6 Освоить использование операторов
- 1.7 Научиться вводить с клавиатуры
- 1.8 Научиться работать с предложениями **if** и **for**
- 1.9 Получить представление об использовании программных блоков
- 1.10 Начать использовать функции
- 1.11 Узнать о ключевых словах C++
- 1.12 Освоить понятие идентификатора

Если есть язык, определяющий существо современного программирования, то это язык C++. Он превосходно удовлетворяет требованиям разработки высокопроизводительного программного обеспечения. Его синтаксис стал стандартом для профессиональных языков программирования, а используемые в нем стратегические принципы проектирования проходят красной нитью через всю вычислительную технику. C++ послужил также основой для разработки таких языков, как Java и C#. Можно сказать, что C++ открыл путь ко всему современному программированию.

Цель этого модуля — дать вам представление о языке C++, включая его историю, присущую ему стратегию проектирования, а также наиболее важные средства этого языка. Безусловно, наибольшая сложность в изучении языка программирования заключается в том, что ни один элемент языка не существует в отрыве от других. Напротив, все компоненты языка работают совместно. Эта взаимозависимость затрудняет обсуждение какого-либо аспекта C++ без затрагивания других аспектов. С целью преодоления указанной трудности в этом модуле дается краткий обзор ряда средств C++, включая общую форму программы на этом языке, некоторые управляющие предложения и операторы языка. Здесь мы не будем вдаваться в детали, но обратим внимание на общие концепции, характерные для любых C++-программ.

Цель

1.1. Краткая история C++

История C++ начинается с языка C. Причина этого ясна: C++ построен на фундаменте C. Таким образом, C++ является надмножеством языка C. C++ был получен путем развития языка C и добавления в него аппарата поддержки объектно-ориентированного программирования (о чем будет подробнее сказано ниже). Кроме того, C++ отличается от C некоторыми дополнительными средствами, в частности, расширенным набором библиотечных программ. Однако значительную часть своей мощности и красоты язык C++ унаследовал непосредственно от C. Поэтому для того, чтобы полностью понять и оценить C++, вы должны прежде всего познакомиться с тем, как и почему возник язык C.

Язык C: Заря современной эры программирования

Изобретение языка C определило начало современной эры программирования. Влияние этого языка невозможно переоценить, поскольку он

фундаментально изменил наш подход и отношение к программированию. Стратегия проектирования, заложенная в язык С, и его синтаксис влияют на все появляющиеся языки программирования. Язык С явился одной из главных революционных сил в развитии программирования.

Язык С был изобретен и впервые реализован Деннисом Ритчи на машине PDP-11 фирмы DEC, использующей операционную систему UNIX. С явился результатом эволюционного процесса, начавшегося со старого языка под названием BCPL. Этот язык был разработан Мартином Ричардсом. Идеи BCPL нашли отражение в языке под названием В, который был изобретен Кеном Томпсоном, и который в дальнейшем привел к разработке в 1970-х гг. языка С.

Перед изобретением С компьютерные языки в основном разрабатывались либо в качестве академических упражнений, либо в административно создаваемых комитетах. С языком С получилось иначе. Этот язык был спроектирован, реализован и развит работающими программистами-практиками, и в нем нашли отражение их подходы к разработке программ. Средства этого языка были отточены, протестированы, продуманы и переработаны людьми, которые реально использовали язык в своей работе. В результате С сразу привлек массу сторонников и быстро стал любимым языком программистов всего мира.

Язык С вырос из революции *структурного программирования*, произошедшей в 1960-х гг. Перед появлением концепции структурного программирования написание больших программ вызывало значительные сложности, потому что логика построения программы имела тенденцию вырождаться в так называемый “макаронный код” — запутанную массу переходов, вызовов подпрограмм и возвратов, затрудняющих изучение и модификацию программы. Структурные языки подошли к решению этой проблемы путем добавления тщательно разработанных управляющих предложений, подпрограмм с локальными переменными и ряда других усовершенствований. С появлением структурных языков стало возможно писать относительно большие программы.

Хотя в то время были и другие структурные языки, в частности, Pascal, язык С оказался первым языком, в котором успешно сочетались мощь, элегантность и выразительность. Его лаконичный, но в то же время легко осваиваемый синтаксис вместе со стратегией, утверждающей главенство программиста (а не языка), быстро завоевал симпатии программистов. Сегодня это уже трудно представить, но С стал, можно сказать, той струей свежего воздуха, которую так долго ждали программисты. В результате С быстро превратился в наиболее широко используемый язык структурного программирования 1980-х гг.

Потребность в C++

Прочитав написанное выше, вы можете удивиться, зачем же был изобретен язык C++. Если С оказался успешным языком компьютерного

программирования, то почему возникла необходимость в чем-то другом? Ответ заключается в *тенденции к усложнению*. На протяжении всей истории программирования нарастающая сложность программ служила стимулом для разработки лучших способов справиться с нею. C++ появился как выражение этого процесса. Чтобы лучше понять связь возрастающей сложности программ с развитием компьютерных языков, придется немного углубиться в историю вычислительной техники.

Со времени изобретения вычислительной машины подходы к программированию претерпели драматические изменения. Когда появились первые компьютеры, их программирование выполнялось вводом машинных двоичных команд с передней панели компьютера с помощью переключателей. Пока размер программ не превышал нескольких сотен команд, такой подход был вполне работоспособным. Когда программы стали длиннее, был изобретен язык ассемблера, в результате чего программисты получили возможность работать с более объемными и сложными программами, используя символическое представление машинных команд. Дальнейшее усложнение программ привело к появлению языков высокого уровня, предоставивших программистам дополнительные средства, облегчающие работу со сложными программами.

Первым широко используемым компьютерным языком был, конечно, FORTRAN. Хотя FORTRAN был весьма впечатляющим первым шагом, вряд ли его можно считать языком, способствующим написанию ясных, легко понимаемых программ. В 1960-х гг. родилось структурное программирование, новый метод, использование которого было естественно при работе на языках вроде С. С появлением структурного программирования впервые стало возможным написание относительно сложных программ без большой затраты труда и времени. Однако при достижении программным проектом определенного размера даже с использованием методов структурного программирования сложность проекта могла превысить возможности программиста. К концу 1970-х гг. многие проекты стали приближаться к этой границе.

В качестве пути решения этой проблемы стал разрабатываться новый принцип написания программ, получивший название *объектно-ориентированного программирования* (ООП). С помощью ООП программист получил возможность создавать более длинные и сложные программы. Однако язык С не поддерживал объектно-ориентированное программирование. Желание иметь объектно-ориентированную модификацию С привело к созданию C++.

Таким образом, хотя С является одним из наиболее любимых и широко используемых профессиональных языков программирования в мире, наступил момент, когда сложность разрабатываемых программ превысила его возможности. При достижении программой определенного размера она становится настолько сложной, что программист уже не может воспринять ее, как единое целое. Целью C++ является преодоление этого барьера и предоставление программисту возможности понимать и обслуживать более сложные программы.

C++ родился

C++ был изобретен Бьярном Страуструпом в 1979 г. в Bell Laboratories в Муррей Хилл, штат Нью-Джерси. Сначала Страуструп назвал новый язык “C с классами”. Однако в 1983 г. язык получил название C++.

Страуструп построил C++ на базе C, сохранив все возможности C, его характерные особенности и достоинства. Он перенял также и базовую стратегию C, согласно которой программист, а не язык определяет структуру программы. Важно понимать, что Страуструп не создал совершенно новый язык программирования. Напротив, он расширил уже имеющийся весьма успешный язык.

Большая часть средств, добавленных Страуструпом к C, предназначена для поддержки объектно-ориентированного программирования. В сущности, C++ является объектно-ориентированной модификацией C. Построив новый язык на базе C, Страуструп обеспечил плавный переход к внедрению ООП. Вместо того, чтобы изучать совершенно новый язык, программисту, работающему на C, было достаточно освоить лишь несколько новых средств, после чего он мог пожинать все плоды методологии объектно-ориентированного программирования.

Создавая C++, Страуструп понимал, что добавляя к языку C поддержку объектно-ориентированного программирования, необходимо было в то же время сохранить дух языка C, включая его эффективность, гибкость и стратегию программирования. К счастью, эта цель была достигнута. C++ предоставляет программисту все возможности C, добавляя к ним могущество объектов.

Хотя первоначально C++ создавался в качестве средства разработки очень больших программ, его использование отнюдь не ограничивается этой сферой. В действительности объектно-ориентированные средства C++ могут эффективно применяться для решения практически любых задач программирования. Легко найти примеры разработки на C++ таких программных проектов, как текстовые или графические редакторы, базы данных, персональные файловые системы, сетевые утилиты и коммуникационные программы. В силу того, что C++ воспринял эффективность C, его часто используют для разработки высокопроизводительного системного программного обеспечения. Он также широко используется и для программирования в системе Windows.

Эволюция C++

С момента своего появления C++ претерпел три серьезных модификации, каждая из которых расширяла и изменяла язык. Первая модификация была предложена в 1985 г., вторая — в 1990 г. Третья модификация появилась в процессе стандартизации C++. Несколько лет назад

началась разработка стандарта для C++. С этой целью организациями ANSI (Американский национальный институт стандартов) и ISO (Международная организация по стандартизации) был создан объединенный комитет по стандартизации. Первый вариант предлагаемого стандарта был выпущен 25 января 1994 г. В этот вариант комитет ANSI/ISO по C++ (членом которого я являлся) включил средства, определенные Страуструпом, с добавлением некоторых новых. В целом первый вариант стандарта отражал состояние C++ на тот момент времени.

Вскоре после завершения работы над первым вариантом стандарта C++ произошло событие, которое заставило серьезно расширить стандарт: создание Александром Степановым стандартной библиотеки шаблонов (Standard Template Library, STL). STL представляет собой набор обобщенных процедур, с помощью которых можно обрабатывать данные. Эта библиотека сочетает в себе могущество и элегантность. Но она также весьма велика. Выпустив первый вариант стандарта C++, комитет проголосовал за включение STL в спецификацию C++. Добавление STL расширило возможности C++ далеко за пределы его первоначального определения. С другой стороны, включение STL, хотя и вполне оправданное, замедлило процесс стандартизации C++.

Уместно заметить, что стандартизация C++ заняла значительно больше времени, чем это можно было предполагать. В процессе стандартизации к языку было добавлено много новых средств, а также внесено большое количество мелких изменений. В результате вариант C++, определенный комитетом ANSI/ISO, оказался значительно объемнее и сложнее первоначального замысла Страуструпа. Окончательный набросок стандарта был выпущен комитетом 14 ноября 1997 г., и весь стандарт стал действовать в 1998 г. Эту спецификацию C++ обычно называют *стандартным C++*.

Настоящая книга описывает как раз стандартный C++. Это вариант C++, поддерживаемый всеми основными компиляторами языка C++, включая Visual C++ фирмы Microsoft и C++ Builder фирмы Borland. Таким образом, все программы и вообще все сведения, излагаемые в этой книге, являются переносимыми.

Цель

1.2. Как C++ соотносится с языками Java и C#

Как скорее всего известно большинству читателей, относительно недавно появилось два новых языка: Java и C#. Язык Java был разработан фирмой Sun Microsystems; C# создан фирмой Microsoft. Поскольку иногда возникает непонимание того, как связаны эти два языка с языком C++, мы кратко обсудим этот вопрос.

C++ является родительским языком по отношению к языкам Java и C#. Хотя и в Java, и в C# добавлены, удалены или модифицированы некоторые свойства, в целом синтаксис всех этих трех языков практически одинаков. Более того, объектная модель, используемая в C++, аналогична тем, что используются языками Java и C#. Наконец, общее ощущение программиста при работе на этих языках оказывается весьма схожим. Это значит, что если вы знаете C++, вы с легкостью изучите Java или C#. Справедливо также и обратное. Если вы знакомы с Java или C#, изучить C++ не составит труда. Использование в Java и C# синтаксиса и объектной модели C++ имело в виду именно эту цель; оно обеспечило быстрое принятие новых языков сонмом опытных C++-программистов.

Основное отличие Java и C# от C++ заключается в том, что они предназначены для разных вычислительных сред. C++ разрабатывался с целью обеспечения создания высокопроизводительных программ для конкретного типа процессора и определенной операционной системы. Например, если вы хотите написать программу, которая будет выполняться на процессоре Pentium фирмы Intel под управлением операционной системы Windows, то лучше C++ вы ничего не найдете.

Java и C# создавались для обеспечения специфических требований, возникающих при написании программ реального времени, работающих в среде сети Интернет (C# также имел своей целью задачу упрощения разработки программных компонентов). Интернет объединяет много разных типов процессоров и операционных систем. С его широким распространением особое значение приобрела возможность создания межплатформенных, переносимых программ.

Первым языком, решающим эту задачу, стал Java. Используя этот язык, можно писать программы, работающие в самых различных вычислительных средах. Java-программа может свободно перемещаться в сети Интернет. Однако, ценой, которую вы платите за переносимость, является эффективность; Java-программы выполняются медленнее программ, написанных на C++. То же справедливо и по отношению к C#. Вывод таков: если вы разрабатываете высокопроизводительное программное обеспечение, используйте C++. Если вам нужны высокоомобильные, переносимые программы, используйте Java или C#.

СПРОСИМ У ЭКСПЕРТА

Вопрос: Каким образом Java и C# создают межплатформенные, переносимые программы, и почему C++ не обладает этой возможностью?

Ответ: Java и C#, в отличие от C++, могут создавать межплатформенные, переносимые программы благодаря формату объектного модуля, формируемого компилятором. В случае C++ компилятор формирует машинный код, который затем может непосредственно выполняться центральным

процессором. Таким образом, объектный модуль привязан к конкретному процессору и определенной операционной системе. Если вы захотите выполнить C++-программу в другой системе, вам придется перекомпилировать исходный текст программы в машинный код, предназначенный для выбранной вами среды. Для того, чтобы иметь возможность выполнять C++-программу в различных средах, необходимо создать для нее несколько различных вариантов выполнимых модулей.

Переносимость программ, написанных на Java или C#, объясняется тем, что эти программы компилируются не в машинный, а в псевдокод, представляющий собой некоторый промежуточный язык. В случае Java этот промежуточный язык носит название *байт-кода*. В случае C# он называется *промежуточным языком Microsoft* (Microsoft Intermediate Language, MSIL). В обоих случаях псевдокод выполняется специальной исполняющей системой. Java-программы используют виртуальную машину Java (Java Virtual Machine, JVM). Для C#-программ соответствующая система носит название Common Language Runtime (CLR). В результате Java-программа может выполняться в любой среде, для которой имеется виртуальная машина Java; C#-программа может выполняться в любой среде, если для нее имеется реализация CLR.

Поскольку исполняющие системы Java и C# располагаются в логическом плане между программой и центральным процессором, при выполнении Java- и C#-программ возникают издержки, которые отсутствуют у C++-программ. Поэтому C++-программы выполняются как правило быстрее, чем эквивалентные им программы, написанные на Java или C#.

Минутная тренировка

1. От какого языка произошел C++?
 2. Какое главное обстоятельство привело к созданию C++?
 3. C++ является родительским языком по отношению к Java и C#. Правильно ли это?
-
1. C++ произошел от C.
 2. Главным обстоятельством, приведшим к созданию C++, явилась возрастающая сложность программ.
 3. Правильно.
-

Цель

1.3.

Объектно-ориентированное программирование

Центральным свойством C++ является поддержка объектно-ориентированного программирования (ООП). Как уже отмечалось, ООП послужило стимулом к созданию C++. В силу этого стоит рассмотреть

хотя бы самые базовые принципы ООП перед тем, как вы приступите к написанию даже простейшей C++-программы.

Объектно-ориентированное программирование вобрало в себя лучшие идеи структурного программирования и скомбинировало их с некоторыми новыми концепциями. В результате возник новый и лучший способ организации программы. Вообще программу можно организовать двумя способами, положив во главу угла либо коды (описывающие, что происходит), либо данные (над которыми выполняются действия). Если программа использует только методы структурного программирования, то обычно в основе ее организации лежат коды. При таком подходе можно сказать, что “коды воздействуют на данные”.

Объектно-ориентированные программы работают как раз наоборот. В основе их организации лежат данные, и их принцип заключается в том, что “данные управляют доступом к коду”. Используя объектно-ориентированный язык, вы определяете данные и процедуры, которым разрешается обрабатывать эти данные. В результате тип данного однозначно определяет, какого рода операции допустимо выполнять над этим данным.

С целью поддержки принципов объектно-ориентированного программирования все объектно-ориентированные языки, включая C++, обеспечивают три характерных принципа: инкапсуляцию, полиморфизм и наследование. Рассмотрим эти принципы подробнее.

Инкапсуляция

Инкапсуляция представляет собой программный механизм, который связывает данные с обрабатывающими их кодами и защищает их, и другие от внешних воздействий и ошибочных действий. В объектно-ориентированном языке коды и данные могут быть связаны так, что вместе они создают автономный *черный ящик*. Внутри этого ящика содержатся все необходимые данные и коды. При связывании таким образом данных и кодов создается *объект*. Другими словами, объект представляет собой устройство, поддерживающее инкапсуляцию.

Внутри объекта коды или данные или и те, и другие могут иметь атрибут *private*, что делает их *закрытыми* для внешнего мира, или *public*, что *открывает* эти элементы объекта. Закрытые коды и данные известны и доступны только из других частей того же объекта. Другими словами, к закрытым кодам и данным нельзя обратиться ни из какого фрагмента программы, существующего вне объекта. Если же код или данные объявлены с атрибутом **public**, то доступ к ним открыт из любых частей программы, несмотря на то, что эти элементы определены внутри объекта. Обычно открытые элементы объекта используются для обеспечения контролируемого интерфейса с закрытыми элементами того же объекта.

В C++ базовой единицей инкапсуляции является *класс*. Класс определяет содержание объекта. Класс описывает как данные, так и коды, предназначенные для операций над этими данными. C++ использует спецификацию класса при конструировании *объектов*. Объекты являются экземплярами класса. Таким образом, класс в сущности представляет собой набор чертежей, по которым строится объект.

Код и данные, составляющие класс, называются *членами* класса. Конкретно, *члены-переменные*, называемые также *переменными экземпляра*, — это данные, определенные в классе. *Члены-функции*, или просто *функции* — то коды, предназначенные для операций над данными. Функция — это термин C++, обозначающий подпрограмму.

СПРОСИМ У ЭКСПЕРТА

Вопрос: Я сталкивался с обозначением подпрограммы термином *метод*. Метод и функция — это одно и то же?

Ответ: В общем случае, да. Термин *метод* получил распространение вместе с языком Java. То, что программисты на C++ называют функцией, Java-программисты обозначают словом метод. Этот термин стал так широко использоваться, что его часто применяют и по отношению к функциям C++.

Полиморфизм

Полиморфизм (от греческого слова, означающего “много форм”) обозначает средство, позволяющее посредством единого интерфейса получить доступ к целому классу действий. Простым примером полиморфизма может служить рулевое колесо автомобиля. Рулевое колесо (интерфейс) остается одним и тем же, независимо от того, какой тип рулевого механизма используется в данном автомобиле. Другими словами, рулевое колесо действует одинаково для любых автомобилей: с непосредственным приводом на колеса, с гидравлическим усилителем или с реечной передачей. Поворот рулевого колеса влево заставляет автомобиль двигаться влево независимо от типа рулевого механизма. Достоинство единого интерфейса, очевидно, заключается в том, что если вы умеете пользоваться рулевым колесом, вы можете ездить на любых автомобилях.

Тот же принцип используется в программировании. Рассмотрим, например, стек (список, действующий по правилу “первым вошел, последним вышел”). Пусть вашей программе требуется три стека различных видов. Один стек используется для целых чисел, другой для чисел с плавающей точкой, а третий для одиночных символов. Алгоритм реализации всех трех стеков будет одинаков, несмотря на то, что

данные, заносимые в разные стеки, различаются. Если вы используете не объектно-ориентированный язык, вам придется создать три набора программ обслуживания стеков, и для каждого набора использовать различающиеся имена. Однако в C++, учитывая наличие полиморфизма, вы можете создать один обобщенный набор программ обслуживания стеков, который будет одинаково хорошо работать во всех трех ситуациях. В этом случае, если вы знаете, как использовать один стек, вы точно так же сможете использовать все три.

В общем случае концепция полиморфизма часто выражается фразой “один интерфейс, много методов”. Это означает возможность работать обобщенный интерфейс для группы схожих действий. Полиморфизм уменьшает сложность программы, обеспечивая обращение посредством одного интерфейса к *обобщенному классу действий*. Выбор же *конкретного действия* (другими словами, метода), соответствующего каждой ситуации, возлагается на компилятор. Вам, программисту, не требуется осуществлять этот выбор вручную. Вам только нужно помнить о наличии обобщенного интерфейса и использовать его в необходимых случаях.

Наследование

Наследование является процессом, который позволяет одному объекту приобретать свойства другого объекта. Важность наследования определяется тем, что оно поддерживает концепцию иерархической классификации. Легко сообразить, что большая часть наших знаний построена по иерархическому (т. е. сверху вниз) принципу. Например, антоновка является частью класса *яблок*, который, в свою очередь, есть часть класса *фруктов*; фрукты же входят в более широкий класс *пищевых продуктов*. Класс *пищевые продукты* обладает определенными качествами (съедобность, пищевая ценность и т. д.), которые, логично предположить, приложимы и к его подклассу *фрукты*. В дополнение к этим качествам класс *фрукты* обладает специфическими характеристиками (сочность, сладость и др.), которые выделяют его среди других пищевых продуктов. Класс *яблоки* определяет качества, характерные для яблок (растут на деревьях, не являются тропическими продуктами и т. д.). Класс *антоновка*, в свою очередь, наследует все качества всех предшествующих классов и определяет лишь те качества, которые выделяют антоновку среди прочих яблок.

Если не пользоваться иерархией, то каждый объект должен был бы явно определять все свои характеристики. При использовании же наследования объект определяет лишь те качества, которые делают его уникальным в рамках его класса. Более общие качества он может наследовать от родительского класса. Таким образом, механизм наследования позволяет объекту быть специфическим экземпляром более общего класса.

Минутная тренировка

1. Перечислите принципы ООП.
 2. Что является базовой единицей инкапсуляции в C++?
 3. Как называются подпрограммы в C++?
1. Принципами ООП являются инкапсуляция, полиморфизм и наследование.
 2. Базовой единицей инкапсуляции в C++ является класс.
 3. Подпрограммы в C++ называются *функциями*.

СПРОСИМ У ЭКСПЕРТА

Вопрос: Вы утверждаете, что объектно-ориентированное программирование (ООП) является эффективным способом работы с большими программами. Однако представляется, что использование ООП может привести к существенным издержкам при работе с небольшими программами. Справедливо ли это применительно к языку C++?

Ответ: Нет. Ключевым свойством языка C++ является то, что он *позволяет* писать объектно-ориентированные программы, но *не требует*, чтобы вы использовали ООП. В этом заключается одно из важных отличий C++ от языков Java/C#, которые строго базируются на объектной модели, требующей, чтобы любая программа была хотя бы в минимальной степени объектно-ориентированной. В противоположность этому C++ предоставляет вам право выбора. Кроме того, большая часть объектно-ориентированных средств C++ прозрачны во время выполнения программы, поэтому му издержки оказываются незначительными или даже вовсе отсутствуют.

Цель

1.4. Первая простая программа

Наконец наступило время заняться программированием. Начнем с компиляции и запуска приведенной ниже короткой C++-программы.

```
/*  
    Это простая C++-программа.  
  
    Назовите этот файл Sample.cpp.  
*/  
  
#include <iostream>  
using namespace std;  
  
// C++-программа начинает свое выполнение с функции main().  
int main()
```



```
{  
    cout << "C++ является мощным программным средством.";  
  
    return 0;  
}
```

Вы должны выполнить следующие три шага:

1. Ввести текст программы.
2. Откомпилировать программу.
3. Запустить программу.

Перед тем, как приступить к описанию этих шагов, уточним значение двух терминов: исходный код и объектный код. *Исходным кодом* называется текст программы, написанный на том или ином языке; исходный код пишется и читается программистом. *Объектный код* представляет собой машинную, выполняемую процессором форму программы. Объектный код создается компилятором.

Ввод программы

Программы, обсуждаемые в этой книге, можно получить на веб-сайте www.osborne.com. Однако программы можно вводить и вручную. Собственноручный ввод программ поможет вам запомнить ключевые правила. Для ввода программ вручную вам понадобится текстовый редактор (не текстовый процессор). Текстовые процессоры обычно вместе с текстом сохраняют информацию о форматах. Эта информация поставит в тупик компилятор C++. Если вы работаете на платформе Windows, вы можете воспользоваться редактором WordPad (Блокнот) или любой другой привычной для вас программой ввода текста.

Имя файла, в котором содержится исходный код, в принципе может быть каким угодно. Однако C++-программы обычно хранятся в файлах с расширением **.cpp**. Таким образом, вы можете дать файлу с C++-программой любое имя, но он должен иметь расширение **.cpp**. Подготавливая первый пример, назовите исходный файл **Sample.cpp**, чтобы вам было легче читать последующие пояснения. Для большинства остальных программ этой книги вы можете выбирать имена произвольно.

Компиляция программы

Способ компиляции файла **Sample.cpp** зависит от компилятора и от того, какие опции вы используете. Кроме того, многие компиляторы, в частности, Microsoft Visual C++ и Borland C++ Builder, предоставляют два различающихся способа компиляции программы:

компилятор командной строки и интегрированную среду разработки (Integrated Development Environment, IDE). Поэтому затруднительно привести обобщенные инструкции по компиляции C++-программы. Вам придется познакомиться с инструкциями к вашему компилятору.

Все же мы приведем некоторые рекомендации. Если вы используете Visual C++ или C++ Builder, то простейший способ компиляции и запуска программ этой книги заключается в использовании компиляторов командной строки, входящих в эти пакеты. Например, для компиляции **Sample.cpp** с помощью Visual C++ вам следует ввести на командной строке следующую команду:

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

Опция **-GX** ускоряет компиляцию. Чтобы использовать компилятор командной строки Visual C++, вы должны прежде всего запустить командный файл **VCVARS32.BAT**, входящий в пакет Visual C++. (Visual Studio .NET также предоставляет готовую среду командной строки, которую можно активизировать, выбрав пункт Visual Studio .NET Command Prompt из списка инструментальных средств, перечисленных в меню Microsoft Visual Studio .NET, которое активизируется выбором **Start | Programs [Пуск | Программы]** с панели задач). Чтобы откомпилировать **Sample.cpp** с помощью C++ Builder, введите следующую команду:

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

В результате компиляции вы получите выполнимый объектный код. В среде Windows выполнимый файл будет иметь то же имя, что и исходный файл, но расширение **.exe**. Таким образом, выполнимый вариант программы **Sample.cpp** будет находиться в файле **Sample.exe**.

Запуск программы

После того, как программа откомпилирована, ее можно запустить на выполнение. Поскольку компилятор C++ создает выполнимый объектный код, для запуска программы достаточно ввести ее имя на командной строке. Так, для запуска программы **Sample.exe** введите следующую команду:

```
C:\...\>Sample
```

Выполняясь, программа выведет на экран следующую строку:

C++ является мощным программным средством.

Если вы используете интегрированную среду разработки (IDE), вы сможете запустить программу, выбрав в меню пункт Run. Познакомьтесь с инструкциями к вашему конкретному компилятору. Впрочем, программы этой книги проще компилировать и запускать с командной строки.

Последнее замечание. Программы этой книги являются консольными приложениями, которые выводят свои результаты не в окно Windows, а на экран DOS. Другими словами, они выполняются в сеансе командной строки. Разумеется, C++ превосходно подходит для программирования в системе Windows. Более того, он является наиболее распространенным языком разработки прикладных программ для системы Windows (приложений Windows). Однако, программы этой книги не используют графический интерфейс пользователя системы Windows (Graphic User Interface, GUI). Причину этого легко понять: программы для Windows по своей природе сложны и велики по объему. Для создания даже простейшего приложения Windows потребуется от 50 до 70 строк кода, которые можно считать издержками графического интерфейса. Если для демонстрации свойств C++ составлять приложения Windows, то каждое будет содержать сотни строк кода. С другой стороны, консольные программы оказываются значительно короче и именно их обычно используют при обучении языку. После того, как вы освоили C++, вы сможете без труда приложить свои знания к разработке приложений Windows.

Первый программный пример строка за строкой

Программа **Sample.cpp**, несмотря на свою краткость, включает несколько ключевых средств, общих для всех C++-программ. Рассмотрим внимательно каждую часть этой программы. Программа начинается со строк

```
/*  
Это простая C++-программа.  
  
Назовите этот файл Sample.cpp.  
*/
```

Эти строки представляют собой *комментарий*. Как и большинство языков программирования, C++ допускает включать в исходный код программы замечания программиста. Содержимое комментариев игнорируется компилятором. Цель комментариев – описать или объяснить выполняемые программой операции любому, кто будет читать ее текст. В нашем примере комментарий идентифицирует программу. В

более сложных программах вы будете использовать комментарии для того, чтобы объяснить, для чего введена в программу каждая ее деталь, и каким образом она выполняет свои функции. Другими словами, с помощью комментариев вы даете описание того, что делает ваша программа, как бы “с места события”.

В C++ используются комментарии двух видов. Один, который вы только что видели, называется *многострочным комментарием*. Этот вид комментария начинается со знаков /* (знак деления, за которым стоит звездочка). Такой комментарий заканчивается только когда в программе встретится комбинация */. Весь текст, располагаемый между этими двумя знаками комментария, полностью игнорируется компилятором. Многострочные комментарии могут содержать одну или несколько строк. Второй вид комментариев (однострочных) также демонстрируется в нашей программе; мы рассмотрим его чуть позже.

Следующая строка кода выглядит таким образом:

```
#include <iostream>
```

Язык C++ определяет несколько *заголовков*, которые содержат информацию либо необходимую, либо полезную для вашей программы. Рассматриваемая программа требует подключения заголовка `<iostream>`, который поддерживает систему ввода-вывода C++. Этот заголовок поставляется вместе с компилятором. Заголовки включаются в программу с помощью директивы **#include**. Позже мы рассмотрим более подробно назначение заголовков и их использование программой.

Далее в программе стоит такая строка:

```
using namespace std;
```

Эта строка указывает компилятору, что он должен использовать *пространство имен std*. Пространства имен являются относительно новым добавлением к C++. Подробное обсуждение этого средства будет проведено позже; здесь мы дадим только его краткое описание. Пространство имен создает декларативный район, в который помещаются различные элементы программы. Элементы, объявленные в одном пространстве имен, отделены от элементов, объявленных в другом пространстве. Пространства имен оказывают помощь в организации больших программ. Предложение **using** информирует компилятор о том, что вы хотите использовать пространство имен `std`. В этом пространстве имен объявлена вся библиотека стандартного C++. Используя пространство имен `std`, вы упрощаете доступ к стандартной библиотеке. (Поскольку пространства имен являются относительно новым средством, ваш компилятор может их не поддерживать. В этом случае см. Приложение С, в котором описан обходный путь.)

Следующая строка программы представляет собой комментарий:

```
// C++-программа начинает свое выполнение с функции main().
```

В этой строке демонстрируется второй вид комментариев, допустимых в C++: *однострочный комментарий*. Этот вид комментария начинается со знаков `//` и заканчивается в конце строки. Обычно программисты используют многострочные комментарии для включения в программу длинных детальных пояснений, а однострочные — когда требуется лишь краткое замечание. Разумеется, выбор вида комментария и стиля его использования — это вопрос вкуса программиста.

Следующая строка, как это поясняется в предшествующем комментарии, представляет собой начало программы:

```
int main()
```

Все C++-программы состоят из одной или нескольких функций. Как уже отмечалось ранее, функция — это подпрограмма. Любая функция C++ должна иметь имя, при этом единственная функция, которая должна включаться в каждую C++-программу, называется **main()**. Функция **main()** — это то место в программе, где начинается и где (чаще всего) заканчивается ее выполнение. (Строго говоря, выполнение C++-программы начинается с вызова функции **main()**, и в большинстве случаев заканчивается, когда осуществляется возврат из этой функции.) Открывающая фигурная скобка, которая стоит в следующей строке, отмечает начало кода функции **main()**. Слово **int**, предшествующее **main()**, задает тип данного, возвращаемого функцией **main()**. Как вы узнаете позже, C++ поддерживает несколько встроенных типов данных, и **int** является одним из них. Это обозначение происходит от слова *integer* (целое).

Далее в программу включена строка:

```
cout << "C++ является мощным программным средством.";
```

Это предложение консольного вывода. Оно приводит к выводу на экран сообщения C++ **является мощным программным средством**. Вывод на экран осуществляется с помощью оператора вывода `<<`. Оператор `<<` действует так, что выражение (каким бы оно ни было), стоящее справа от него, выводится на устройство, указанное слева. **cout** представляет собой предопределенный идентификатор, обозначающий консольный вывод, который, как правило, закреплен за экраном. Таким образом, рассматриваемое предложение выводит на экран сообщение. Заметьте, что это предложение заканчивается знаком точки с запятой. Так заканчиваются все предложения C++.

Сообщение “C++ является мощным программным средством.” представляет собой *строку*. В C++ строкой называется последовательность символов, заключенная в двойные кавычки. Строки широко используется в программах на C++.

Следующая строка программы осуществляет выход из **main()**:

```
return 0;
```

Эта строка завершает функцию **main()** и заставляет ее вернуть значение 0 в вызывающий процесс (которым обычно является операционная система). Для большинства операционных систем возвращаемое значение 0 указывает на то, что программа завершается правильно. Другие значения свидетельствуют о завершении программы в результате возникновения какой-либо ошибки. **return** является одним из ключевых слов C++ и используется для возврата значения из функции. Все ваши программы должны при нормальном (т. е. без ошибок) завершении возвращать 0.

Завершающая фигурная скобка в конце программы формально заканчивает программу.

Обработка синтаксических ошибок

Введите, откомпилируйте и выполните предыдущую программу (если вы этого еще не сделали). Как вы, возможно, знаете по опыту, при вводе текста программы в ваш компьютер легко случайно напечатать что-то неправильно. К счастью, если в программе встречается неправильная строка, компилятор, пытаясь откомпилировать программу, сообщает о *синтаксической ошибке*. Большинство компиляторов пытается найти смысл в программе, что бы вы в ней ни написали, и поэтому сообщение об ошибке, выводимое компилятором, не всегда будет соответствовать действительной ошибке. Если, например, в предыдущей программе случайно опустить открывающую фигурную скобку после **main()**, то компилятор может указать на предложение **cout**, как место синтаксической ошибки. Если вы получаете сообщение о синтаксической ошибке, то для нахождения ошибочного места вам, возможно, придется проанализировать несколько последних строк кода.

Минутная тренировка

1. Где начинается выполнение C++-программы?
2. Что такое **cout**?
3. К чему приводит использование предложения **#include <iostream>**?

1. C++-программа начинает выполнение с функции `main()`.
2. `cout` представляет собой предопределенный идентификатор, связанный с консольным выводом.
3. Это предложение включает в текст программы заголовок `<iostream>`, который поддерживает ввод-вывод.

Спросим у эксперта

Вопрос: Помимо сообщений об ошибках, мой компилятор может выводить предупреждающие сообщения нескольких видов. Чем предупреждения (warnings) отличаются от ошибок (errors) и какие виды предупреждающих сообщений следует активизировать?

Ответ: Большинство C++-компиляторов, помимо сообщений о фатальных синтаксических ошибках, могут еще выводить предупреждающие сообщения нескольких видов. Сообщения об ошибках возникают, если в программе имеются безусловно неправильные места, например, отсутствует точка с запятой в конце предложения. Предупреждающие сообщения говорят о наличии подозрительного, но формально правильного кода. Решать, насколько эти подозрения оправданы, придется вам, программисту.

Предупреждения также используются для указания на неэффективные конструкции или устаревшие средства языка. Обычно вы можете сами выбрать, какого рода предупреждающие сообщения вы хотите видеть. Программы этой книги написаны в соответствии со стандартным C++, и, если их ввести без ошибок, они при компиляции не будут генерировать какие-либо предупреждающие сообщения.

Для примеров этой книги вы можете использовать режим вывода сообщений, устанавливаемый компилятором по умолчанию ("нормальный" режим). Однако при этом вам стоит обратиться к документации по вашему компилятору и определить, какие возможности вывода предупреждающих сообщений имеются в вашем распоряжении. Многие компиляторы обладают весьма изощренными средствами, которые могут помочь вам обнаруживать тонкие ошибки перед тем, как они вырастут в большие проблемы. Изучение системы предупреждений вашего компилятора вполне стоит затраченного труда и времени.

Цель

1.5.

Вторая простая программа

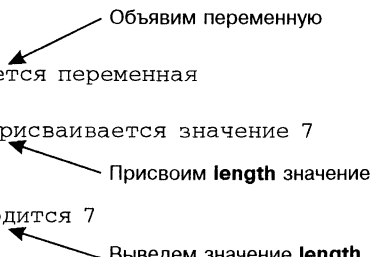
Возможно, в программировании нет более фундаментальной конструкции, чем переменная. *Переменная* представляет собой именованную ячейку памяти, которой может быть присвоено определенное значение. В дальнейшем, по ходу выполнения программы, значение переменной может быть изменено. Другими словами, содержимое переменной не фиксируется, а поддается изменениям.

Приводимая ниже программа создает переменную с именем **length** (длина), присваивает ей значение 7 и затем выводит на экран сообщение **Длина равна 7**.

```
// Использование переменной.

#include <iostream>
using namespace std;

int main()
{
    int length; // здесь объявляется переменная
    length = 7; // здесь length присваивается значение 7
    cout << " Длина равна ";
    cout << length; // здесь выводится 7
    return 0;
}
```



Как уже отмечалось ранее, имена в C++ можно выбирать произвольно. Так, вводя текст программы, вы можете присвоить файлу с программой любое удобное для вас имя. Например, файлу с этой программой можно дать имя **VarDemo.cpp**.

В этой программе вводятся две новых концепции. Во-первых, предложение

```
int length; // здесь объявляется переменная
```

объявляет переменную с именем **length** типа **int** (целое число). В C++ все переменные, перед тем, как их использовать, должны быть объявлены, при этом в объявлении переменной должно быть указано, какого рода значения будут содержаться в этой переменной. Эта называется *типом* переменной. В нашем случае **length** может содержать только целочисленные значения, т. е. целые числа в диапазоне от -32768 до 32767. Для того, чтобы в языке C++ объявить целочисленную переменную, перед ее именем следует поставить ключевое слово **int**. В дальнейшем вы увидите, что C++ поддерживает большое количество встроенных типов переменных. (Вы можете создавать и собственные типы данных.)

Второе важное средство используется в следующей строке кода:

```
length = 7; // здесь length присваивается значение 7
```

Как следует из комментария, в этом предложении переменной **length** присваивается значение 7. В C++ в качестве оператора присваи-

вания используется один знак равенства. Этот оператор копирует значение, находящееся справа от него, в переменную, имя которой указано слева от оператора. После выполнения этого предложения переменная **length** будет содержать число 7.

Следующее предложение выводит на экран значение переменной **length**:

```
cout << length; // здесь выводится 7
```

Если вы хотите вывести на экран значение переменной, просто укажите имя этой переменной справа от знака **<<** в предложении **cout**. В нашем конкретном случае, поскольку **length** содержит число 7, именно это число будет выведено на экран. Перед тем, как приступить к изучению следующего раздела, попробуйте присвоить **length** другие значения и проконтролируйте результат.

Цель

1.6. Использование операторов

Как и большинство других компьютерных языков, C++ поддерживает весь диапазон арифметических операторов, которые позволяют вам манипулировать числовыми значениями, используемыми в программе. К арифметическим операторам относятся:

- + Сложение
- − Вычитание
- * Умножение
- / Деление

Эти операторы используются в C++ в точности так же, как и в обычной алгебре.

В приведенной ниже программе оператор ***** используется для вычисления площади прямоугольника по данным длине и ширине.

```
// Использование оператора.
```

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

```
int main()
{
    int length; // объявление переменной
    int width;  // объявление другой переменной
    int area;   // объявление третьей переменной
```

```
    length = 7; // здесь length присваивается значение 7
```

```
width = 5; // здесь width присваивается значение 5

area = length * width; // вычисление площади
cout << "Площадь равна ";
cout << area; // здесь выводится 35

return 0;
}
```

Присвоим произведение **length** и **width** переменной **area**.

В программе объявляются три переменные: **length**, **width** и **area**. Переменной **length** присваивается значение 7, а переменной **width** — значение 5. Далее программа вычисляет произведение и присваивает полученное значение переменной **area**. Вывод программы выглядит следующим образом:

Площадь равна 35

В нашей программе в действительности не было необходимости вводить переменную **area**. Программу можно было составить следующим образом:

```
// Упрощенный вариант программы для вычисления площади.

#include <iostream>
using namespace std;

int main()
{
    int length; // объявление переменной
    int width; // объявление другой переменной

    length = 7; // здесь length присваивается значение 7
    width = 5; // здесь width присваивается значение 5

    cout << "Площадь равна ";
    cout << length*width; // здесь выводится 35

    return 0;
}
```

Непосредственный вывод **length * width**.

В этом варианте программы площадь вычисляется в предложении **cout** путем умножения **length** на **width**. Полученный результат выводится на экран.

Еще одно замечание перед тем, как мы двинемся дальше: в одном предложении объявления можно объявить две или несколько пере-

менных. Просто разделите их имена запятыми. Например, переменные **length**, **width** и **area** можно было объявить таким образом:

```
int length, width, area; // все переменные объявляются в
                        // одном предложении
```

Объявление нескольких переменных в одном предложении является весьма распространенной практикой в профессионально написанных C++-программах.

Минутная тренировка

1. Должна ли переменная быть объявлена перед ее использованием?
 2. Покажите, как присвоить переменной **min** значение 0.
 3. Можно ли в одном предложении объявления объявить более одной переменной?
- 1 Да, в C++ переменные должны быть объявлены перед их использованием
 - 2 `min = 0;`
 - 3 Да, в одном предложении объявления можно объявить две или любое число переменных
-

Цель

1.7. Ввод с клавиатуры

В предыдущих примерах данные, над которыми выполнялись действия, определялись в программах явным образом. Например, приведенная выше программа определения площади вычисляет площадь прямоугольника размером 7 на 5, причем эти размеры являются частью самой программы. Разумеется, алгоритм вычисления площади прямоугольника не зависит от его размеров, и наша программа была бы значительно полезнее, если бы она запрашивала у пользователя размеры прямоугольника, а пользователь вводил бы их с клавиатуры.

Для того, чтобы дать пользователю возможность ввода данных в программу с клавиатуры, вы будете использовать оператор `>>`. Это оператор C++ ввода. Чтобы прочитать данное с клавиатуры, используйте предложение такого вида:

```
cin >> переменная;
```

Здесь **cin** представляет собой еще один предопределенный идентификатор, автоматически предоставляемый C++. **cin** — это сокращение от *console input* (консольный ввод); по умолчанию **cin** связан с клавиатурой, хотя его можно перенаправить и на другие устройства. Вводимое данное поступает в указанную в предложении переменную.

Ниже приводится программа вычисления площади, которая позволяет пользователю вводить размеры прямоугольника:

```
/*
    Интерактивная программа, вычисляющая
    площадь прямоугольника.
*/

#include <iostream>
using namespace std;

int main()
{
    int length; // объявление переменной
    int width;  // объявление другой переменной

    cout << "Введите длину: ";           ← Ввод с клавиатуры значения
    cin >> length; // ввод длины           переменной length

    cout << "Введите ширину: ";           ← Ввод с клавиатуры значения
    cin >> width;  // ввод ширины          переменной width

    cout << "Площадь равна ";
    cout << length * width; // вывод площади

    return 0;
}
```

Ниже приведен пример работы программы:

```
Введите длину: 8
Введите ширину: 3
Площадь равна 24
```

Обратите особое внимание на эти строки:

```
cout << "Введите длину: ";
cin >> length; // ввод длины
```

Предложение **cout** запрашивает у пользователя ввод данных. Предложение **cin** считывает ввод пользователя, сохраняя значение в **length**. Таким образом, значение, введенное пользователем (в данном случае оно должно быть целым числом) помещается в переменную (в данном случае **length**), указанную справа от оператора **>>**. В результате после выполнения предложения **cin** переменная **length** будет содержать длину прямоугольника. (Если пользователь вводит

нечисловое данное, **length** будет присвоено значение 0.) Предложения, запрашивающие и читающие ширину, действуют в точности так же.

Некоторые дополнительные возможности вывода

До сих пор мы использовали лишь простейшие виды предложений **cout**. Однако **cout** позволяет формировать значительно более сложные предложения вывода. Вот два полезных усовершенствования. Во-первых, вы можете с помощью одного предложения **cout** выводить более одной порции информации. Например, в программе вычисления площади для вывода на экран используются две строки:

```
cout << "Площадь равна ";  
cout << length * width; // вывод площади
```

Эти два предложения можно написать более компактно:

```
cout << "Площадь равна " << length * width;
```

В такой редакции в одном предложении **cout** используются два оператора вывода. В данном случае на экран выводится строка "Площадь равна ", а за ней значение площади. Вообще же вы можете в одно предложение вывода включать сколь угодно длинные цепочки операций вывода. Просто используйте отдельный знак << для каждого выводимого элемента.

Вторая возможность, предоставляемая предложениями **cout**, заключается в выводе нескольких строк данных. До сих пор нам не нужно было переходить на следующую строку экрана (другими словами, выполнять последовательность возврат каретки — перевод строки). Однако необходимость в такой операции возникнет очень скоро. В C++ последовательность возврат каретки — перевод строки генерируется с помощью *символа новой строки*. Для того, чтобы включить в строку этот символ, используйте комбинацию знаков \n (обратная косая черта, за которой стоит строчная буква n). Чтобы посмотреть, как действует комбинация \n, рассмотрите следующую программу:

```
/*
```

```
Эта программа демонстрирует действие кода \n, который  
осуществляет переход в начало следующей строки.
```

```
*/
```

```
include <iostream>
```

```
using namespace std;
int main()
{
    cout << "один\n";
    cout << "два\n";
    cout << "три";
    cout << "четыре";

    return 0;
}
```

Эта программы выведет на экран следующее:

```
один
два
тричетыре
```

Символ новой строки может быть помещен в любое место выводимой строки, не обязательно в ее конце. Поэкспериментируйте с ним, чтобы досконально разобраться, как он влияет на форму вывода.

Минутная тренировка

1. Как выглядит оператор ввода C++?
2. С каким устройством связан **cin** по умолчанию?
3. Что обозначает комбинация **\n**?

- 1 Оператором ввода является **>>**
 - 2 **cin** по умолчанию связан с клавиатурой
 3. Комбинация **\n** обозначает символ новой строки
-

Еще один тип данных

В рассмотренных выше программах использовались переменные типа **int**. Однако такие переменные могут содержать только целые числа. В результате их нельзя использовать в тех случаях, когда число может быть дробным. Например, переменная типа **int** может содержать значение 18, но не 18 и три десятых, что записывается в программах как 18.3. К счастью, кроме **int**, в C++ определен еще целый ряд различных типов данных. Для тех чисел, которые имеют дробную часть, в C++ определено два типа данных с плавающей точкой: **float** и **double**, представляющие значения с одинарной и двойной точностью соответственно. Из этих двух типов **double** используется, пожалуй, чаще.

Чтобы объявить переменную типа **double**, используйте предложение вроде следующего:

```
double result;
```

Здесь **result** — это имя переменной, которая имеет тип **double**. Поскольку переменная **result** принадлежит типу с плавающей точкой, она может содержать такие значения, как 88.56, 0.034 или -107.03.

Чтобы лучше разобраться в том, чем **double** отличается от **int**, рассмотрите следующую программу:

```
/*
    Эта программа иллюстрирует различия
    типов int and double.
*/

#include <iostream>
using namespace std;

int main() {
    int ivar; // объявление переменной типа int
    double dvar; // объявление переменной с плавающей точкой

    ivar = 100; // присвоить ivar значение 100

    dvar = 100.0; // присвоить dvar значение 100.0

    cout << "Исходное значение ivar: " << ivar << "\n";
    cout << " Исходное значение dvar: " << dvar << "\n";

    cout << "\n"; // вывести пустую строку
    // теперь разделим обе переменные на 3
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    cout << "ivar после деления: " << ivar << "\n";
    cout << "dvar после деления: " << dvar << "\n";
    return 0;
}
```

← Вывод на экран
пустой строки

Вывод этой программы показан ниже:

```
Исходное значение ivar: 100
Исходное значение dvar: 100
```



```
ivar после деления: 33  
dvar после деления: 33.3333
```

Когда на 3 делится **ivar**, выполняется целочисленное деление и результат оказывается равным 33 — дробная часть результата теряется. Однако, когда на 3 делится **dvar**, дробная часть сохраняется.

В программе использован еще один новый прием. Обратите внимание на эту строку:

```
cout << "\n"; // вывести пустую строку
```

Это предложение выводит на экран пустую строку. Используйте его, если вы хотите разделить части вывода пустыми строками.

Спросим у эксперта

Вопрос: Зачем в C++ предусмотрены различные типы данных для целых чисел и значений с плавающей точкой? Почему не сделать все числа одного типа?

Ответ: C++ предоставляет различные типы данных, чтобы вы могли писать более эффективные программы. Целочисленная арифметика работает быстрее вычислений с плавающей точкой. И если вам в программе не нужны дробные числа, то нет смысла идти на издержки, связанные с использованием типов **float** и **double**. Кроме того, объем памяти, требуемый для размещения данных одного типа, может быть меньше, чем для данных другого типа. Предоставляя различные типы данных, C++ дает вам возможность наилучшим образом использовать системные ресурсы. Наконец, некоторые алгоритмы требуют использования (или по крайней мере выигрывают от использования) определенных типов данных. Набор встроенных типов данных C++ обеспечивает максимальную гибкость при разработке программ.

Проект 1-1

Преобразование футов в метры

Хотя рассмотренные выше программы и иллюстрируют некоторые важные свойства языка C++, в практическом плане они вряд ли полезны. Однако, несмотря на небольшой объем сведений, полученный вами к этому моменту, вы уже можете написать программу, имеющую практический смысл. В данном проекте мы создадим программу, которая будет преобразовывать футы в метры. Программа запрашивает у

пользователя длину в футах и выводит на экран значение этой длины, преобразованное в метры.

Метр приблизительно эквивалентен 3,28 фута. Таким образом, нам следует использовать данные с плавающей точкой. Для выполнения преобразования программа объявляет две переменные типа **double**. В одной будет содержаться число футов, во второй – результат преобразования в метры.

Шаг за шагом

1. Создайте новый файл C++, назвав его **FtoM.cpp**. (Как уже отмечалось, имена файлов можно выбирать произвольно, так что при желании вы можете использовать и другое имя.)

2. Начните писать программу со строк, которые объясняют назначение программы, включите заголовок `<iostream>` и задайте пространство имен `std`:

```
/*  
    Проект 1-1  
  
    Эта программа преобразует футы в метры  
  
    Назовите эту программу FtoM.cpp.  
*/  
  
#include <iostream>  
using namespace std;
```

3. Начните функцию `main()` с объявления переменных `f` и `m`:

```
int main() {  
    double f; // содержит длину в футах  
    double m; // содержит результат преобразования в метры
```

4. Добавьте код, который введет число футов:

```
cout << "Введите длину в футах: ";  
cin >> f; // читает число футов
```

5. Добавьте код, который выполнит преобразование и выведет результат:

```
m = f / 3.28; // преобразование в метры  
cout << f << " футов составляет " << m << " метров.";
```

6. Завершите программу, как это показано ниже:

```
    return 0;  
}
```

7. Законченная программа должна выглядеть таким образом:.

```
/*  
    Проект 1-1  
  
    Эта программа преобразует футы в метры  
  
    Назовите эту программу FtoM.cpp.  
*/  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    double f; // содержит длину в футах  
    double m; // содержит результат преобразования в метры  
  
    cout << "Введите длину в футах: ";  
    cin >> f; // читает число футов  
  
    m = f / 3.28; // преобразование в метры  
    cout << f << " футов составляет " << m << " метров."  
  
    return 0;  
}
```

8. Откомпилируйте и запустите программу. Пример ее вывода:

```
Введите длину в футах: 5  
5 футов составляет 1.52439 метров.
```

9. Попробуйте ввести другие значения. После этого измените программу так, чтобы она преобразовывала метры в футы.

Минутная тренировка

1. Каково ключевое слово C++ для целочисленного типа данных?
2. Что такое **double**?
3. Как вывести на экран пустую строку?

- 1 Для целочисленных данных предусмотрен тип **int**
 - 2 **double** – это ключевое слово для типа данных с плавающей точкой и с двойной точностью
 - 3 Для вывода на экран пустой строки воспользуйтесь `\n`.
-

Цель

1.8. Два управляющих предложения

Каждая функция выполняется предложение за предложением, от начала к концу. Однако с помощью различных управляющих предложений, включенных в язык C++, можно изменить этот поток выполнения. Позже мы рассмотрим управляющие предложения детально, здесь же кратко остановимся только на двух таких предложениях, так как мы будем использовать их в дальнейших примерах программ.

Предложение if

С помощью условного предложения языка C++ **if** можно селективно выполнять определенный участок программы. Предложение **if** языка C++ действует так же, как и в любом другом языке. Оно, в частности, синтаксически идентично предложениям **if** языков C, Java и C#. Его простейшая форма выглядит следующим образом:

if(условие) предложение;

Здесь *условие* представляет собой выражение, результат которого воспринимается либо как истина (**true**), либо как ложь (**false**). В C++ истина соответствует ненулевому значению, а ложь — нулевому. Если условие истинно, предложение будет выполняться. Если условие ложно, предложение не выполняется. Например, следующий фрагмент выводит на экран фразу **10 меньше 11**, потому что 10 действительно меньше 11.

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

Рассмотрим, однако, следующее предложение:

```
if(10 > 11) cout << "эта строка не выводится";
```

В этом случае, поскольку 10 не больше 11, предложение **cout** выполняться не будет. Разумеется, операнды внутри предложения **if** не обязательно должны быть константами. В качестве операндов можно использовать и переменные.

В C++ определен полный набор операторов отношения, которые используются в условных выражениях. Эти операторы перечислены ниже:

| Оператор | Значение |
|----------|----------------------|
| < | Меньше чем |
| <= | Меньше чем или равно |
| > | Больше чем |
| >= | Больше чем или равно |
| == | Равно |
| != | Не равно |

Обратите внимание на то, что проверка на равенство требует использования двойного знака равенства.

Ниже приводится программа, иллюстрирующая использование предложения **if**:

// Демонстрация использования **if**.

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

```
int main() {
    int a, b, c;
```

```
    a = 2;
    b = 3;
```

```
    if(a < b) cout << "a меньше чем b\n";
```

← Предложение **if**

```
    // это предложение ничего не выведет на экран
    if(a == b) cout << "это вы не увидите\n";
```

```
    cout << "\n";
```

```
    c = a - b; // c содержит -1
    cout << "c содержит -1\n";
    if(c >= 0) cout << "c неотрицательно\n";
    if(c < 0) cout << "c отрицательно\n";
```

```
    cout << "\n";
```

```
    c = b - a; // c теперь содержит 1
    cout << "c содержит 1\n";
    if(c >= 0) cout << "c неотрицательно\n";
    if(c < 0) cout << "c отрицательно\n";
```

```
    return 0;
}
```

Ниже приведен вывод этой программы:

```
а меньше чем b

с содержит -1
с отрицательно

с содержит 1
с неотрицательно
```

Цикл for

С помощью конструкции *цикла* вы можете повторно выполнять некоторую последовательность кодов. В C++ предусмотрен богатый ассортимент конструкций циклов. Здесь мы рассмотрим только цикл **for**. Если вы знакомы с C# или Java, вам будет приятно узнать, что цикл **for** действует в C++ точно так же, как и в этих двух языках. Ниже приведена простейшая форма цикла **for**:

for(инициализация; условие; приращение) предложение;

В этой конструкции *инициализация* устанавливает начальное значение управляющей переменной. *условие* является выражением, которое анализируется в каждом шаге цикла. *приращение* представляет собой выражение, определяющее, как будет изменяться значение управляющей переменной в каждом шаге цикла.

Приводимая ниже программа демонстрирует простой цикл **for**. Она выводит на экран числа от 1 до 100.

// Программа, иллюстрирующая цикл for.

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

```
int main()
```

```
{
```

```
    int count;
```

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

```
        cout << count << " ";
```

```
    return 0;
```

```
}
```

← Это цикл **for**

В этой программе переменная **count** инициализируется числом 1. В каждом шаге цикла проверяется условие

```
count <= 100
```

Если результат истинен, значение **count** выводится на экран, после чего **count** увеличивается на 1. Когда **count** достигает значения, большего 100, условие становится ложным и цикл прекращает свое выполнение.

В профессионально написанной C++-программе вы почти никогда не встретите предложение вроде

```
count=count+1
```

потому что C++ включает в себя специальный инкрементный оператор, выполняющий эту операцию более эффективно. Инкрементный оператор записывается как **++** (два последовательных знака плюс). Этот оператор увеличивает операнд на 1. С его помощью предыдущее предложение **for** запишется таким образом:

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

Именно такая форма будет использоваться в дальнейшем в этой книге.

В C++ имеется и декрементный оператор, который записывается как **--**. Он уменьшает операнд на 1.

Минутная тренировка

1. Зачем служит предложение **if**?
 2. Зачем служит предложение **for**?
 3. Назовите операторы отношения C++.
 1. В C++ предложение **if** является условным.
 2. Предложение **for** образует одну из форм цикла в C++.
 3. В C++ имеются следующие операторы отношения: **==**, **!=**, **<**, **>**, **<=** и **>=**
-

Цель

1.9. Использование программных блоков

Еще одним ключевым элементом C++ является *программный блок*. Программный блок представляет собой группу из двух или более предложений. Блок образуется заключением его предложений

в фигурные скобки. Блок, будучи создан, становится логической единицей, которую можно использовать в любом месте, где может находиться одиночное предложение. В частности, блок может служить целевым объектом (мишенью) в предложениях `if` и `for`. Рассмотрим пример предложения `if`:

```
if (w < h) {  
    v = w * h;  
    w = 0;  
}
```

В этом фрагменте если `w` меньше чем `h`, выполняются оба предложения внутри блока. Таким образом, эти два предложения внутри блока образуют логическую единицу, в которой одно предложение не может выполняться без второго. Во всех случаях, когда вам требуется логически связать два или несколько предложений, вы заключаете их в блок. С использованием программных блоков многие алгоритмы реализуются с большей ясностью и эффективностью.

Рассмотрим программу, которая использует программный блок для предотвращения деления на 0:

// Демонстрация программного блока.

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

```
int main() {  
    double result, n, d;
```

```
    cout << "Введите значение: ";  
    cin >> n;
```

```
    cout << "Введите делитель: ";  
    cin >> d;
```

Мишенью этого предложения `if` является весь блок

```
// целевым объектом предложения if является блок
```

```
if (d != 0) {  
    cout << "d не равно нулю, поэтому делить можно" << "\n";  
    result = n / d;  
    cout << n << " / " << d << " равно" << result;  
}
```

```
    return 0;
```

```
}
```


Ниже приведен пример прогона программы:

```
Введите значение: 10
Введите делитель: 2
d не равно нулю, поэтому делить можно
10 / 2 равно 5
```

В этом случае мишенью предложения `if` является программный блок, а не просто одиночное предложение. Если проверяемое в предложении `if` условие истинно (как это имеет место в примере прогона программы), то выполняются все три предложения, заключенные в блок. Попробуйте задать в качестве делителя ноль и посмотрите на результат. В этом случае весь код внутри блока не выполняется.

Спросим у эксперта

Вопрос: Приводит ли использование программного блока к потере эффективности по времени? Другими словами, потребляют ли скобки `{` и `}` дополнительное время при выполнении программы?

Ответ: Нет. Программный блок не добавляет никаких издержек. Наоборот, благодаря тому, что программные блоки способствуют упрощению некоторых алгоритмов, их использование обычно увеличивает скорость выполнения и эффективность.

Как будет показано далее, программные блоки имеют дополнительные свойства и области использования. Однако основное их назначение заключается в создании логически неразрывных единиц кода.

Знак точки с запятой и позиционирование

В C++ знак точки с запятой обозначает конец предложения. Другими словами, каждое предложение должно заканчиваться точкой с запятой. Как вы знаете, блок представляет собой последовательность логически связанных предложений, окруженных открывающей и закрывающей фигурной скобкой. Блок *не* заканчивается знаком точки с запятой. Поскольку в блок входит группа предложений, и каждое из них заканчивается точкой с запятой, то вполне логично, что весь блок не имеет в конце точки с запятой; на конец блока указывает закрывающая фигурная скобка.

C++ не рассматривает конец строки текста как конец предложения — только точка с запятой заканчивает предложение. По этой при-

чине не имеет значения, в каком месте строки вы размещаете предложение. Например, для C++

```
x = y;  
y = y + 1;  
cout << x << " " << y;
```

будет значить то же самое, что

```
x = y; y = y + 1; cout << x << " " << y;
```

Более того, отдельные элементы предложения тоже могут располагаться на отдельных строках. Например, приведенный ниже фрагмент вполне правилен:

```
cout << "Это длинное предложение. Сумма равна : " << a +  
b + c + d + e + f;
```

Разделение длинной строки на несколько более коротких часто используется для повышения наглядности программы. Кроме того, умышленное деление на строки помогает избежать автоматического перехода на следующую строку в неудачном месте.

Практика использования отступов

Вы могли заметить, что в рассмотренных программах некоторые предложения писались с отступом. C++ относится к языкам свободной формы, что означает возможность располагать предложения в любом месте строки. Однако с течением лет в практике программистов утвердился определенный стиль использования отступов, существенно способствующий ясности и удобочитаемости текста программы. В настоящей книге мы придерживаемся этого стиля и рекомендуем вам делать то же. Согласно общепринятому стилю вы отступаете вправо на один уровень после каждой открывающей фигурной скобки и перемещаетесь назад на один уровень после каждой закрывающей фигурной скобки. Есть, однако, определенные предложения, в которые удобно вводить дополнительные отступы; мы рассмотрим их позже.

Минутная тренировка

1. Как создается программный блок? Для чего он предназначен?
2. В C++ предложения заканчиваются _____.
3. Все предложения C++ должны начинаться и заканчиваться на одной строке. Правильно ли это?

1. Блок начинается открывающей фигурной скобкой ({) и заканчивается закрывающей фигурной скобкой (}). Блок служит для создания логической единицы кода.
2. Точкой с запятой
3. Неправильно.

Проект 1-2 Создание таблицы преобразования футов в метры

Этот проект демонстрирует использование цикла **for**, предложения **if**, а также программных блоков при разработке программы, выводящей на экран таблицу преобразования футов в метры. Таблица начинается с 1 фута и заканчивается на 100 футах. После каждых 10 футов выводится пустая строка. С этой целью в программе используется переменная с именем **counter**, которая отсчитывает выводимые строки. Обратите особое внимание на эту переменную.

Шаг за шагом

1. Создайте новый файл с именем **FtoMTable.cpp**.
2. Введите следующую программу:

```
/*
    Проект 1-2

    Эта программы выводит на экран таблицу преобразования
    футов в метры.

    Назовите эту программу FtoMTable.cpp.
*/

#include <iostream>
using namespace std;

int main() {
    double f; // содержит длину в футах
    double m; // содержит результат преобразования в метры
    int counter;

    counter = 0; ← Счетчик строк первоначально
                  устанавливается в 0.
```

```
for(f = 1.0; f <= 100.0; f++) {  
    m = f / 3.28; // преобразуем в метры  
    cout << f << " футов составляет " << m << " метров.\n";  
    counter++; ← Инкремент счетчика строк в  
                ← каждом шаге цикла  
    // после каждых 10 строк вывести пустую строку  
    if(counter == 10) { ← Если counter рав-  
        cout << "\n"; // вывод пустой строки      но 10, вывести  
        counter = 0; // сброс счетчика строк        пустую строку.  
    }  
}  
  
return 0;  
}
```

3. Обратите внимание на использование переменной **counter** для вывода пустой строки после каждых десяти строк. Первоначально эта переменная устанавливается в 0 вне цикла **for**. Внутри цикла она инкрементируется после каждого преобразования. Когда значение **counter** достигает 10, выводится пустая строка, **counter** сбрасывается в 0 и процесс продолжается.

4. Откомпилируйте и запустите программу. Ниже приведена часть ее вывода. Обратите внимание на форму вывода чисел. В тех случаях, когда число оказывается дробным, оно выводится с десятичной дробной частью.

```
1 футов составляет 0.304878 метров.  
2 футов составляет 0.609756 метров.  
3 футов составляет 0.914634 метров.  
4 футов составляет 1.21951 метров.  
5 футов составляет 1.52439 метров.  
6 футов составляет 1.82927 метров.  
7 футов составляет 2.13415 метров.  
8 футов составляет 2.43902 метров.  
9 футов составляет 2.7439 метров.  
10 футов составляет 3.04878 метров.  
  
11 футов составляет 3.35366 метров.  
12 футов составляет 3.65854 метров.  
13 футов составляет 3.96341 метров.  
14 футов составляет 4.26829 метров.  
15 футов составляет 4.57317 метров.  
16 футов составляет 4.87805 метров.  
17 футов составляет 5.18293 метров.  
18 футов составляет 5.4878 метров.  
19 футов составляет 5.79268 метров.  
20 футов составляет 6.09756 метров.
```

```
21 футов составляет 6.40244 метров.  
22 футов составляет 6.70732 метров.  
23 футов составляет 7.0122 метров.  
24 футов составляет 7.31707 метров.  
25 футов составляет 7.62195 метров.  
26 футов составляет 7.92683 метров.  
27 футов составляет 8.23171 метров.  
28 футов составляет 8.53659 метров.  
29 футов составляет 8.84146 метров.  
30 футов составляет 9.14634 метров.  
  
31 футов составляет 9.45122 метров.  
32 футов составляет 9.7561 метров.  
33 футов составляет 10.061 метров.  
34 футов составляет 10.3659 метров.  
35 футов составляет 10.6707 метров.  
36 футов составляет 10.9756 метров.  
37 футов составляет 11.2805 метров.  
38 футов составляет 11.5854 метров.  
39 футов составляет 11.8902 метров.  
40 футов составляет 12.1951 метров.
```

5. Попробуйте самостоятельно изменить программу так, чтобы она выводила пустую строку после каждый 25 строк вывода.

Цель

1.10. Знакомимся с функциями

C++-программа конструируется из строительных блоков, называемых *функциями*. В Модуле 5 мы обсудим функции более детально, здесь же ограничимся кратким обзором. Начнем с определения термина *функция*: функцией называется подпрограмма, содержащая одно или несколько предложений C++.

У каждой функции есть имя, и это имя используется для вызова функции. Для того, чтобы вызвать функцию, просто укажите в исходном тексте вашей программы ее имя, сопровождаемой парой круглых скобок. Предположим, что некоторая функция названа **MyFunc**. Для того, чтобы вызвать **MyFunc**, вы должны написать:

```
MyFunc ( ) ;
```

Вызов функции приводит к тому, что управление передается этой функции, в результате чего выполняется код, содержащийся внутри функции. После того, как выполняются все предложения, составляю-

шие функцию, управление возвращается назад в вызывающую программу. Таким образом, функция выполняет некоторую задачу, в которой возникла необходимость по ходу программы.

Некоторые функции требуют указания одного или нескольких *аргументов*, которые передаются функции при ее вызове. Аргумент представляет собой значение, передаваемое функции. Перечень аргументов указывается внутри круглых скобок в предложении вызова функции. Если, например, функция **MyFunc()** требует в качестве аргумента целое число, то приведенная ниже строка вызовет **MyFunc()** с передачей ей в качестве аргумента числа 2:

```
MyFunc(2);
```

Если аргументов два или больше, они разделяются запятыми. В этой книге термин *список аргументов* будет относиться к перечню аргументов, разделяемых запятыми. Учтите, что не всем функциям требуются аргументы. Если аргументы не нужны, при вызове функции скобки остаются пустыми.

Функция может возвращать значение в вызывающий ее код. Не все функции возвращают значение, хотя многие это делают. Значение, возвращаемое функцией, может быть присвоено переменной в вызывающем коде путем помещения вызова функции с правой стороны предложения присваивания. Если, например, функция **MyFunc()** возвращает значение, ее можно вызвать таким образом:

```
x = MyFunc(2);
```

Это предложение выполняется следующим образом. Прежде всего вызывается **MyFunc()**. Когда осуществляется возврат из функции, возвращаемое ею значение присваивается переменной *x*. Допустимо также использовать вызов функции в выражении. Например,

```
x = MyFunc(2) + 10;
```

В этом случае возвращаемое функцией **MyFunc()** значение прибавляется к 10, и результат присваивается переменной *x*. Вообще всегда, когда имя функции встречается в предложении, она вызывается автоматически, что дает возможность получить возвращаемое ею значение.

Повторим: *аргументом* называется значение, передаваемое в функцию. *Возвращаемое значение* — это данное, которое передается назад в вызывающий код.

Ниже приведена короткая программа, демонстрирующая вызов функции. Она использует одну из встроенных в C++ функций, которая называется **abs()**, чтобы вывести на экран абсолютное значение

числа. Функция **abs()** принимает один аргумент, преобразует его в абсолютное значение и возвращает результат.

```
// Использование функции abs( ).
```

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

```
int main()
```

```
{
```

```
    int result;
```

```
    result = abs(-10);
```

```
    cout << result;
```

```
    return 0;
```

```
}
```

Здесь вызывается функция **abs()** и выполняется присваивание возвращаемого ею значения переменной **result**

В приведенной программе значение -10 передается функции **abs()** в качестве аргумента. Функция **abs()** получает аргумент, с которым она вызывается, и возвращает его абсолютное значение, которое в данном случае составляет 10. Это значение присваивается переменной **result**. В результате программа выводит на экран “10”.

Обратите внимание на включение в программу заголовка **<cstdlib>**. Этот заголовок требуется для функции **abs()**. Во всех случаях, если вы используете библиотечную функцию, вы должны включать заголовок, в котором она описана.

Вообще говоря, вы будете использовать в своих программах функции двух видов. Один вид функций — это те функции, текст которых вы пишете сами; примером такой функции является **main()**. Позже вы научитесь писать и другие прикладные функции. Реальные C++-программы всегда содержат большое число функций, написанных пользователем.

Функции второго вида предоставляются компилятором. Примером такой функции является **abs()**, использованная в предыдущей программе. Программы, которые вам придется писать, всегда будут содержать как функции, созданные вами, так и функции, предоставленные компилятором.

Обозначая функции в тексте этой книги, мы пользовались и будем пользоваться впредь соглашением, общепринятым в среде программистов на C++. После имени функции ставятся круглые скобки. Если, например, функция имеет имя **getval**, то при использовании этого имени в тексте книги мы будем писать **getval()**. Такой прием поможет вам отличать имена переменных от имен функций.

Библиотеки C++

Как только что было сказано, функция `abs()` поставляется вместе с вашим компилятором C++. Эта функция, как и много других, хранится в *стандартной библиотеке*. Мы на протяжении всей книги будем использовать в примерах программ различные библиотечные функции.

В C++ определен большой набор функций, содержащихся в стандартной библиотеке функций. Эти функции выполняют различные задачи общего назначения, включая операции ввода-вывода, математические вычисления и обработку строк. Если вы используете библиотечную функцию, компилятор C++ автоматически связывает объектный код этой функции с объектным кодом вашей программы.

Стандартная библиотека C++ очень велика; она содержит большое количество функций, которые наверняка потребуются вам при написании программ. Библиотечные функции действуют как строительные блоки, из которых вы собираете нужную вам конструкцию. Вам следует познакомиться с документацией к библиотеке вашего компилятора. Вас может удивить, насколько разнообразны библиотечные функции. Если вы напишете функцию, которую будете использовать снова и снова, ее также можно сохранить в библиотеке.

В дополнение к стандартной библиотеке функций, каждый компилятор C++ содержит *библиотеку классов*, являющуюся объектно-ориентированной библиотекой. Однако, чтобы начать использовать эту библиотеку, вам придется обождать, пока вы познакомитесь с классами и объектами.

Минутная тренировка

1. Что такое функция?
 2. Функция вызывается указанием ее имени. Правильно ли это?
 3. Что представляет собой стандартная библиотека функций C++?
- 1 Функция — это подпрограмма, содержащая одно или несколько предложений C++
 - 2 Правильно
 - 3 Стандартная библиотека C++ представляет собой собрание функций, предоставляемых всеми компиляторами C++.
-

Цель

1.11. Ключевые слова C++

К настоящему времени в стандартном C++ определены 63 ключевых слова; все они приведены в табл. 1-1. Вместе с формальным синтаксисом C++ они образуют язык программирования C++. Ранние версии

C++ определяли еще ключевое слово **overload**, но теперь оно устарело. Имейте в виду, что язык C++ различает прописные и строчные буквы, при этом он требует, чтобы все ключевые слова вводились строчными буквами (на нижнем регистре клавиатуры).

Таблица 1-1. Ключевые слова C++

| | | | |
|-------------|------------|------------------|--------------|
| asm | auto | bool | break |
| case | catch | char | class |
| const | const_cast | continue | default |
| delete | do | double | dynamic_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 | |

Цель

1.12. Идентификаторы

В C++ к идентификаторам относятся имена, присвоенные функциям, переменным или любым другим определенным пользователем объектам. Идентификатор может состоять из одного или нескольких символов. Имена переменных допустимо начинать с любой буквы латинского алфавита или со знака подчеркивания. Далее может идти буква, цифра или знак подчеркивания. Знаки подчеркивания обычно используются для повышения наглядности имен переменных, например, **line_count** (счетчик строк). Прописные и строчные буквы рассматриваются компилятором, как различные символы; таким образом, для C++ **myvar** и **MyVar** являются различающимися именами. При назначении имен переменным или функциям следует иметь в виду важное ограничение: ключевые слова C++ нельзя использовать в качестве идентификаторов. Кроме того, нельзя использовать в произвольных целях predefined идентификаторы, такие, как **cout**.

Вот несколько примеров правильных идентификаторов:

| Test | x | y2 | MaxIncr |
|------|------|--------|------------------|
| up | _top | my_var | simpleInterest23 |

Не забывайте, что недопустимо начинать идентификатор с цифры. Так, **9800K** не может служить идентификатором. Хороший стиль программирования требует, чтобы ваши идентификаторы отражали существо обозначаемых ими объектов.

Минутная тренировка

1. Что является ключевым словом, **for**, **For** или **FOR**?
 2. Какие символы может содержать идентификатор C++?
 3. Являются имена **index21** и **Index21** одним и тем же идентификатором?
 - 1 Ключевым словом является **for**. В C++ все ключевые слова состоят из строчных букв
 - 2 Идентификатор C++ может содержать буквы латинского алфавита, цифры и знак подчеркивания
 - 3 Нет, C++ различает символы нижнего и верхнего регистров
-

✓ Вопросы для самопроверки

1. Говорят, что C++ находится в центре современного мира программирования. Поясните это утверждение.
2. Компилятор C++ создает объектный код, который может быть непосредственно выполнен компьютером. Справедливо ли это утверждение?
3. Каковы три основных принципа объектно-ориентированного программирования?
4. В какой точке C++-программы начинается ее выполнение?
5. Что такое заголовок?
6. Что такое `<iostream>`? Каково назначение следующей программной строки?

```
#include <iostream>
```

7. Что такое пространство имен?
8. Что такое переменная?
9. Какое (или какие) из приведенных ниже имен переменных неправильны?
 - a. count
 - b. _count

c. count27

d. 67count

e. if

10. Каким образом в программу вводится однострочный комментарий?
А многострочный?
11. Приведите обобщенную форму предложения `if`. Выполните то же для цикла `for`.
12. Каким образом в программе задается программный блок?
13. Сила тяжести на Луне составляет приблизительно 17% от земной. Напишите программу, которая выведет на экран таблицу перевода земных фунтов в их лунные эквиваленты (1 фунт равен 453 граммам, хотя в данном случае это не имеет значения). Вычислите 100 значений таблицы от 1 до 100 фунтов. После каждых 25 фунтов выведите пустую строку.
14. Год на Юпитере (т. е. время, требуемое Юпитеру для полного оборота вокруг Солнца) составляет около 12 земных лет. Напишите программу, которая преобразует юпитерианские годы в земные. Предоставьте возможность пользователю вводить с клавиатуры число юпитерианских лет. Обеспечьте возможность ввода дробного числа лет.
15. Как изменяется ход выполнения программы, когда вызывается функция?
16. Напишите программу, которая усредняет абсолютные значения пяти произвольных чисел, вводимых пользователем. Выведите на экран результат.

Модуль 2 Знакомимся с данными, типами и операторами

Цели, достигаемые в этом модуле

- 2.1 Изучить типы данных C++
- 2.2 Научиться использовать литералы
- 2.3 Узнать о создании инициализированных переменных
- 2.4 Рассмотреть подробнее арифметические операторы
- 2.5 Познакомиться с логическими операторами и операторами отношений
- 2.6 Изучить возможности оператора присваивания
- 2.7 Узнать об операциях составных присваиваний
- 2.8 Разобраться в преобразовании типов в операциях присваивания
- 2.9 Разобраться в преобразовании типов в выражениях
- 2.10 Научиться использовать приведение типа
- 2.11 Освоить правила применения пробелов и скобок

Ядром языка программирования являются его типы данных и операторы. Эти элементы определяют границы языка и области его применения. Как и следует ожидать, C++ поддерживает богатый набор типов данных и операторов, что делает его удобным для решения самого широкого круга задач программирования.

Типы данных и операторы представляют собой весьма обширный предмет. Мы начнем здесь с изучения фундаментальных типов данных C++ и его наиболее широко используемых операторов. Мы также рассмотрим более подробно переменные и выражения.

Почему так важны типы данных

Тип, которому принадлежит конкретная переменная, важен потому, что он определяет допустимые операции и диапазон значений, которые может принимать эта переменная. В C++ определено несколько типов данных, и каждый тип имеет вполне определенные характеристики. Поскольку типы данных отличаются друг от друга, все переменные должны быть объявлены перед их использованием, и объявление переменной всегда содержит спецификатор типа. Компилятор требует предоставления ему этой информации для того, чтобы он мог сгенерировать правильный код. В C++ отсутствует концепция “безтиповых” переменных.

Вторая причина важности типов данных заключается в том, что некоторые базовые типы данных тесно привязаны к строительным блокам, которыми оперирует компьютер: байтам и словам. Таким образом, C++ позволяет вам работать с теми же типами данных, что и сам процессор. Этим, в частности, объясняется возможность писать на C++ чрезвычайно эффективные программы системного уровня.

Цель

2.1. Типы данных C++

C++ предоставляет встроенные типы данных, соответствующие целым, символам, значениям с плавающей точкой и булевым переменным. Обычно именно такого рода данные хранятся и обрабатываются программой. Как вы узнаете из дальнейших разделов этой книги, C++ позволяет вам конструировать более сложные типы, к которым относятся классы, структуры и перечислимые типы, однако и эти типы в конечном счете составляются из встроенных типов данных.

Ядром системы типов C++ являются семь базовых типов данных, перечисленных ниже:

| Тип | Значение |
|---------|---|
| char | Символ |
| wchar_t | "Широкий" символ |
| int | Целое |
| float | Число с плавающей точкой |
| double | Число с плавающей точкой удвоенной точности |
| bool | Булева переменная |
| void | Пустой тип |

C++ допускает указание перед некоторыми базовыми типами модификаторов. Модификатор изменяет значение базового типа, более точно приспособляя его к потребностям определенных ситуаций. Модификаторы типов данных перечислены ниже:

signed (со знаком)
 unsigned (без знака)
 long (длинный)
 short (короткий)

Модификаторы **signed**, **unsigned**, **long** и **short** приложимы к типу **int**. Модификаторы **signed** и **unsigned** приложимы к типу **char**. Тип **double** может быть модифицирован описателем **long**. В табл. 2-1 перечислены все допустимые комбинации базовых типов и модификаторов типа. В таблице показаны также гарантированные минимальные диапазоны значений для каждого типа, как это специфицировано стандартом ANSI/ISO C++.

Таблица 2-1. Все числовые типы данных, определенные в C++, и их минимальные гарантированные диапазоны значений согласно спецификации стандарта ANSI/ISO C++

| Тип | Минимальный диапазон |
|--------------------|------------------------------|
| char | От -127 до 127 |
| unsigned char | От 0 до 255 |
| signed char | От -127 до 127 |
| int | От -32767 до 32767 |
| unsigned int | От 0 до 65535 |
| signed int | То же, что int |
| short int | От -32767 до 32767 |
| unsigned short int | От 0 до 65535 |
| signed short int | То же, что short int |
| long int | От -2147483647 до 2147483647 |

| | |
|-------------------|---|
| signed long int | То же, что long int |
| unsigned long int | От 0 до 4294967295 |
| float | От $1E-37$ до $1E+37$, с шестью значащими цифрами |
| double | От $1E-37$ до $1E+37$, с десятью значащими цифрами |
| long double | От $1E-37$ до $1E+37$, с десятью значащими цифрами |

Обратите внимание на то, что минимальные диапазоны, указанные в табл. 2-1, значат именно это: *минимальные* диапазоны. Компилятор C++ вполне может увеличить один или несколько диапазонов допустимых значений, и большинство компиляторов так и поступает. Таким образом, диапазоны допустимых значений данных в C++ зависят от реализации. Например, на компьютерах, выполняющих арифметические операции в дополнительном коде (что характерно для большинства современных компьютеров) диапазон чисел **int** составит по меньшей мере — 32768 ... 32767. Однако во всех случаях диапазон типа **short int** будет поддиапазоном типа **int**, который, в свою очередь, будет поддиапазоном типа **long int**. То же справедливо для типов **float**, **double** и **long double**. В данном случае термин *поддиапазон* обозначает диапазон, меньший или равный исходному. Так, **int** и **long int** могут иметь одинаковые диапазоны, но диапазон **int** не может быть больше, чем у **long int**.

Поскольку C++ определяет лишь минимальные диапазоны, которые должны поддерживаться типами данных, то для того, чтобы определить фактические допустимые диапазоны чисел, вы должны обратиться к документации вашего компилятора. В качестве примера в табл. 2-2 приведены данные о типичной ширине в битах и диапазонах типов данных C++ в 32-разрядной среде, используемой, например, системой Windows XP.

Рассмотрим теперь каждый тип данных более детально.

Таблица 2-2. Типичные ширины в битах и диапазоны типов данных C++ в 32-разрядной среде

| Тип | Ширина, бит | Типичный диапазон |
|--------------------|-------------|------------------------------|
| char | 8 | От -128 до 127 |
| unsigned char | 8 | От 0 до 255 |
| signed char | 8 | От -128 до 127 |
| int | 32 | От -2147483648 до 2147483647 |
| unsigned int | 32 | От 0 до 4294967295 |
| signed int | 32 | От -2147483648 до 2147483647 |
| short int | 16 | От -32768 до 32767 |
| unsigned short int | 16 | От 0 до 65535 |
| signed short int | 16 | От -32768 до 32767 |

| | | |
|-------------------|-------------|--------------------------------|
| long int | 32 | То же, что int |
| signed long int | 32 | То же, что signed int |
| unsigned long int | 32 | То же, что unsigned int |
| float | 32 | От 1.8E-38 до 3.4E+38 |
| double | 64 | От 2.2E-308 до 1.8E+308 |
| long double | 80 | От 1.2E-4392 до 1.2E+4392 |
| bool | Неприменимо | true(истина) или false (ложь) |
| wchar_t | 16 | От 0 до 65535 |

Целые числа

Как вы уже знаете из Модуля 1, переменные типа **int** содержат целые величины, у которых нет дробных частей. Переменные этого типа часто используются для управления циклами и условными предложениями, а также для счета. Операции с величинами типа **int**, поскольку у них отсутствуют дробные части, выполняются значительно быстрее, чем для типов с плавающей точкой.

В силу важности целых чисел для программирования, в C++ для них определен ряд вариантов. Как было показано в табл. 2-1, целые могут быть короткими, обычными и длинными. Далее, для каждого типа имеются варианты со знаком и без знака. Целое со знаком может содержать как положительные, так и отрицательные значения. По умолчанию типы **int** рассматриваются как числа со знаком. Таким образом, использование описателя **signed** является избыточным (хотя и разрешенным), потому что объявление по умолчанию предполагает значение со знаком. Целые без знака могут содержать только положительные значения. Чтобы создать целое без знака, используйте модификатор **unsigned**.

Различие между целыми со знаком и без знака заключается в том, каким образом интерпретируется старший бит числа. Если объявлено целое со знаком, то компилятор C++ при генерации кода предполагает, что старший бит числа используется как *флаг знака*. Если флаг знака равен 0, число положительное; если он равен 1, число отрицательное. Отрицательные числа почти всегда представляются в *дополнительном коде*. Для получения отрицательного числа этим методом все биты числа (за исключением флага знака) инвертируются, и к полученному числу прибавляется 1. После этого флаг знака устанавливается в 1.

Целые со знаком используются во множестве различных алгоритмов, однако для них характерен в два раза меньший диапазон возможных абсолютных значений, чем для их аналогов без знака. Рассмотрим, например, 16-битовое целое, конкретно, число 32767:

```
01111111 11111111
```


Для значения со знаком, если установить в 1 его старший бит, число будет интерпретироваться как -1 (в предположении, что используется метод дополнительного кода). Если, однако, вы объявили это число, как **unsigned int**, то при установке в 1 старшего бита число станет равным 65535.

Для того, чтобы разобраться в различиях интерпретации C++ целых чисел со знаком и без знака, рассмотрим эту простую программу:

```
#include <iostream>

/* Эта программа демонстрирует различие между
   целыми числами со знаком и без знака. */

using namespace std;

int main()
{
    short int i; // короткое целое со знаком
    short unsigned int j; // короткое целое без знака

    j = 60000;
    i = j;
    cout << i << " " << j;

    return 0;
}
```

60000 находится в допустимом диапазоне значений для **unsigned short int**, но, как правило, будет вне допустимого диапазон для чисел **signed short int**. В результате при присваивания его переменной *i* оно будет интерпретироваться как отрицательное значение

Вывод этой программы приведен ниже:

```
-5536 60000
```

Вывод именно этих значений определяется тем, что комбинация битов, представляющая число 60000 как короткое целое без знака, будет интерпретироваться как -5536 , если рассматривать это значение как короткое целое со знаком (в предположении, что числа 16-разрядные).

В C++ предусмотрены сокращенные обозначения для объявления целых чисел **unsigned**, **short** и **long**. Вы можете просто использовать **unsigned**, **short** и **long** без **int**. Тип **int** в этом случае предполагается по умолчанию. Таким образом, оба следующие предложения объявляют целые переменные без знака:

```
unsigned x;
unsigned int y;
```

Символы

Переменные типа **char** содержат 8-битовые коды ASCII символов, например, **A**, **z** или **G**, или любое 8-битовое число. Для того, чтобы задать в программе символ, его следует заключить в одиночные кавычки. В следующем примере переменной **ch** присваивается код символа **X**:

```
char ch;  
ch = 'X';
```

Значение переменной типа **char** можно вывести на экран с помощью предложения **cout**. Так, следующая строка выводит значение переменной **ch**:

```
cout << "Это ch: " << ch;
```

При выполнении этого предложения на экран будет выведено:

```
Это ch: X
```

Тип **char** можно модифицировать с помощью модификаторов **signed** или **unsigned**. Вообще говоря, будет ли тип **char** по умолчанию со знаком или без знака, зависит от реализации. Однако для большинства компиляторов **char** считается типом со знаком. В таких средах использование с **char** модификатора **signed** является избыточным. В этой книге будет предполагаться, что переменные типа **char** — это переменные со знаком.

Тип **char** может содержать значения, отличные от набора символов ASCII. Этот тип можно также использовать как “маленькое” целое с диапазоном значений обычно от -128 до 127 ; такую переменную можно использовать вместо **int** в тех случаях, когда в данной ситуации не нужны большие числа. Например, в приведенной ниже программе переменная типа **char** используется для управления циклом, который выводит на экран символы латинского алфавита:

```
// Эта программа выводит латинский алфавит.
```

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

```
int main()  
{
```

```
    char letter;
```

```
    for(letter = 'A'; letter <= 'Z'; letter++)
```

Использование переменной типа **char** для управления циклом **for**.



```
cout << letter;  
  
return 0;  
}
```

Цикл `for` позволяет решить поставленную нами задачу, так как символ `A` представляется в компьютере значением 65, а значения кодов букв от `A` до `Z` следуют подряд в возрастающем порядке. Поэтому переменной `letter` первоначально присваивается значение 'A'. В каждом шаге цикла `letter` получает приращение 1. В результате после первой итерации переменная `letter` содержит код буквы 'B'.

Тип `wchar_t` содержит символы, входящие в расширенный набор. Как вы, возможно, знаете, многие человеческие языки, например, китайский, используют большое число символов, больше, чем помещаются в 8 бит, предоставляемых типом `char`. Тип `wchar_t` и был добавлен в C++ для поддержки этих языков. Хотя в этой книге мы не будем использовать тип `wchar_t`, вам придется обратиться к нему, если вы будете адаптировать программы для международного рынка.

Спросим у эксперта

Вопрос: Почему в C++ определены лишь минимальные диапазоны для встроенных типов данных, а не их точные границы?

Ответ: Не задавая точных границ, C++ позволяет каждому компилятору оптимизировать типы данных применительно к используемой операционной среде. Отчасти поэтому C++ позволяет создавать высокопроизводительное программное обеспечение. Стандарт ANSI/ISO C++ только устанавливает, что встроенные типы должны удовлетворять определенным требованиям. Например, стандарт требует, чтобы тип `int` "имел естественный размер, задаваемый архитектурой исполнительной среды". Таким образом, в 32-разрядной среде `int` будет иметь размер 32 бит. В 16-разрядной среде длина `int` составит 16 бит. Было бы неэффективно и избыточно заставлять 16-разрядный компилятор реализовывать, например, `int` с 32-битовым диапазоном значений. Идеология C++ позволяет избежать этого. Разумеется, стандарт C++ задает такой минимальный диапазон для встроенных типов, который будет доступен во всех средах. В результате если вы в своих программах не выходите за рамки этих минимальных диапазонов, то ваши программы будут переносимы в другие среды. Последнее замечание: каждый компилятор C++ специфицирует диапазоны базовых типов в заголовке `<climits>`.

Минутная тренировка

1. Каковы семь базовых типов данных?
2. В чем заключается различие между целыми со знаком и без знака?

3. Может ли переменная типа **char** использоваться в качестве “маленького” целого?
 - 1 **char**, **wchar_t**, **int**, **float**, **double**, **bool**, **void** представляют собой семь базовых типов данных
 - 2 Целое со знаком может содержать как положительные, так и отрицательные значения. Целое без знака может содержать лишь положительные значения.
 - 3 Да
-

Типы данных с плавающей точкой

Переменные типов **float** и **double** используются либо при необходимости иметь дробные числа, либо когда ваше приложение имеет дело с очень большими или, наоборот, очень маленькими числами. Типы **float** и **double** различаются значением наибольшего (наименьшего) числа, которое может содержаться в переменных этих типов. Обычно тип **double** может хранить число, приблизительно на 10 порядков большее, чем **float**.

Из этих двух типов чаще используется **double**. Одна из причин этого заключается в том, что многие математические функции из библиотеки функций C++ используют значения типа **double**. Например, функция **sqrt()** возвращает значение типа **double**, являющееся квадратным корнем из аргумента того же типа. В приводимой ниже программе функция **sqrt()** используется для вычисления длины гипотенузы прямоугольного треугольника при заданных длинах двух его катетов.

```
/*
    Использование теоремы Пифагора для нахождения
    длины гипотенузы прямоугольного треугольника
    при заданных длинах двух его катетов.
*/

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

int main() {
    double x, y, z;

    x = 5.0;
    y = 4.0

    z = sqrt(x*x + y*y);
    cout << "Гипотенуза равна " << z;

    return 0;
}
```

Заголовок **<cmath>** нужен для функции **sqrt()**.

Функция **sqrt()** является частью математической библиотеки C++

Ниже приведен вывод программы:

Гипотенуза равна 6.40312

Еще одно замечание относительно приведенной выше программы. Поскольку `sqrt()` является частью стандартной библиотеки функций C++, в программу следует включить стандартный заголовок `<cmath>`.

Тип **long double** позволяет работать с очень большими или очень маленькими числами. Наиболее полезен этот тип в программах для научных исследований. Например, типом **long double** можно пользоваться для анализа астрономических данных.

Булев тип данных

Тип **bool** является относительно новым добавлением в C++. Он хранит булевы (т. е. истина или ложь) значения. В C++ определены две булевы константы, **true** (истина) и **false** (ложь), которые только и можно использовать с переменными этого типа.

Перед тем, как перейти к дальнейшему, важно понять, как определяются в C++ значения **true** и **false**. Одной из фундаментальных концепций C++ является интерпретация любого ненулевого значения как **true**, а нулевого — как **false**. Эта концепция полностью совместима с типом **bool**, потому что при использовании в булевом выражении C++ автоматически преобразует любое ненулевое значение в **true**, а нулевое — в **false**. Справедливо также и обратное: при использовании в небулевом выражении **true** преобразуется в 1, а **false** — в 0. Автоматическое преобразование нулевых и ненулевых значений в их булевы эквиваленты оказывается особенно важным в управляющих предложениях, как это будет показано в Модуле 3.

Ниже приводится программа, демонстрирующая использование типа **bool**:

```
// Демонстрация булевых значений.
```

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

int main() {
    bool b;

    b = false;
    cout << "b равно " << b << "\n";

    b = true;
```

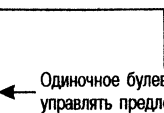
```
cout << "b равно " << b << "\n";

// булево значение может управлять предложением if
if(b) cout << "Это выполняется.\n";

b = false;
if(b) cout << "Это не выполняется.\n";

// результатом действия оператора отношения является
// значения true/false
cout << "10 > 9 есть " << (10 > 9) << "\n";

return 0;
}
```



Ниже приведен вывод этой программы:

```
b равно 0
b равно 1
Это выполняется.
10 > 9 есть 1
```

В приведенной программе есть три интересных момента. Во-первых, как видно из вывода программы, если значение типа **bool** выводится с помощью **cout**, на экране отображается 0 или 1. Как будет показано в дальнейшем, существуют опции вывода, которые позволяют вывести на экран слова “false” и “true”.

Во-вторых, значение булевой переменной может само по себе использоваться для управления предложением **if**. Нет необходимости писать предложение **if** таким образом:

```
if(b == true) ...
```

В-третьих, результатом действия оператора отношения, например, **<**, является булево значение. Именно поэтому выражение **10 > 9** выводит значение 1. Далее, вокруг выражения **10 > 9** необходимы дополнительные скобки, потому что оператор **<<** имеет более высокий приоритет, чем **<**.

Тип void

Тип **void** описывает выражение, не имеющее значения. Это может показаться странным, но далее в книге будет показано, как используется **void**.

Минутная тренировка

1. Каково основное различие между **float** и **double**?
 2. Какие значения может иметь переменная типа **bool**? В какое булево значение преобразуется ноль?
 3. Что такое **void**?
-
1. Основное различие между **float** и **double** состоит в допустимой величине значений, содержащихся в переменных этих типов
 2. Переменные типа **bool** могут иметь значения либо **true**, либо **false**. Ноль преобразуется в **false**
 3. **void** представляет собой тип величины, не имеющей значения
-

Проект 2-1 Разговор с Марсом

В точке, где Марс находится ближе всего к Земле, он все же отстоит от Земли приблизительно на 34000000 миль. Считая, что на Марсе есть кто-то, с кем вам хочется поговорить, какова будет задержка между моментом отправления радиосигнала с Земли и моментом достижения им Марса? В данном проекте создается программа, отвечающая на этот вопрос. Вспомним, что радиосигнал распространяется со скоростью света, т. е. приблизительно 186000 миль в секунду. Для того, чтобы определить задержку, следует разделить расстояние на скорость света. Выведем на экран значение задержки в секундах, а также в минутах.

Шаг за шагом

1. Создайте файл с именем **Mars.cpp**.
2. Для вычисления задержки вам понадобятся значения с плавающей точкой. Почему? Потому что временной интервал будет иметь дробную часть. Вот переменные, используемые программой:

```
double distance; // расстояние
double lightspeed; // скорость света
double delay; // задержка
double delay_in_min; // задержка в минутах
```

3. Присвойте переменным **distance** и **lightspeed** начальные значения, как показано ниже:

```
distance = 34000000.0; // 34000000 миль
lightspeed = 186000.0; // 186000 миль в секунду
```

4. Для вычисления задержки разделите **distance** на **lightspeed**. Вы получите задержку в секундах. Присвойте это значение переменной **delay** и выведите результат. Эти шаги показаны ниже:

```
delay = distance / lightspeed;  
  
cout << "Временная задержка при разговоре с Марсом: " <  
    delay << " секунд.\n";
```

5. Разделите число секунд в **delay** на 60, чтобы получить задержку в минутах; выведите этот результат с помощью таких строк:

```
delay_in_min = delay / 60.0;  
cout << "Это составляет " << delay_in_min << " минут.";
```

6. Ниже приведена полная программа **Mars.cpp**:

```
/*  
Проект 2-1  
  
Разговор с Марсом  
*/  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    double distance;  
    double lightspeed;  
    double delay;  
    double delay_in_min;  
  
    distance = 34000000.0; // 34000000 миль  
    lightspeed = 186000.0; // 186000 миль в секунду  
  
    delay = distance / lightspeed;  
  
    cout << "Временная задержка при разговоре с Марсом: " <<  
        delay << " секунд.\n";  
  
    delay_in_min = delay / 60.0;  
  
    cout << "Это составляет " << delay_in_min << " минут.";  
  
    return 0;  
}
```


7. Откомпилируйте и запустите программу. Вы получите следующий результат:

Временная задержка при разговоре с Марсом: 182.796 секунд.
Это составляет 3.04659 минут.

8. Попробуйте самостоятельно получить время задержки, возникающей при двусторонней связи с Марсом.

Цель

2.2. Литералы

Литералами называются фиксированные значения, которые может прочитать человек, и которые не могут быть изменены программой. Например, значение 101 представляет собой целочисленный литерал. Литералы часто называют *константами*. По большей части литералы и их использование интуитивно настолько понятны, что мы использовали их в той или иной форме во всех предыдущих программах. Теперь наступил момент, когда понятие литерала следует обсудить с формальной точки зрения.

В C++ литералы могут быть любого из базовых типов данных. Способ представления литерала зависит от его типа. Как упоминалось ранее, *символьные* литералы заключаются в одиночные кавычки. Например, 'a' и '%' суть символьные литералы.

Целочисленные литералы задаются в виде чисел без дробной части. Например, 10 и -100 суть целочисленные константы. Литералы с *плавающей точкой* требуют указания десятичной точки, за которой может следовать дробная часть числа. Например, 11.123 есть константа с плавающей точкой. C++ допускает также использование для чисел с плавающей точкой обозначений, принятых в научном мире (с указанием порядка в виде степени десяти).

Все литеральные значения имеют тот или иной тип данного, однако этот факт вызывает вопрос. Как вы знаете, существуют несколько типов целых чисел, например, `int`, `short int` или `unsigned long int`. Вопрос заключается в следующем: Как компилятор определяет тип литерала? Например, какой тип имеет число 123.23, `float` или `double`? Ответ на этот вопрос состоит из двух частей. Во-первых, компилятор C++ автоматически принимает определенные предположения относительно типа литерала и, во-вторых, вы можете для литерала явно указать его тип.

По умолчанию компилятор C++ помещает целочисленный литерал в минимальный по размеру и совместимый с конкретным значением литерала тип данного, начиная с `int`. Таким образом, для 16-битовых целых чисел литерал 10 будет иметь тип `int` по умолчанию, но литерал

103000 получит тип **long**. Хотя число 10 могло бы поместиться в тип **char**, компилятор не поступит таким образом, так как типы целочисленных литералов начинаются с **int**.

По умолчанию литералы с плавающей точкой получают тип **double**. Так, значение 123.23 будет типа **double**.

Практически для всех программ, которые вы будете составлять, уяся язык, умолчания компилятора вполне будут вас устраивать. В тех случаях, когда предположения по умолчанию о типах числовых литералов, которые принимает C++, вас не устраивают, вы можете задать конкретный тип числового литерала с помощью суффикса. Для типов с плавающей точкой чтобы задать для литерала тип **float**, следует вслед за числом указать спецификатор *F*. Если сопроводить число спецификатором *L*, число приобретет тип **long double**. Для целочисленных типов суффикс *U* указывает на тип **unsigned**, а *L* на **long**. (Для задания типа **unsigned long** следует использовать оба спецификатора, и *U*, и *L*.) Ниже приведено несколько примеров:

| Тип данного | 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

Как вы, возможно, знаете, в программировании иногда оказывается проще использовать не десятичную, а восьмеричную или шестнадцатеричную систему счисления. *Восьмеричная* система имеет в качестве основания число 8 и использует цифры от 0 до 7. *Шестнадцатеричная* система имеет в качестве основания число 16 и использует цифры от 0 до 9 плюс буквы от *A* до *F*, которые обозначают 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 — это то же самое, что 16 в десятичной системе. Поскольку эти две системы счисления используются очень часто, в C++ предусмотрена возможность задавать целочисленные литералы не только в десятичной системе, но и в двух других. Шестнадцатеричный литерал должен начинаться с 0x (ноль, за которым идет буква x). Восьмеричный литерал начинается с нуля. Вот два примера:

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

Спросим у эксперта

Вопрос: Вы показали, как задается символьный литерал. Можно ли тем же способом задать литерал типа `wchar_t`?

Ответ: Нет. Константе, состоящая из “широких” символов (т. е. из данных типа `wchar_t`) должен предшествовать спецификатор `L`. Например:

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

Здесь переменной `wc` присваивается значение константы из “широких” символов, эквивалентной букве `A`. В обычном каждодневном программировании вы, скорее всего, не будете использовать “широкие” символы, однако они могут вам понадобиться, если вы занимаетесь переводом программ на другие языки.

Строковые литералы

Помимо литералов предопределенных типов, C++ поддерживает еще один тип литералов: строку. *Строка* представляет собой последовательность символов, заключенную в двойные кавычки. Пример строки: “Это проверка”. В предыдущих программных примерах вы уже сталкивались со строками в некоторых предложениях `cout`. Обратите внимание на важное обстоятельство: хотя C++ позволяет определять строковые константы, в нем нет встроенного типа строковых данных. Строки, как вы вскоре увидите, описываются в C++, как массивы символов. (C++, однако, *имеет* строковый тип в своей библиотеке классов.)

Символьные Esc-последовательности

Из большинства отображаемых на экране символов можно образовывать символьные литералы, заключая их в одиночные кавычки, однако несколько символов, например, символ возврата каретки, создают проблемы при работе в текстовом редакторе. Кроме того, некоторые символы, например, одиночные или двойные кавычки, имеют в C++ определенное значение, и их нельзя использовать непосредственно в качестве символьных констант. Для таких случаев в C++ предусмотрено специальное средство — *управляющие*, или *Esc-последовательности*, которые иногда называют *символьными константами с обратным слешем*; эти константы перечислены в табл. 2-3. Теперь вы видите, что комбинация `\n`, которая встречалась в наших программах, является одной из таких Esc-последовательностей.

Таблица 2-3. Символьные Esc-последовательности

| Код | Значение |
|------------------|---|
| <code>\b</code> | Шаг назад |
| <code>\f</code> | Перевод страницы |
| <code>\n</code> | Новая строка |
| <code>\r</code> | Возврат каретки |
| <code>\t</code> | Горизонтальная табуляция |
| <code>\"</code> | Двойные кавычки |
| <code>\'</code> | Символ одиночной кавычки |
| <code>\\</code> | Обратная косая черта, обратный слеш |
| <code>\v</code> | Вертикальная табуляция |
| <code>\a</code> | Звуковой сигнал |
| <code>\?</code> | ? |
| <code>\N</code> | Восьмеричная константа (где N есть сама восьмеричная константа) |
| <code>\xN</code> | Шестнадцатеричная константа (где N есть сама шестнадцатеричная константа) |

Спросим у эксперта

Вопрос: Является ли строка, состоящая из одного символа, символьным литералом? Например, `"k"` и `'k'` — это одно и то же?

Ответ: Нет. Не следует путать строки с символами. Символьный литерал представляет одиночный символ (букву, цифру и проч.) типа **char**. Строка же, даже содержащая лишь одну букву, все еще является строкой. Хотя строки состоят из символов, строки и символы являются объектами разных типов.

В приведенной ниже программе иллюстрируется использование некоторых Esc-последовательностей:

// Демонстрация некоторых Esc-последовательностей.

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

int main()
{
    cout << "один\tдва\tтри\n";
    cout << "123\b\b45";
    return 0;
}
```

Последовательность `\b\b` выполнит два шага назад и затирание 2 и 3

Вывод этой программы:

```
один    два    три
145
```

В этой программе в предложении `cout` используются символы табуляции для позиционирования слов “два” и “три”. Следующее предложение `cout` выводит символы 123. Далее выводятся два символа возврата на шаг, которые затирают выведенные уже 2 и 3. Наконец, отображаются символы 4 и 5.

Минутная тренировка

1. Каков тип литерала 10 по умолчанию? А каков тип литерала 10.0?
 2. Каким образом задать 100 в виде `long int`? Каким образом задать 100 в виде `unsigned int`?
 3. Что такое `\b`?
 - 1 10 имеет тип `int`, а 10.0 — `double`
 - 2 100 в виде `long int` записывается как 100L. 100 в виде `unsigned int` записывается как 100U.
 - 3 `\b` представляет собой Esc-последовательность, выполняющую возврат на шаг
-

Цель

2.3. Подробнее о переменных

Мы уже познакомились с переменными в Модуле 1. Здесь мы поговорим о них подробнее. Как вы уже знаете, переменные объявляются с помощью предложения такого вида:

```
тип имя-переменной;
```

Здесь *тип* — это тип данного, соответствующий объявляемой переменной, а *имя-переменной* — ее имя. Вы может объявить переменную любого допустимого типа. Создавая переменную, вы создаете тем самым экземпляр ее типа. Таким образом, свойства переменной определяются ее типом. Например, переменная типа `bool` содержит булевы значения; ее нельзя использовать для хранения значений с плавающей точкой. При этом недопустимо изменять тип переменной в течение времени ее существования. Так, из переменной типа `int` нельзя сделать переменную типа `double`.

Инициализация переменной

Переменной можно присвоить значение одновременно с ее объявлением. Для этого укажите после имени переменной знак равенства, а

за ним — присваиваемое переменной значение. Такое предложение носит название *инициализации переменной*. Общая форма предложения инициализации выглядит так:

тип переменная = значение;

Здесь *значение* — это значение, присваиваемое переменной одновременно с ее созданием.

Приведем несколько примеров:

```
int count = 10; // присвоим count начальное значение 10
char ch = 'X'; // инициализируем ch кодом буквы X
float f = 1.2F; // f инициализируется значением 1.2
```

При объявлении двух или нескольких переменных одного типа в форме списка с разделяющими запятыми, вы можете присвоить одной или нескольким переменным начальные значения. Например,

```
int a, b = 8, c = 19, d; // b и c получают начальные значения
```

В этом примере инициализируются только переменные **b** и **c**.

Динамическая инициализация

Хотя в предыдущих примерах для инициализации переменных использовались только константы, C++ позволяет инициализировать переменные динамически, с помощью любых выражений, образующих определенные значения в то время, когда объявляется инициализируемая переменная. Ниже приведена короткая программа, вычисляющая объем цилиндра при заданных значениях радиуса его основания и высоты:

// Демонстрация динамической инициализации.

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

int main() {
    double radius = 4.0, height = 5.0;

    // динамически инициализируем переменную volume
    double volume = 3.1416 * radius * radius * height;

    cout << "Объем равен " << volume;

    return 0;
}
```

volume инициализируется динамически во время выполнения

В приведенной программе объявляются три локальные переменные — **radius**, **height** и **volume**. Первые две, **radius** и **height**, инициализируются константами. Однако переменная **volume** инициализируется динамически; ей присваивается значение объема цилиндра. Здесь важно отметить, что инициализирующее выражение может использовать любые элементы, принимающие определенные значения к моменту инициализации, включая вызовы функций, другие переменные и литералы.

Операторы

C++ предоставляет богатый набор операторов. *Оператором* называется символ, обозначающий для компилятора указание на выполнение определенного математического или логического действия. В C++ предусмотрены четыре основных класса операторов: *арифметических*, *побитовых*, *отношения* и *логических*. Кроме этого, в C++ имеются несколько дополнительных операторов, которые обслуживают определенные специфические ситуации. В этой главе мы рассмотрим операторы арифметические, отношения и логические. Также будет рассмотрен оператор присваивания. Операторы побитовые и специальные будут описаны позже.

Цель

2.4. Арифметические операторы

В C++ определены следующие арифметические операторы:

| Оператор | Значение |
|----------|---------------------------------|
| + | Сложение |
| − | Вычитание (также унарный минус) |
| * | Умножение |
| / | Деление |
| % | Деление по модулю |
| ++ | Инкремент |
| -- | Декремент |

Операторы **+**, **−**, ***** и **/** выполняют те же действия, что и в обычной алгебре. Их можно применять к любым встроенным типам данных. Их также можно использовать с переменными типа **char**.

Оператор % (деление по модулю) возвращает остаток от целочисленного деления. Вспомним, что когда знак / используется с целым числом, получаемый остаток отбрасывается; например, $10/3$ при целочисленном делении будет равно 3. Для получения остатка в операции деления следует воспользоваться оператором %. Так, $10 \% 3$ равно 1. В C++ оператор % может использоваться только с целочисленными операндами; к типам с плавающей точкой он неприменим.

Следующая программа демонстрирует использование операции деления по модулю:

```
// Демонстрация использования оператора %.
```

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

int main()
{
    int x, y;

    x = 10;
    y = 3;
    cout << x << " / " << y << " равно " << x / y <<
        " с остатком " << x % y << "\n";

    x = 1;
    y = 2;
    cout << x << " / " << y << " равно " << x / y << "\n" <<
        x << " % " << y << " равно " << x % y;

    return 0;
}
```

Вывод программы приведен ниже:

```
10 / 3 равно 3 с остатком 1
1 / 2 равно 0
1 % 2 равно 1
```

Инкремент и декремент

В Модуле 1 нам уже встречались операторы инкремента и декремента ++ и --. Эти операторы обладают некоторыми очень интересными свойствами. Начнем с того, что точно определим действие операторов инкремента и декремента.

Оператор инкремента добавляет к операнду 1, а оператор декремента вычитает 1. Таким образом,

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

эквивалентно

```
x++;
```

и

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

эквивалентно

```
--x;
```

Оба оператора, и инкремента и декремента, могут как предшествовать операнду (префикс), так и следовать за операндом (постфикс). Например,

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

может быть записано и как

```
++x; // префиксная форма
```

и как

```
x++; // постфиксная форма
```

В приведенном примере безразлично, используется ли инкремент как префикс или как постфикс. Однако если инкремент или декремент используется как часть большего выражения, возникает важное различие. Если оператор инкремента или декремента предшествует операнду, C++ выполняет операцию до получения значения операнда с целью использования его в оставшейся части выражения. Если же оператор следует за операндом, C++ сначала получит значение операнда, и лишь затем выполнит его инкремент или декремент. Рассмотрим такой пример:

```
x = 10;
```

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

В этом случае y получит значение 11. Однако, если написать эти строки иначе:

```
x = 10;
```

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

значение y окажется равным 10. В обоих случаях конечное значение x будет 11; разница в том, когда это случится. Возможность управлять моментом выполнения операций инкремента и декремента оказывается весьма полезной.

Относительные приоритеты арифметических операторов показаны ниже:

| | |
|--------|-------------------|
| Высший | ++ -- |
| | - (унарный минус) |
| | * / % |
| Низший | + - |

Операторы, имеющие одинаковый приоритет, выполняются компилятором слева направо. Разумеется, для изменения порядка выполнения арифметических операций могут быть использованы скобки (круглые). Скобки рассматриваются C++ точно так же, как и любыми другими компьютерными языками: они переводят операцию или группу операций на более высокий уровень приоритета.

Спросим у эксперта

Вопрос: Имеет ли оператор инкремента ++ какое-либо отношение к имени языка C++?

Ответ: Да! Как вы знаете, C++ построен на базе языка C. C++ добавил в C несколько усовершенствований, большинство которых поддерживает объектно-ориентированное программирование. Таким образом, C++ представляет собой расширение (инкремент) языка C, и добавление к имени C символов ++ (которые, разумеется, образуют оператор инкремента) вполне оправдано.

Страуструп первоначально назвал C++ “C с классами”, однако по предложению Рика Маскитти он позже изменил имя на C++. Хотя новый язык в любом случае был обречен на успех, присвоение ему имени C++ действительно гарантировало ему место в истории, потому что любой программист на C сразу же его узнавал!

Цель

2.5. Операторы отношения (сравнения) и логические

В выражениях *оператор отношения* и *логический оператор* слово *отношение* обозначает взаимоотношение двух величин, т. е. результат их

сравнения, а слово *логический* обозначает способ, которым объединяются истинное и ложное значения. Поскольку операторы отношения образуют истинный или ложный результат, они часто используются совместно с логическими операторами. Именно по этой причине они обсуждаются здесь вместе.

Операторы отношения и логические перечислены в табл. 2-4. Обратите внимание на отношения равенства и неравенства: в C++ *не равно* обозначается знаками `!=`, а *равно* – двойным знаком равенства, `==`. В C++ результат операции отношения или логической образует результат типа `bool`. Другими словами, результат операции отношения или логической всегда равен либо `true`, либо `false`.

Таблица 2-4. Операторы отношения и логические в C++

| Операторы отношения | |
|-------------------------|----------------------|
| Оператор | Значение |
| <code>></code> | Больше чем |
| <code>>=</code> | Больше чем или равно |
| <code><</code> | Меньше чем |
| <code><=</code> | Меньше чем или равно |
| <code>==</code> | Равно |
| <code>!=</code> | Не равно |
| Операторы логические | |
| Оператор | Значение |
| <code>&&</code> | И |
| <code> </code> | ИЛИ |
| <code>!</code> | НЕ |



Замечание

Для старых компиляторов результат операции отношения и логической может быть целым числом, принимающим значение 0 или 1. Это различие носит в основном академический характер, потому что C++, как это уже отмечалось, автоматически преобразует `true` в 1, а `false` в 0 и наоборот.

Операнды, участвующие в операции отношения, могут принадлежать почти любому типу; необходимо только, чтобы их сравнение имело смысл. Операнды логической операции должны

образовывать истинный или ложный результат. Поскольку ненулевое значение истинно, а нулевое ложно, это означает, что логические операторы допустимо использовать с любым выражением, дающим нулевой или ненулевой результат. Фактически можно использовать любое выражение, кроме тех, результат которых есть **void**.

Логические операторы используются для поддержки базовых логических операций И (AND), ИЛИ (OR) и НЕ (NOT), согласно следующей таблице истинности:

| р | q | р И q | р ИЛИ q | НЕ р |
|--------|--------|--------|---------|--------|
| Ложь | Ложь | Ложь | Ложь | Истина |
| Ложь | Истина | Ложь | Истина | Истина |
| Истина | Истина | Истина | Истина | Ложь |
| Истина | Ложь | Ложь | Истина | Ложь |

Ниже приведена программа, демонстрирующая несколько операторов отношения и логических:

```
// Демонстрация операторов отношения и логических.
```

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

int main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) cout << "i < j\n";
    if(i <= j) cout << "i <= j\n";
    if(i != j) cout << "i != j\n";
    if(i == j) cout << "это не будет выполняться\n";
    if(i >= j) cout << " это не будет выполняться\n";
    if(i > j) cout << " это не будет выполняться\n";

    b1 = true;
    b2 = false;
    if(b1 && b2) cout << " это не будет выполняться\n";
    if(!(b1 && b2)) cout << "!(b1 && b2) есть истина\n";
    if(b1 || b2) cout << "b1 || b2 есть истина\n";

    return 0;
}
```

Ниже приведен вывод этой программы:

```
i < j
i <= j
i != j
!(b1 && b2) есть истина
b1 || b2 есть истина
```

Оба типа операторов, и логические, и отношения, имеют более низкий приоритет, чем арифметические операторы. Это означает, что выражение вроде `10 > 1 + 12` дает тот же результат, что и вариант `10 > (1 + 12)`. Результат, разумеется, ложен.

С помощью логических операторов можно объединять любое количество операторов отношения. Например, в этом выражении объединены три оператора отношения:

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

В приведенной ниже таблице показаны относительные приоритеты операторов отношения и логических:

| | |
|--------|-----------|
| Высший | ! |
| | > >= < <= |
| | == != |
| | && |
| Низший | |

Проект 2-2:

Конструирование логической операции исключающее ИЛИ

В C++ не определен логический оператор, выполняющий операцию исключающего ИЛИ (XOR). Исключающее ИЛИ — это двухместная операция, результат которой истинен лишь в том случае, когда один и только один операнд истинен. Она имеет следующую таблицу истинности:

| p | q | p XOR q |
|--------|--------|---------|
| Ложь | Ложь | Ложь |
| Ложь | Истина | Истина |
| Истина | Ложь | Истина |
| Истина | Истина | Ложь |

Некоторые программисты считают, что отсутствие операции XOR в C++ – серьезная ошибка. Другие придерживаются точки зрения, что отсутствие логического оператора XOR отражает рациональность языка C++ – стремление к отсутствию избыточности. Действительно, логическую операцию XOR нетрудно создать с помощью предоставляемых C++ трех базовых логических операций.

В этом проекте мы сконструируем операцию XOR с помощью операторов `&&`, `||` и `!`. После этого вы сможете сами решить, является ли отсутствие логического оператора XOR ошибкой проектирования или элегантной чертой языка!

Шаг за шагом

1. Создайте файл с именем **XOR.cpp**.
2. Если `p` и `q` являются булевыми переменными, логический оператор XOR конструируется таким образом:

`(p || q) && !(p && q)`

Рассмотрим это выражение детально. Прежде всего над `p` и `q` выполняется операция ИЛИ. Если результат истинен, это значит, что по меньшей мере один из операндов истинен. Далее, над `p` и `q` выполняется операция И. Результат будет истинен, если оба операнда истинны. Затем этот результат инвертируется с помощью оператора НЕ. Выражение `!(p && q)` будет истинно, когда либо `p`, либо `q`, либо оба вместе ложны. Наконец, этот результат логически умножается (операция И) на результат операции `(p || q)`. В конце концов все выражение будет истинно, если один из операндов (но не оба вместе) истинен.

3. Ниже приводится полный текст программы **XOR.cpp**. Она демонстрирует результаты операции XOR для всех четырех возможных комбинаций значений (т. е. сочетаний true/false) операндов `p` и `q`.

```
/*
```

```
Проект 2-2
```

```
Реализация операции XOR с помощью логических операторов C++.
```

```
*/
```

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
```

```
int main()
{
    bool p, q;

    p = true;
    q = true;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = false;
    q = true;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = true;
    q = false;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    p = false;
    q = false;

    cout << p << " XOR " << q << " равно " <<
        ( (p || q) && !(p && q) ) << "\n";

    return 0;
}
```

4. Откомпилируйте и запустите программу. Ниже приведен ее вывод:

```
1 XOR 1 равно 0
0 XOR 1 равно 1
1 XOR 0 равно 1
0 XOR 0 равно 0
```

5. Обратите внимание на внешние скобки, окружающие операцию XOR в предложениях `cout`. Эти скобки необходимы в силу действия приоритетов операторов C++. Оператор `<<` имеет более высокий приоритет, чем логические операторы. Чтобы убедиться в этом, удалите внешние скобки и перекомпилируйте программу. Вы увидите, что компилятор даст сообщение об ошибке.

Минутная тренировка

1. Для чего предназначен оператор %? К данным каких типов он применим?
 2. Каким образом вы объявите переменную типа `int` с именем `index` и с начальным значением 10?
 3. Каков тип результата операций отношения или логических?
- 1 Оператор деления по модулю % возвращает остаток от целочисленного деления. Его можно использовать с целыми числами.
2 `int index = 10;`
3. Результат операций отношения и логических имеет тип `bool`
-

Цель

2.6. Оператор присваивания

Уже в Модуле 1 вы использовали оператор присваивания. Теперь наступило время рассмотреть его с формальной точки зрения. *Оператор присваивания* обозначается одиночным знаком равенства, `=`. Действие этого оператора в C++ практически такое же, как и в любом другом компьютерном языке. Его общая форма:

переменная = выражение;

Здесь переменная принимает значение используемого выражения.

Оператор присваивания имеет любопытное свойство: он позволяет создавать цепочку присваиваний. Рассмотрим, например, такой фрагмент:

```
int x, y, z;
```

```
x = y = z = 100; // присвоить x, y, и z значение 100
```

В этом фрагменте переменные `x`, `y` и `z` получают значение 100 в одном предложении. Такое присваивание допустимо, так как оператор `=` выдает значение выражения, стоящего справа от него. Значение выражения `z = 100` есть 100, и оно присваивается переменной `y`, а затем, в свою очередь, переменной `x`. С помощью “цепочки присваиваний” легко придать группе переменных одно и то же значение.

Цель

2.7. Составные присваивания

В C++ предусмотрены специальные операторы составных присваиваний, которые упрощают написание некоторых предло-

жений присваивания. Приведенное ниже предложение присваивания

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

может быть записано с использованием составного присваивания таким образом:

```
x += 10;
```

Операторная пара `+=` указывают компилятору, что переменной `x` следует присвоить значение `x` плюс 10.

Вот другой пример. Предложение

```
x = x - 100;
```

равносильно следующему:

```
x -= 100;
```

Оба предложения присваивают переменной `x` значение `x` минус 100.

Для большинства двухместных операторов (т. е. операторов, требующих двух операндов) в C++ предусмотрена возможность составного присваивания. Таким образом, выражение вида

переменная = переменная оператор выражение;

может быть преобразовано в такую сжатую форму:

переменная оператор = выражение;

Из-за того, что предложения составного присваивания короче своих развернутых эквивалентов, операторы составного присваивания иногда называют операторами *краткого присваивания*.

Операторы составного присваивания обладают двумя достоинствами. Во-первых, они более компактны по сравнению со своими “полными” эквивалентами. Во-вторых, они могут порождать более эффективный выполнимый код (потому что при их использовании операнд оценивается лишь один раз). По этим причинам операторы составного присваивания широко используются в профессиональном программировании на C++.

Цель

2.8. Преобразование типов в операциях присваивания

В тех случаях, когда переменные одного типа сопоставляются с переменными другого типа, выполняется операция *преобразования типа*.

В предложениях присваивания правило преобразования типа весьма простое: значение правой стороны операции присваивания (т. е. стороны выражения) преобразуется в тип левой стороны (переменной-мишени), как это проиллюстрировано ниже:

```
int x;  
char ch;  
float f;  
  
ch = x; /* строка 1 */  
x = f; /* строка 2 */  
f = ch; /* строка 3 */  
f = x; /* строка 4 */
```

В строке 1 левые (старшие) 8 бит целочисленной переменной `x` отбрасываются, и в `ch` помещаются младшие 8 бит. Если значение `x` было между -128 и 127 , `ch` и `x` будут иметь совпадающие значения. В противном случае значение `ch` будет отражать только младшие биты `x`. В строке 2 переменная `x` примет целую часть `f`. В строке 3 `f` преобразует 8-битовое целое значение, хранящееся в `ch`, в то же значение, но в формате с плавающей точкой. Это же произойдет в строке 4, только здесь `f` преобразует в формат с плавающей точкой значение типа `int`.

В процессе преобразования целых (`int`) значений в символы или длинных целых в обычные целые теряется соответствующее количество старших битов числа. Во многих 32-разрядных средах это приведет при преобразовании `int` в `char` к потере 24 бит, а при преобразовании `int` в `short` — 16 бит. В случае преобразования типа с плавающей точкой в `int` будет теряться дробная часть. Если размер типа-мишени недостаточен для восприятия результата, получится бессмысленное значение.

Не забывайте: хотя C++ автоматически преобразует любые встроенные типы друг в друга, результат не обязательно будет соответствовать вашим ожиданиям. Будьте осторожны при смешивании типов в выражениях.

Выражения

Операторы, переменные и литералы используются в качестве компонентов *выражений*. Возможно, вы уже представляете себе общую форму выражения из опыта программирования или из алгебры. Однако некоторые особенности выражений мы все же обсудим.

Цель

2.9. Преобразование типа в выражениях

В тех случаях, когда переменные различающихся типов смешиваются в выражении, они преобразуются к одному типу. Прежде всего все значения **char** и **short int** автоматически повышаются до **int**. Далее, все операнды повышаются до самого высокого типа, участвующего в выражении. Повышение типа осуществляется по мере выполнения последовательных операций. Например, если один операнд имеет тип **int**, а другой — **long int**, тип **int** повышается до **long int**. Или если любой из операндов имеет тип **double**, другой операнд повышается до **double**. Таким образом, преобразование, например, из **char** в **double** вполне допустимо. После выполнения преобразования каждая пара операндов будет иметь один и тот же тип, и тип результата каждой операции будет совпадать с типом обоих операндов.

Преобразование в **bool** и из **bool**

Как уже упоминалось, значения типа **bool**, в случае их использования в целочисленных выражениях, автоматически преобразуются в целые 0 или 1. Когда целочисленный результат преобразуется в тип **bool**, 0 превращается в **false**, а любое ненулевое значение — в **true**. Хотя тип **bool** является относительно недавним добавлением в C++, его автоматические преобразования в целые или из целых означают, что наличие **bool** не отражается на старых кодах. Кроме того, автоматические преобразования позволяют C++ сохранять исходные определения истины и лжи как нулевого и ненулевого значения.

Цель

2.10. Приведение типа

В C++ предусмотрена возможность принудительного присваивания выражению конкретного типа с помощью конструкции, называемой *приведением типа*. В C++ определены пять видов приведений. Четыре из них дают пользователю возможность полного контроля над приведением; они будут рассмотрены в книге после введения понятия объектов. Однако имеется еще один вид приведения, который можно рассмотреть и использовать уже сейчас. Этот вид является наиболее общим, так как он позволяет преобразовать любой тип в любой другой. Он, кстати, является единственным видом приведения, который

поддерживался ранними версиями C++. Общая форма этой конструкции приведения такова:

(тип) выражение

Здесь *тип* — это тип-мишень, в который требуется преобразовать выражение. Если, например, вы хотите удостовериться, что выражение $x/2$ при вычислении дает тип `float`, вы должны написать:

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

Приведения считаются операторами. В качестве оператора приведение является одноместным (унарным) и имеет тот же относительный приоритет, что и любой другой одноместный оператор.

В некоторых случаях приведение может оказаться чрезвычайно полезным. Например, в качестве счетчика цикла естественно использовать целочисленную переменную, однако вам может понадобиться выполнять с ней какие-либо вычисления, приводящие к дробному числу, как это делается в приводимой ниже программе:

```
// Демонстрация приведения типа.
```

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

```
int main(){
    int i;
    for(i=1; i <= 10; ++i )
        cout << i << " / 2 равно: " << (float) i / 2 << '\n';
    return 0;
}
```

Приведение к типу `float` позволяет наблюдать дробную часть результата



Ниже приведен вывод этой программы:

```
1/ 2 равно: 0.5
2/ 2 равно: 1
3/ 2 равно: 1.5
4/ 2 равно: 2
5/ 2 равно: 2.5
6/ 2 равно: 3
7/ 2 равно: 3.5
8/ 2 равно: 4
9/ 2 равно: 4.5
10/ 2 равно: 5
```

Если бы в этом примере не осуществлялось приведение (`float`), то выполнялось бы только целочисленное деление. Приведение

обеспечивает получение (и вывод на экран) дробной части результата.

Цель

2.11. Пробелы и скобки

В C++ допустимо включать в выражение символы табуляции и пробелы, чтобы повысить наглядность и удобочитаемость программы. Например, приводимые ниже выражения эквивалентны, но второе заметно легче читать:

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

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

Скобки (круглые) повышают относительный приоритет содержащихся в них операций, как это имеет место в алгебре. Использование избыточных или дополнительных скобок не приведет к ошибкам или замедлению выполнения выражения. Весьма полезно использовать скобки, чтобы подчеркнуть и сделать наглядным порядок выполнения вычислений, как для себя, так и для тех, кто, возможно, в дальнейшем захочет разобраться в вашей программе. Например, какое из приведенных ниже выражений легче читать?

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

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

Проект 2-3

Вычисление регулярных платежей по ссуде

В этом проекте вы должны написать программу, которая вычисляет сумму регулярных платежей по ссуде, например, на покупку автомашины. Исходя из данных значений суммы ссуды, срока займа, числа выплат в год и ставки процента, программа определит величину регулярных платежей. Поскольку в программе выполняются финансовые вычисления, вам понадобятся для них переменные с плавающей точкой. Наиболее распространенным типом такого рода переменных является **double**; мы им и воспользуемся. В этом проекте также демонстрируется одна из новых для нас библиотечных функций C++: **pow()**.

Для вычисления регулярного платежа вы должны использовать следующую формулу:

$$\text{Платеж} = \frac{\text{Процент} * (\text{СуммаСсуды} / \text{ПлатежейВГод})}{1 - ((\text{Процент} / \text{ПлатежейВГод}) + 1)^{\text{ПлатежейВГод} * \text{ЧислоЛет}}}$$

Здесь Процент определяет ставку процента по ссуде, СуммаСсуды содержит начальную сумму ссуды, ПлатежейВГод задает число платежей в год, а ЧислоЛет указывает срок в годах, на который выдана ссуда.

Обратите внимание на то, что в знаменателе этой формулы вы должны возвести одно значение в степень другого. Для выполнения этой операции следует использовать функцию **pow()**. Вызов этой функции осуществляется следующим образом:

результат = **pow** (*основание*, *показатель степени*);

Функция **pow()** возвращает значение основания, возведенного в указанный показатель степени. Аргументы **pow()** являются значениями **double**; возвращает **pow()** также значение типа **double**.

Шаг за шагом

1. Создайте файл с именем **RegPay.cpp**
2. Ниже перечислены переменные, используемые в программе:

```
double Principal;      // начальная сумма ссуды
double IntRate;        // ставка процента по ссуде,
                        // например, 0.075
double PayPerYear;     // число платежей в год
double NumYears;       // число лет, на которые выдана
                        // ссуда
double Payment;        // регулярный платеж
double numer, denom;   // временные рабочие переменные
double b, e;           // основание и порядок для вызова
                        // функции pow()
```

Обратите внимание на то, что объявление каждой переменной сопровождается комментарием, поясняющим смысл этой переменной. Такой комментарий облегчит любому, заинтересовавшемуся вашей программой, понять, зачем нужна та или иная переменная. Хотя мы не будем включать такие детальные комментарии в большинство приводимых в книге коротких программ, однако в принципе это весьма полезная практика, особенно когда вы начнете писать более сложные и длинные программы.

3. Добавьте в программу следующие строки кода, которые будут вводить информацию о ссуде:

```
cout << "Введите сумму ссуды: ";
cin >> Principal;

cout << "Введите процентную ставку (например, 0.075): ";
cin >> IntRate;

cout << "Введите число платежей в год: ";
cin >> PayPerYear;

cout << "Введите число лет: ";
cin >> NumYears;
```

4. Добавьте строки, выполняющие финансовые вычисления:

```
numer = IntRate * Principal / PayPerYear;

e = -(PayPerYear * NumYears);
b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;
```

5. Завершите программу, выведя на экран сумму регулярного платежа, как это показано ниже:

```
cout << "Платеж составит " << Payment;
```

6. Далее следует полный текст программы **RegPay.cpp**:

```
/*
    Проект 2-3      Вычисление регулярного платежа по ссуде.

    Назовите этот файл RegPay.cpp
*/

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

int main() {
    double Principal;          // начальная сумма ссуды
```

```
double IntRate;           // ставка процента по ссуде,
                           // например, 0.075
double PayPerYear;        // число платежей в год
double NumYears;          // число лет, на которые выдана
                           // ссуда
double Payment;           // регулярный платеж
double numer, denom;      // временные рабочие переменные
double b, e;              // основание и порядок для
                           // вызова функции pow()

cout << "Введите сумму ссуды: ";
cin >> Principal;

cout << "Введите процентную ставку (например, 0.075): ";
cin >> IntRate;

cout << "Введите число платежей в год: ";
cin >> PayPerYear;

cout << "Введите число лет: ";
cin >> NumYears;
numer = IntRate * Principal / PayPerYear;

e = -(PayPerYear * NumYears);
b = (IntRate / PayPerYear) + 1;

denom = 1 - pow(b, e);

Payment = numer / denom;

cout << "Платеж составит " << Payment;

return 0;
}
```

Ниже приведен ход пробного запуска программы:

```
Введите сумму ссуды: 10000
Введите процентную ставку (например, 0.075): 0.075
Введите число платежей в год: 12
Введите число лет: 5
Платеж составит 200.379
```

7. Попробуйте самостоятельно модифицировать программу, чтобы она выводила полную сумму процентов, выплаченных за все время существования ссуды.

✓ Вопросы для самопроверки

1. Какие типы целых чисел поддерживает C++?
2. Какой тип будет присвоен по умолчанию числу 12.2?
3. Какие значения может иметь переменная типа `bool`?
4. Какой тип данных соответствует длинному целому?
5. Какая Esc-последовательность создает символ табуляции? А какая Esc-последовательность дает звуковой сигнал?
6. Строка окружается двойными кавычками. Справедливо ли это утверждение?
7. Перечислите шестнадцатеричные цифры.
8. Приведите общую форму объявления переменной с ее одновременной инициализацией.
9. Каково назначение оператора `%`? Можно ли его использовать с переменными с плавающей точкой?
10. Опишите, чем различаются префиксная и постфиксная формы оператора инкремента.
11. Какие из приведенных ниже операторов относятся к логическим операторам C++?
 - A. `&&`
 - B. `##`
 - C. `||`
 - D. `$$`
 - E. `!`
12. Каким другим способом можно записать предложение

`x = x + 12;`

13. Что такое приведение типа?
14. Напишите программу, которая находит все простые числа в диапазоне от 1 до 100.

Модуль

3

Предложения управления программой

Цели, достигаемые в этом модуле

- 3.1 Узнать больше о предложении if
- 3.2 Изучить предложение switch
- 3.3 Рассмотреть еще раз предложение for
- 3.4 Разобраться в использовании цикла while
- 3.5 Научиться использовать цикл do-while
- 3.6 Понять, где надо использовать предложение break
- 3.7 Научиться использовать предложение continue
- 3.8 Познакомиться со вложенными циклами
- 3.9 Рассмотреть предложение goto

В этом модуле обсуждаются предложения, управляющие ходом выполнения программы. В C++ имеются три категории предложений управления программой: предложения *выбора*, которые включают в себя **if** и **switch**; предложения *циклов*, которые включают в себя циклы **for**, **while** и **do-while**; и предложения *переходов*, включающие в себя **break**, **continue**, **return** и **goto**. К предложению **return** мы вернемся позже; здесь будут рассмотрены все оставшиеся предложения управления, включая и предложения **if** и **for**, с которыми вы уже кратко познакомились.

Цель

3.1. Предложение **if**

Предложение **if** уже использовалось нами в Модуле 1. Теперь наступило время изучить его более детально. Полная форма предложения **if** выглядит так:

```
if(выражение) предложение;  
else предложение;
```

Здесь мишенями **if** и **else** являются одиночные предложения. Предложение **else** не является обязательным. Мишенями и для **if**, и для **else** могут быть блоки предложений. Общая форма **if** с использованием блоков выглядит так:

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

Если условное выражение истинно, мишень для **if** выполняется; в противном случае выполняется мишень для **else**, если она существует. Ни при каких обстоятельствах эти две мишени не будут выполняться вместе. Условное выражение, управляющее конструкцией **if-else**, может быть любым допустимым в C++ выражением, дающим в результате истину или ложь.

Приводимая ниже программа демонстрирует использование предложения **if** в упрощенном варианте игры “Угадайте магическое число”. Программа генерирует случайное число, запрашивает вашу догадку и выводит сообщение **** Правильно ****, если вы угадали ма-

гическое число. В программе используется еще одна библиотечная функция C++, называемая **rand()**, которая возвращает случайно выбранное целое число. Эта функция требует подсоединения заголовка **<cstdlib>**.

// Программа "Магическое число".


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

```
int main()
{
    int magic; // магическое число
    int guess; // догадка пользователя

    magic = rand(); // получим случайное число
```

```
    cout << "Вводите вашу догадку: ";
    cin >> guess;
```

Если догадка соответствует "магическому числу", выводится сообщение



```
    if(guess == magic) cout << "*** Правильно ***";
```

```
    return 0;
}
```

Программа с помощью предложения **if** определяет, совпадает ли догадка пользователя с магическим числом. Если совпадает, на экран выводится поощряющее сообщение.

Приводимая ниже программа является усовершенствованием предыдущей. В этом варианте используется предложение **else** для вывода соответствующего сообщения в случае ввода пользователем неправильного числа:

// Программа "Магическое число": 1-е усовершенствование.

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

```
int main()
{
    int magic; // магическое число
    int guess; // догадка пользователя

    magic = rand(); // получим случайное число
```

```

cout << "Вводите вашу догадку: ";
cin >> guess;

if(guess == magic) cout << "*** Правильно ***";
else cout << "...Жаль, но вы ошиблись.";

return 0;
}

```

↑ При неправильной догадке
тоже выводится сообщение

Условные выражения

Начинающих программистов иногда сбивает с толку то обстоятельство, что любое допустимое в C++ выражение может быть использовано для управления предложением `if`. Отсюда видно, что условное выражение не обязательно должно содержать только операторы отношения или логические и операнды типа `bool`. Требуется только, чтобы результат вычисления управляющего предложения мог рассматриваться как истинный или ложный. Вспомним (см. предыдущий модуль), что значение 0 автоматически преобразуется в `false`, а все ненулевые значения в `true`. Таким образом, любое выражение, результатом которого является 0 или ненулевое значение, может быть использовано для управления предложением `if`. Приведем в качестве примера программу, которая считывает с клавиатуры два целых числа и выводит их частное. Предложение `if`, управляемое вторым числом, используется для предотвращения деления на ноль.

```

// Использование значения value для управления
// предложением if.

```

```

#include <iostream>
using namespace std;

```

```

int main()
{
    int a, b;
    cout << "Введите делимое: ";
    cin >> a;
    cout << "Введите делитель: ";
    cin >> b;
    if(b) cout << "Результат: " << a / b << '\n';
    else cout << "Деление на ноль недопустимо.\n";

    return 0;
}

```

Обратите внимание на то, что для управления этим предложением `if` достаточно просто переменной `b`. Нет необходимости использовать оператор отношения

Ниже приведены два пробных прогона программы:

Введите делимое: 12
Введите делитель: 2
Результат: 6

Введите делимое: 12
Введите делитель: 0
Деление на ноль недопустимо.

Обратите внимание на то, что значение **b** (делитель) проверяется на ноль с помощью конструкции **if(b)**. Такой метод вполне допустим, так как если **b** равно 0, условие, управляющее предложением **if**, ложно, и выполняется предложение **else**. В противном случае условие истинно (не 0), и выполняется деление. Нет никакой необходимости писать это предложение **if** таким образом:

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

Такая форма предложения избыточна, неэффективна и многими программистами на C++ считается дурным стилем.

Вложенные предложения if

Вложенным называется такое предложение **if**, которое служит мишенью для другого **if** или **else**. Вложенные **if** широко используются в программировании. Главное, что надо помнить о вложенных **if** в C++, это то, что предложение **else** всегда относится к ближайшему к нему предложению **if**, находящемуся в том же блоке, что и **else**, но еще не имеющего своего **else**. Вот пример:

```
if(i)
{
    if(j) result = 1;
    if(k) result = 2;
    else result = 3; // это else относится к if(k)
}
else result = 4; // это else относится к if(i)
```

Как показывают комментарии, последнее **else** не относится к **if(j)**, (хотя оно является ближайшим **if** без своего **else**), потому что оно не находится в том же блоке. Внутреннее **else** относится к **if(k)**, которое является ближайшим к этому **else**.

Вы можете выполнить дальнейшее усовершенствование программы “Магическое число”, добавив в нее вложенное предложе-

ние **if**. Это дополнение сообщит пользователю, в какую сторону он ошибся.

// Программа "Магическое число": 2-е усовершенствование.

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int magic; // магическое число
```

```
    int guess; // догадка пользователя
```

```
    magic = rand(); // получим случайное число
```

```
    cout << " Вводите вашу догадку: ";
```

```
    cin >> guess;
```

```
    if (guess == magic) {
```

```
        cout << "*** Правильно **\n";
```

```
        cout << magic << " и есть магическое число.\n";
```

```
    }
```

```
    else {
```

```
        cout << "...Жаль, но вы ошиблись.";
```

```
        if(guess >> magic) cout << " Ваше число слишком велико.\n";
```

```
        else cout << " Ваше число слишком мало.\n";
```

```
    }
```

```
    return 0;
```

```
}
```

Здесь вложенное предложение **if** предоставляет пользователю дополнительную информацию

Цепочка if-else-if

В программировании широко используется конструкция, основанная на вложенных предложениях **if** и называемая **if-else-if-цепочкой** или **if-else-if-лестницей**. Она выглядит следующим образом:

```
if(условие)
```

```
    предложение;
```

```
else if(условие)
```

```
    предложение;
```

```
else if(условие)
```


```
    предложение;
```

```
.  
.   
.   
else  
    предложение;
```

Условные выражения оцениваются сверху вниз. Как только обнаруживается истинный результат, выполняется предложение, относящееся к этому условию, и оставшаяся часть цепочки пропускается. Если ни одно из условий не дало истинного результата, тогда выполняется последнее предложение **else**. Это последнее предложение часто выступает в качестве действия по умолчанию; другими словами, если все предшествующие условия не удовлетворяются, выполняется это последнее предложение **else**. Если последнего предложения **else** в данной конструкции нет, тогда не выполняется никаких действий.

Приводимая ниже программа демонстрирует цепочку **if-else-if**:

```
// Демонстрация цепочки if-else-if.  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int x;  
  
    for(x=0; x<6; x++) {  
        if(x==1) cout << "x равно единице\n";  
        else if(x==2) cout << "x равно двум\n";  
        else if(x==3) cout << "x равно трем\n";  
        else if(x==4) cout << "x равно четырем\n";  
        else cout << "x не находится между 1 и 4\n";  
    }  
    return 0;  
}
```



Эта программа выводит следующее:

```
x не находится между 1 и 4  
x равно единице  
x равно двум  
x равно трем  
x равно четырем  
x не находится между 1 и 4
```

Легко видеть, что **else** по умолчанию выполняется только если не удовлетворяется ни одно из предшествующих предложений **if**.

Минутная тренировка

1. Условие, управляющее предложением **if**, должно использовать оператор отношения. Правильно или нет?
2. К которому **if** всегда относится **else**?
3. Что представляет собой цепочка **if-else-if**?
 - 1 Неправильно. Условие, управляющее предложением **if**, просто должно давать в результате истину или ложь
 - 2 **else** всегда относится к ближайшему **if** в том же блоке, которое еще не имеет своего **else**
 3. Цепочка **if-else-if** представляет собой последовательность вложенных предложений **if-else**

Цель

3.2. Предложение **switch**

Другой алгоритм выбора в C++ предоставляет предложение **switch**, которое позволяет осуществить выбор требуемого варианта из многих возможных. Другими словами, программа осуществляет выбор нужной альтернативы. То же самое достигается последовательностью вложенных предложений **if**, однако использование **switch** во многих случаях оказывается более эффективным. Работает **switch** таким образом: значение выражения последовательно сравнивается с константами выбора в ветвях **case**. Когда обнаруживается константа, равная значению выражения, выполняется предложение, закрепленное за этой константой. Общая форма предложения **switch** выглядит следующим образом:

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

Анализируемое выражение должно давать в результате либо символ, либо целочисленное значение (выражения с плавающей точкой

недопустимы). Часто в качестве выражения, управляющего выбором, используется просто переменная. Константы в ветвях **case** должны быть либо целыми числами, либо символьными литералами.

Предложение по умолчанию **default** выполняется, если не было найдено константы, соответствующей значению анализируемого выражения. Предложение **default** не является обязательным; при его отсутствии (и если не обнаружено соответствия) никакие действия не выполняются. Если же соответствие обнаружено, выполняются все предложения соответствующей ветви **case**, пока не встретится **break** или, для последней ветви **case**, а также для ветви **default**, пока не закончится вся конструкция **switch**.

Есть четыре существенных момента, которые следует иметь в виду при использовании предложения **switch**:

- **switch** отличается от **if** в том отношении, что **switch** может анализировать выражение только на равенство константам выбора в ветвях **case**, в то время как для **if** условие выполнения может быть любым.
- Никакие две константы в предложении **switch** не могут иметь одно и то же значение. Правда, предложение **switch**, входящее во внешнее предложение **switch**, может иметь в ветвях **case** те же константы.
- Предложение **switch** обычно более эффективно, чем цепочка вложенных **if**.
- Последовательности предложений в ветвях **case** *не есть* блоки. Однако, *все* предложение **switch** образует блок. Важность этого правила станет очевидной по мере изучения вами языка C++.

Приводимая ниже программа демонстрирует использование **switch**. Программа запрашивает у пользователя число между 1 и 3 включительно. После этого программа выводит поговорку, которой присвоен этот номер. В ответ на любое другое число программа выводит сообщение об ошибке.

```
/*
    Простой генератор поговорок, который
    демонстрирует использование предложения switch.
*/

#include <iostream>
using namespace std;

int main()
{
    int num;

    cout << "Введите число от 1 до 3: ";
    cin >> num;
```

```

switch(num) { ← Значение num определяет выполняемую ветвь case
    case 1:
        cout << "Под лежащий камень вода не течет.\n";
        break;
    case 2:
        cout << "Лучше синица в руки, чем журавль в небе.\n";
        break;
    case 3:
        cout << "У дурака деньги долго не держатся.\n";
        break;
    default:
        cout << "Вы можете вводить только 1, 2 или 3.\n";
}
return 0;
}

```

Ниже приведены результаты двух прогонов программы:

Введите число от 1 до 3: 1
Под лежащий камень вода не течет.

Введите число от 1 до 3: 5
Вы можете вводить только 1, 2 или 3.

Формально предложение **break** не обязательно, хотя в большинстве случаев в конструкциях **switch** его приходится использовать. Когда в последовательности предложений в ветви **case** встречается предложение **break**, оно заставляет поток выполнения программы выйти из всего предложения **switch** и перейти на следующее предложение вне **switch**. Если, однако, последовательность ветви **case** не заканчивается предложением **break**, тогда все предложения и выбранной ветви **case**, и всех нижележащих будут выполняться до тех пор, пока не встретится **break** или не закончится все предложение **switch**. Пример такой ситуации демонстрируется в приводимой ниже программе. Можете ли вы сообразить, что она выведет на экран?

```
// switch без предложений break.
```

```

#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;
}
```



Здесь всюду отсутствуют предложения **break**

Эта программа выведет на экран следующее:

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

Мы видим, что если в ветви **case** отсутствует **break**, то продолжают выполняться все последующие ветви **case**.

Допустимо иметь пустые ветви **case**, как это показано ниже:

```
switch(i) {
    case 1:
    case 2: ← Пустые последовательности case
    case 3:
        cout << "i меньше 4";
        break;
    case 4:
        cout << "i равно 4";
        break;
```

В этом фрагменте если `i` имеет значение 1, 2 или 3, то выводится сообщение

`i` меньше 4

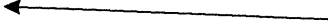
Если же `i` равно 4, то выводится

`i` равно 4

Выстраивание условий `case` “штабелем”, как это показано в последнем примере, является весьма распространенным приемом в тех случаях, когда для нескольких условий выбора должен выполняться один и тот же фрагмент кода.

Вложенные предложения `switch`

Предложение `switch` может входить в последовательность предложений внешнего предложения `switch`. При этом конфликтов не возникает даже если в списках констант `case` внутреннего и внешнего `switch` имеются общие значения. Например, приведенный ниже фрагмент вполне допустим:

```
switch(ch1) {  
  case 'A': cout << "Эта А является частью внешнего switch";  
    switch(ch2) {    
      case 'A':  
        cout << " Эта А является частью внутреннего switch";  
        break;  
      case 'B': // ...  
    }  
    break;  
  case 'B': // ...  
}
```

Минутная тренировка

1. Какого типа должно быть выражение, управляющее предложением `switch`?
2. Что происходит, когда результат выражения в предложении `switch` совпадает с константой `case`?
3. Что происходит, если предложение `case` не завершается предложением `break`?

- 1 Выражение **switch** должно быть целого или символьного типов
- 2 Когда обнаруживается совпадающая с выражением константа **case**, выполняется последовательность предложений, входящих в ветвь соответствующего **case**
- 3 Если предложение **case** не завершается предложением **break**, продолжается выполнение следующей ветви **case**

Спросим у эксперта

Вопрос: При каких условиях в алгоритмах выбора требуемого варианта следует использовать цепочку **if-else-if** вместо предложения **switch**?

Ответ: Как правило, цепочку **if-else-if** удобнее использовать в тех случаях, когда условия, управляющие процессом выбора варианта, не определяются простым значением. Рассмотрим, например, такую последовательность **if-else-if**:

```
if(x < 10) // ...  
else if(y > 0) // ...  
else if(!done) // ...
```

Такую последовательность нельзя преобразовать в предложение **switch**, потому что во всех трех условия анализируют различные переменные и, к тому же, разных типов. Какую переменную выбрать в качестве управляющей для предложения **switch**? Далее, вам придется использовать цепочку **if-else-if**, если условие выбора зависит от значения переменной с плавающей точкой или другого объекта, тип которого недопустим в выражении **switch**.

Проект 3-1

Начинаем строить справочную систему C++

В этом проекте строится простая справочная система, выводящая информацию о синтаксисе управляющих предложений C++. Программа выводит меню, содержащее список управляющих предложений, и затем ожидает, чтобы вы выбрали одно из них. После этого на экран вводится синтаксис соответствующего предложения. В этот первый вариант программы включены только справки по предложениям **if** и **switch**. Другие управляющие предложения будут добавлены в последующих проектах.

Шаг за шагом

1. Создайте новый файл с именем **Help.cpp**.

2. Программа начинает свою работу с вывода следующего меню:

Справка по:

1. if

2. switch

Выберите один из пунктов:

Для реализации пункта 2 вы можете использовать такую последовательность предложений:

```
cout << "Справка по:\n";  
cout << " 1. if\n";  
cout << " 2. switch\n";  
cout << "Выберите один из пунктов: ";
```

3. Далее программа вводит выбранный пользователем номер, как это показано ниже:

```
cin >> choice;
```

4. Получив от пользователя номер, программа выводит на экран соответствующую справку с помощью следующего предложения **switch**:

```
switch(choice) {  
    case '1':  
        cout << "Предложение if:\n\n";  
        cout << "if(условие) предложение;\n";  
        cout << "else предложение;\n";  
        break;  
    case '2':  
        cout << "Предложение switch:\n\n";  
        cout << "switch(выражение) {\n";  
        cout << "  case константа:\n";  
        cout << "    последовательность предложений\n";  
        cout << "    break;\n";  
        cout << " // ... \n";  
        cout << "}\n";  
        break;  
    default:  
        cout << "Этот пункт отсутствует.\n";  
}
```

Обратите внимание на то, как предложение **default** отбирает недопустимые номера. Если, например, пользователь ввел 3, этому числу нет соответствия в ветвях **case**, что и приведет к выполнению последовательности **default**.

5. Ниже приведен полный текст программы **Help.cpp**:

```
/*
    Проект 3-1

    Простая справочная система.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    cout << "Справка по:\n";
    cout << "  1. if\n";
    cout << "  2. switch\n";
    cout << "Выберите один из пунктов: ";
    cin >> choice;

    cout << "\n";

    switch(choice) {
        case '1':
            cout << "Предложение if:\n\n";
            cout << "if(условие) предложение;\n";
            cout << "else предложение;\n";
            break;
        case '2':
            cout << "Предложение switch:\n\n";
            cout << "switch(выражение) {\n";
            cout << "  case константа:\n";
            cout << "    последовательность предложений\n";
            cout << "    break;\n";
            cout << "  // ... \n";
            cout << "  }\n";
            break;
        default:
            cout << "Этот пункт отсутствует.\n";
    }

    return 0;
}
```

Ниже приведен пример работы программы.:

Справка по:
1. if
2. switch

Выберите один из пунктов: 1

Предложение if:

```
if(условие) предложение;  
else предложение;
```

Цель

3.3. Цикл for

Начиная с Модуля 1 вы уже использовали простую форму цикла **for**. Этот цикл обладает удивительными возможностями и гибкостью. Рассмотрим основные черты этой конструкции, начав с ее наиболее традиционной формы.

Общая форма цикла **for** для повторения единственного предложения выглядит так:

```
for(инициализация; выражение; приращение) предложение;
```

Для повторения блока предложений общая форма будет такой:

```
for(инициализация; выражение; приращение)  
{  
    последовательность предложений  
}
```

Инициализация обычно представляет собой предложение присваивания, которое устанавливает начальное значение *переменной управления циклом*, действующей в качестве счетчика шагов цикла. *Выражение* является условным выражением, определяющим, будут ли повторяться далее шаги цикла. *Приращение* определяет величину, на которую будет изменяться переменная управления циклом в каждом шаге цикла. Обратите внимание на то, что эти три основные секции цикла должны разделяться символами точки с запятой. Цикл **for** будет продолжать свое выполнение, пока условное выражение истинно. Как только это выражение станет ложным, произойдет выход из цикла, и выполнение программы продолжится с предложения, следующего за блоком **for**.

Приводимая ниже программа использует цикл **for** для вывода значений квадратных корней из чисел от 1 до 99. В этом примере переменная управления циклом названа **num**.

```
// Показывает квадратные корни из чисел от 1 до 99.
```

```
#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;
}
```

Программа использует стандартную функцию `sqrt()`. Как уже говорилось в Модуле 2, функция `sqrt()` возвращает квадратный корень из передаваемого ей аргумента. Аргумент должен быть типа **double**, и функция возвращает значение также типа **double**. Для вызова функции требуется заголовок `<cmath>`.

Переменная управления циклом может изменяться как в положительном, так и в отрицательном направлении, причем на любую величину. В качестве примера следующая программа выводит числа от 50 до -50 с интервалом 10:

```
// Цикл, выполняемый в отрицательном направлении.

#include <iostream>
using namespace std;

int main()
{
    int i;

    for(i=50; i >= -50; i = i-10) cout << i << ' ' ;
    return 0;
}
```

Цикл, выполняемый в отрицательном направлении

Вывод программы выглядит следующим образом:

```
50 40 30 20 10 0 -10 -20 -30 -40 -50
```

При использовании циклов важно иметь в виду, что условное выражение цикла всегда анализируется в начале шага. Таким образом, код внутри цикла может не выполниться ни одного раза, если

условие с самого начала оказывается ложным. Вот пример такой ситуации:

```
for(count=10; count < 5; count++)
    cout << count; // это предложение не выполнится
```

Этот цикл выполняться не будет, так как управляющая переменная **count** при входе в цикл сразу оказывается больше 5. Поэтому условное выражение **count < 5** ложно с самого начала и в результате не будет выполнено ни одного шага цикла.

Некоторые варианты цикла for

Цикл **for** является одним из наиболее гибких предложений в языке C++, потому что он допускает большое количество разнообразных вариантов. Например, допустимо использовать несколько переменных управления. Рассмотрим следующий фрагмент кода:

```
for(x=0, y=10; x <= y; ++x, --y) ← Использование нескольких пе-
    cout << x << ' ' << y << '\n';          переменных управления циклом
```

Здесь запятыми разделены два инициализирующих предложения и два выражения наращивания. Запятые необходимы, так как они показывают компилятору, что в этом цикле по два предложения инициализации и наращивания. В C++ запятая представляет собой оператор, который означает “сделай это и это”. Запятая широко используется, в частности, в циклах **for**. Допустимо иметь любое число предложений инициализации и наращивания, хотя, если их больше двух-трех, цикл становится слишком неуклюжим.

Спросим у эксперта

Вопрос: Поддерживает ли C++, кроме **sqrt()**, другие математические функции?

Ответ: Да! В дополнение к **sqrt()**, C++ поддерживает широкий набор библиотечных математических функций, например, **sin()**, **cos()**, **tan()**, **log()**, **pow()**, **ceil()**, **floor()** и много других. Если вас интересуют математические программы, вам следует познакомиться с набором математических функций C++. Все компиляторы C++ поддерживают эти функции, и их описание можно найти в документации к вашему компилятору. Математические функции требуют включения в программу заголовка **<cmath>**.

Условное выражение, управляющее циклом, может представлять собой любое допустимое выражение C++. Оно совсем не обязательно

должно использовать переменную управления циклом. В следующем примере цикл продолжает выполняться до тех пор, пока функция **rand()** не вернет число, большее, чем 20000.

```
/*
    Цикл выполняется, пока случайное число не окажется
    больше, чем 20000.
*/

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

int main()
{
    int i;
    int r;

    r = rand();

    for(i=0; r <= 20000; i++) ← Это условное выражение не использует переменную управления циклом
        r = rand();

    cout << "Число равно " << r <<
        ". Оно было получено на шаге " << i << ".";

    return 0;
}
```

Вот пример вывода программы:

Число равно 26500. Оно было получено на шаге 3.

В каждом шаге цикла вызовом **rand()** генерируется новое случайное число. Когда будет получено число, большее 20000, условие цикла становится ложным и цикл прекращает свою работу.

Опущенные секции

Еще одно свойство цикла **for**, отличающее C++ от многих других компьютерных языков, заключается в возможности опускать отдельные секции определения цикла. Если, например, вы хотите описать цикл, который должен выполняться до тех пор, пока с клавиатуры не будет введено число 123, это можно сделать таким образом:

```
// Цикл без приращения.

#include <iostream>
using namespace std;

int main()
{
    int x;

    for(x=0; x != 123; ) { ← Выражение приращения отсутствует.
        cout << "Введите число: ";
        cin >> x;
    }

    return 0;
}
```

В этом определении цикла **for** отсутствует секция, характеризующая приращение. Это значит, что в каждом шаге цикла выполняется проверка, не равна ли переменная *x* числу 123, но больше никаких действий не предпринимается. Если, однако, вы вводите с клавиатуры 123, условие выполнения цикла становится ложным и цикл прекращает свое выполнение. Если в определении цикла отсутствует секция, описывающая приращение, то при выполнении цикла управляющая им переменная изменяться не будет.

Другой вариант цикла **for** можно получить, если переместить секцию инициализации за пределы цикла, как это показано в приводимом ниже фрагменте:

```
x = 0; ← x инициализируется вне цикла

for( ; x<10; )
{
    cout << x << ' ';
    ++x;
}
```

Здесь опущена секция инициализации, а *x* инициализируется перед входом в цикл. Размещение инициализирующих действий вне цикла обычно выполняется лишь в тех случаях, когда алгоритм вычисления начального значения настолько сложен, что его нельзя разместить внутри предложения **for**. Обратите внимание на то, что в приведенном примере секция приращения размещена внутри тела цикла.

Бесконечный цикл for

Вы можете создать *бесконечный цикл* (т. е. цикл, который никогда не прекращает своего выполнения), использовав такую конструкцию:

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

Такой цикл будет выполняться бесконечно. Хотя в принципе существуют программистские задачи (например, командный процессор операционной системы), которые требуют организации бесконечного цикла, большинство “бесконечных циклов” являются всего лишь циклами со специальными условиями завершения. В конце этого модуля вы узнаете, как можно остановить цикл такого рода. (Подсказка: это делается с помощью предложения **break**.)

Цикл с отсутствующим телом

В C++ допустимы циклы **for**, в которых отсутствует само тело цикла. Такая конструкция возможна потому, что синтаксически допустимо *пустое предложение*. “Бестелесные” циклы оказываются часто весьма полезными. Например, приведенная ниже программа использует такой цикл для получения суммы чисел от 1 до 10:

// Тело цикла может быть пустым.

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

```
int main()
{
```

```
    int i;
    int sum = 0;
```

```
    // суммируем числа от 1 до 10
```

```
    for(i=1; i <= 10; sum += i++) ;
```

← Этот цикл не имеет тела.

```
    cout << "Сумма равна " << sum;
```

```
    return 0;
}
```

Вывод программы выглядит так:

Сумма равна 55

Заметьте, что процесс суммирования выполняется целиком внутри предложения **for**, и в теле цикла нет необходимости. Обратите особое внимание на выражение приращения:

```
sum += i++
```

Не бойтесь предложений такого рода. В профессиональных программах на C++ они вполне обычны; чтобы разобраться в смысле такого предложения его следует разбить на составные части. Это предложение, выраженное словами, обозначает “прибавьте к **sum** значение **sum** плюс **i**, затем выполните инкремент **i**”. Таким образом, оно делает то же, что и следующая последовательность предложений:

```
sum = sum + i;  
i++;
```

Объявление переменных управления циклом внутри цикла **for**

Часто переменная, управляющая циклом **for**, требуется только внутри этого цикла и не используется нигде в другом месте. В этом случае можно объявить управляющую переменную в инициализирующей секции предложения **for**. Такой прием демонстрируется в приводимой ниже программе, которая вычисляет сумму чисел от 1 до 5 и факториал числа 5:

```
// Объявление переменной управления циклом внутри  
// предложения for.  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    int sum = 0;  
    int fact = 1;  
  
    // вычислим факториал числа 5  
    for(int i = 1; i <= 5; i++) { ← здесь i объявлена внутри цикла for  
        sum += i; // i известна всюду внутри цикла  
        fact *= i;  
    }  
}
```

```
// однако здесь i неизвестна.  
  
cout << "Сумма равна " << sum << "\n";  
cout << "Факториал равен " << fact;  
  
return 0;  
}
```

Вывод этой программы:

Сумма равна 15
Факториал равен 120

Если вы объявляете переменную внутри цикла **for**, то вы должны иметь в виду одно важное обстоятельство: эта переменная будет известна только внутри предложения **for**. Другими словами, используя терминологию языков программирования, *область видимости* переменной ограничена циклом **for**. Вне цикла **for** эта переменная просто не существует. Поэтому в предыдущем примере к *i* нельзя обратиться за пределами цикла **for**. Если вам нужно использовать переменную управления циклом где-либо еще в программе, вы не должны объявлять ее внутри цикла **for**.



Замечание

Область видимости переменной, объявленной в секции инициализации цикла **for**, изменялась с течением времени. Первоначально эта переменная была доступна и после цикла **for**, но в процессе стандартизации C++ положение изменилось. Современный C++ по стандарту ANSI/ISO ограничивает область видимости управляющей переменной только циклом **for**. Некоторые компиляторы, однако, не накладывают на нее этого ограничения. Вам следует выяснить, каковы правила в отношении этой переменной в той среде, в которой вы работаете.

Перед тем, как приступить к изучению дальнейшего материала, вам стоит поэкспериментировать с различными вариантами цикла **for**. Вы увидите, что эта конструкция полна скрытых возможностей.

Минутная тренировка

1. Могут ли секции предложения **for** быть пустыми?
2. Покажите, как создать бесконечный цикл **for**.
3. Какова область видимости переменной, объявленной в предложении **for**?

1. Да. Все три части предложения **for** — инициализация, условие и наращивание — могут быть пустыми
 2. `for(; ;)`
 3. Область видимости переменной, объявленной внутри предложения **for**, ограничена циклом. Вне цикла эта переменная неизвестна
-

Цель

3.4. Цикл **while**

Другая возможность организовать повторные действия предоставляется циклом **while**. Общая форма этого цикла выглядит следующим образом:

```
while(выражение) предложение;
```

Здесь *предложение* может быть одиночным предложением или блоком предложений. *Выражение* определяет условия, управляющие циклом, и может быть любым допустимым выражением. Предложение выполняется, пока условие остается истинным. Когда условие становится ложным, программное управление передается на строку, непосредственно следующую за циклом.

Приводимая ниже программа иллюстрирует использование цикла **while** для реализации весьма любопытного алгоритма. Практически все компьютеры поддерживают набор символов, расширенный по сравнению с обычным набором ASCII. Расширенные символы, если они существуют, часто включают такие специальные знаки, как символы иностранных языков и научные обозначения. Символы ASCII имеют значения меньше, чем 128. Расширенный набор включает символы начиная с кода 128 и до 255. Наша программа выводит на экран все символы от 32 (пробел) до 255. Запустив программу, вы увидите некоторые очень интересные символы.

```
/*
```

```
 Эта программа выводит все отображаемые на экране символы,
 включая расширенный символьный набор, если он существует.
```

```
*/
```

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

```
int main()
{
```

```
    unsigned char ch;
```

```
    ch = 32;
```

```
while(ch) { ← Использование цикла while.
    cout << ch;
    ch++;
}

return 0;
}
```

Рассмотрим выражение условия выполнения цикла приведенной выше программы. Может вызвать недоумение, почему для управления циклом **while** используется просто переменная **ch**. Поскольку **ch** объявлена как символ без знака, она может содержать значения от 0 до 255. Когда значение этой переменной становится равным 255 и затем еще увеличивается на 1, происходит явление “оборачивания” и **ch** становится равным 0. В результате проверка **ch** на равенство нулю может служить удобным условием окончания цикла.

Как и в случае цикла **for**, цикл **while** анализирует условное выражение в начале каждого шага, из чего следует, что тело цикла может не выполниться ни разу. Приводимая ниже программа иллюстрирует эту характеристику цикла **while**. Программа выводит линию точек. Число точек определяется значением, введенным пользователем. Программа не допускает вывод строк длиннее 80 символов.

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

int main()
{
    int len;

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

    return 0;
}
```

Если **len** выходит за установленные границы, цикл **while** не выполняется ни разу. В противном случае цикл выполняется, пока **len** не достигнет 0.

Тело цикла **while** может не содержать ни одного предложения. Вот пример такого цикла:

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

Этот цикл повторяется, пока случайное число, генерируемое функцией **rand()**, не окажется равным 100.

Цель

3.5. Цикл **do-while**

Последняя конструкция цикла в C++ — цикл **do-while**. В отличие от циклов **for** и **while**, в которых условие выполнения анализируется в начале каждого шага, в цикле **do-while** проверка условия выполняется в конце шага цикла. Отсюда следует, что цикл **do-while** всегда выполняется по меньшей мере 1 раз. Общая форма цикла **do-while** выглядит следующим образом:

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

Если тело цикла состоит из единственного предложения, в фигурных скобках нет необходимости. Однако они часто используются для повышения наглядности конструкции **do-while**. Цикл **do-while** выполняется до тех пор, пока условное выражение истинно.

В следующей программе цикл выполняется, пока пользователь не введет число 100:

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

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

Цикл **do-while** всегда выполняется по меньшей мере один раз.

С помощью цикла **do-while** мы можем выполнить дальнейшее усовершенствование программы “Магическое число”. На этот раз

программа “крутится” в цикле, пока вы не угадаете правильное число.

// Программа “Магическое число”: 3-е усовершенствование.

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

int main()
{
    int magic; // магическое число
    int guess; // догадка пользователя

    magic = rand(); // получим случайное число
    do {
        cout << "Вводите вашу догадку: ";
        cin >> guess;
        if(guess == magic) {
            cout << "*** Правильно ** ";
            cout << magic << " и есть магическое число.\n";
        }
        else {
            cout << "...Жаль, но вы ошиблись.";
            if(guess > magic)
                cout << " Ваше число слишком велико.\n";
            else cout << " Ваше число слишком мало.\n";
        }
    } while(guess != magic);

    return 0;
}
```

Вот результат пробного прогона:

```
Вводите вашу догадку: 10
...Жаль, но вы ошиблись. Ваше число слишком мало.
Вводите вашу догадку: 100
...Жаль, но вы ошиблись. Ваше число слишком велико.
Вводите вашу догадку: 50
...Жаль, но вы ошиблись. Ваше число слишком велико.
Вводите вашу догадку: 41
** Правильно ** 41 и есть магическое число.
```

Последнее замечание: как и для циклов **for** и **while**, в цикле **do-while** тело цикла может быть пустым, хотя практически этой возможностью пользуются редко.

Минутная тренировка

1. В чем основное различие циклов **while** и **do-while**?
 2. Может ли быть пустым тело цикла **while**?
1. В цикле **while** условие проверяется в начале каждого шага цикла. В цикле **do-while** это делается в конце шага. Таким образом, цикл **do-while** всегда выполняется по меньшей мере один раз.
 2. Да, тело цикла **while** (или любого другого цикла C++) может быть пустым.
-

Спросим у эксперта

Вопрос: Учитывая большую гибкость всех циклов C++, по какому критерию я должен выбирать форму цикла? Другими словами, как мне выбрать для решения конкретной задачи правильную форму цикла?

Ответ: Если вам надо выполнить известное число шагов, используйте цикл **for**. Если вам требуется, чтобы во всех случаях выполнялся хотя бы один шаг цикла, используйте цикл **do-while**. Цикл **while** наиболее естественно использовать в тех случаях, когда число шагов цикла заранее неизвестно.

Проект 3-2

Усовершенствование справочной системы C++

В этом проекте выполняется усовершенствование справочной системы, созданной в Проекте 3-1. В данном варианте в справочную систему заносится информация о циклах **for**, **while** и **do-while**. Кроме этого, в программе проверяется номер пункта, выбранного пользователем, и в случае неверного номера запрос на ввод повторяется до тех пор, пока не будет введено допустимое число. Для этого используется цикл **do-while**. Стоит отметить, что использование цикла **do-while** для выбора пункта меню является распространенной практикой, потому что такое действие всегда надо выполнять по меньшей мере один раз.

Шаг за шагом

1. Скопируйте файл **Help.cpp** в новый файл с именем **Help2.cpp**.
2. Измените часть программы, выводющую возможные варианты, так, чтобы она использовала показанный ниже цикл **do-while**:

```
do {
    cout << "Справка по:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << " Выберите один из пунктов: ";

    cin >> choice;

} while( choice < '1' || choice > '5');
```

После внесения этого изменения программа будет “крутиться” в цикле, выводя меню, до тех пор, пока пользователь не введет число между 1 и 5. Как видите, цикл **do-while** в данном случае оказывается весьма полезен.

3. Расширьте предложение **switch**, включив в него справки по циклам **for**, **while** и **do-while**, как это показано ниже:

```
switch(choice) {
    case '1':
        cout << "Предложение if:\n\n";
        cout << "if(условие) предложение;\n";
        cout << "else предложение;\n";
        break;
    case '2':
        cout << "Предложение switch:\n\n";
        cout << "switch(выражение) {\n";
        cout << "    case константа;\n";
        cout << "        последовательность предложений\n";
        cout << "        break;\n";
        cout << "    // ... \n";
        cout << "};\n";
        break;
    case '3':
        cout << "Цикл for:\n\n";
        cout << "for(инициализация; условие; приращение) ";
        cout << " предложение;\n";
        break;
    case '4':
        cout << "Цикл while:\n\n";
        cout << "while(условие) предложение;\n";
        break;
    case '5':
        cout << "Цикл do-while:\n\n";
```

```

cout << "do {\n";
cout << " предложение;\n";
cout << "} while (условие);\n";
break;
}

```

Заметьте, что в этом варианте в предложении **switch** отсутствует ветвь **default**. Поскольку цикл вывода меню обеспечивает ввод пользователем только допустимого номера пункта, отпадает необходимость включать предложение **default** для обработки неправильного ввода.

4. Ниже приводится полный текст программы **Help2.cpp**.

```

/*
Проект 3-2

Усовершенствованная справочная система,
использующая цикл do-while для выбора пункта меню.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;
    do {
        cout << "Справка по:\n";
        cout << " 1. if\n";
        cout << " 2. switch\n";
        cout << " 3. for\n";
        cout << " 4. while\n";
        cout << " 5. do-while\n";
        cout << " Выберите один из пунктов: ";

        cin >> choice;

    } while( choice < '1' || choice > '5');
    cout << "\n\n";
    switch(choice) {
        case '1':
            cout << "Предложение if:\n\n";
            cout << "if(условие) предложение;\n";
            cout << "else предложение;\n";
            break;
        case '2':
            cout << "Предложение switch:\n\n";

```

```
    cout << "switch(выражение) {\n";
    cout << "    case константа:\n";
    cout << "        последовательность предложений\n";
    cout << "        break;\n";
    cout << "    // ... \n";
    cout << "}\n";
    break;
case '3':
    cout << "Цикл for:\n\n";
    cout << "for(инициализация; условие; приращение) ";
    cout << " предложение;\n";
    break;
case '4':
    cout << "Цикл while:\n\n";
    cout << "while(условие) предложение;\n";
    break;
case '5':
    cout << "Цикл do-while:\n\n";
    cout << "do {\n";
    cout << "    предложение;\n";
    cout << "} while (условие);\n";
    break;
}

return 0;
}
```

Спросим у эксперта

Вопрос: Ранее вы объяснили, как можно объявить переменную в секции инициализации цикла **for**. Можно ли объявлять переменные внутри других управляющих предложений C++?

Ответ: Да. В C++ возможно объявить переменную внутри условного выражения в предложениях **if** или **switch**, внутри условного выражения в цикле **while**, а также внутри секции инициализации цикла **for**. Переменная, объявленная в одном из этих мест, имеет область видимости, ограниченную блоком кода, которым управляет данное предложение.

Вы уже видели пример объявления переменной внутри цикла **for**. Вот пример объявления переменной внутри предложения **if**:

```
if(int x = 20) {
    x = x - y;
    if(x > 10) y = 0;
}
```

В предложении **if** объявляется переменная **x**, и ей присваивается значе-

ние 20. Поскольку это значение истинно, мишень **if** выполняется. Поскольку переменные, объявленные в условном предложении, имеют область видимости, ограниченную блоком кода, управляемого этим предложением, **x** становится неизвестным за пределами **if**.

Как уже отмечалось при обсуждении цикла **for**, будет ли переменная, объявленная внутри управляющего предложения, известна только в этом предложении, или она будет доступна и после этого предложения, зависит от компилятора. Перед тем, как использовать такие переменные, вам следует свериться с документацией к вашему компилятору. Разумеется, стандарт C++ ANSI/ISO оговаривает, что переменная известна только внутри того предложения, в котором она объявлена.

Большинство программистов не объявляют переменные внутри каких-либо управляющих предложений, за исключением цикла **for**. Вообще объявление переменных внутри других предложений является спорной практикой, и некоторые программисты считают это дурным тоном.

Цель

3.6. Использование **break** для выхода из цикла

С помощью предложения **break** имеется возможность выполнить немедленный выход из цикла, обойдя проверку условия. Если внутри цикла встречается предложение **break**, цикл немедленно завершается, и управление передается на предложение, следующее за циклом. Вот простой пример:

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

int main()
{
    int t;

    // Цикл от 0 до 9, не до 100!
    for(t=0; t<100; t++) {
        if(t==10) break; ← Выход из цикла for, когда t становится равным 10
        cout << t << ' ';
    }

    return 0;
}
```

Вывод этой программы:

0 1 2 3 4 5 6 7 8 9

Как видно из вывода программы, она отображает на экране числа от 0 до 9 и после этого завершается. Программа не доходит до 100, потому что предложение **break** заставляет цикл завершиться раньше.

Если циклы вложены друг в друга (т. е. когда один цикл включает в себя другой), **break** осуществит выход только из самого внутреннего цикла. Вот пример этого:

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

int main()
{
    int t, count;

    for(t=0; t<10; t++) {
        count = 1;
        for(;;) {
            cout << count << ' ';
            count++;
            if(count==10) break;
        }
        cout << '\n';
    }

    return 0;
}
```

Вот вывод этой программы:

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

Как вы видите, программа выводит на экран числа от 1 до 9 десять раз. Каждый раз, когда во внутреннем цикле **for** встречается **break**, управление передается внешнему циклу **for**. Обратите внимание на то, что внутренний цикл **for** является бесконечным циклом, который завершается с помощью предложения **break**.

Предложение **break** может быть использовано с любым предложением цикла. Оно особенно удобно, когда некоторое определенное ус-

ловие должно привести к немедленному завершению цикла. Одним из примеров такой методики является завершение бесконечных циклов, что было проиллюстрировано в предыдущем примере.

Еще одно замечание: **break** в предложении **switch** будет влиять на выполнение только этого предложения **switch**, но не цикла того или иного рода, в который это предложение может входить.

Цель

3.7. Использование **continue**

Существует возможность принудительно завершить некоторый шаг цикла, обойдя нормальную управляющую структуру цикла. Это действие выполняется с помощью предложения **continue**. Это предложение заставляет цикл перейти к следующему шагу, обойдя любой код между **continue** и условным выражением, управляющим циклом. Пример этого можно увидеть в следующей программе, которая выводит на экран четные числа от 0 до 100:

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

int main()
{
    int x;
    for(x=0; x<=100; x++) {
        if(x%2) continue; ← Досрочно завершить шаг, если x нечетно
        cout << x << ' ';
    }

    return 0;
}
```

Выводятся только четные числа, потому что нечетное число вызывает выполнение предложения **continue**, которое приводит к выходу из этого шага цикла с обходом предложения вывода **cout**. Вспомните, что оператор **%** возвращает остаток от целочисленного деления. Когда **x** нечетно, остаток равен 1, т. е. результат деления истинен. Если **x** четно, остаток равен 0, т. е. результат ложен.

В циклах **while** и **do-while** предложение **continue** вызывает передачу управления непосредственно на условное предложение, после чего цикл продолжается обычным образом. В случае **for** выполняется секция приращения цикла, затем условное выражение, после чего цикл продолжается.

Минутная тренировка

1. Что происходит внутри цикла, когда выполняется предложение **break**?

2. К чему приводит выполнение предложения `continue`?
 3. В случае, когда в цикл входит предложение `switch`, приводит ли предложение `break` в этом `switch` к завершению цикла?
1. Предложение `break` внутри цикла приводит к немедленному завершению цикла. Выполнение продолжается с первой строки кода после цикла.
 2. Предложение `continue` приводит к завершению данного шага цикла с обходом оставшегося в нем кода.
 3. Нет.
-

Проект 3-3

Завершаем разработку справочной системы C++

Проект накладывает завершающие штрихи на справочную систему C++, разработанную в предыдущем проекте. В этом варианте в систему добавляется справка по предложениям `break`, `continue` и `goto`. Программа также позволяет пользователю запросить справку по нескольким предложениям языка. С этой целью в программу добавлен внешний цикл, который выполняется до тех пор, пока пользователь в ответ на предложение программы выбрать пункт меню не введет `q`.

Шаг за шагом

1. Скопируйте `Help2.cpp` в новый файл с именем `Help3.cpp`.
2. Окружите весь программный код бесконечным циклом `for`. Выйдите из этого цикла посредством предложения `break` в случае ввода `q`. Поскольку этот цикл окружает весь остальной код, выход из него заставляет завершиться всю программу.
3. Измените цикл меню, как это показано ниже:

```
do {
    cout << "Справка по:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << " 6. break\n";
    cout << " 7. continue\n";
    cout << " 8. goto\n";
    cout << "Выберите один из пунктов (q для завершения): ";
    cin >> choice;
} while( choice < '1' || choice > '8' && choice != 'q');
```

Заметьте, что этот цикл включает предложения **break**, **continue** и **goto**. Он также позволяет ввести **q**, как один из допустимых символов.

4. Расширьте предложение **switch**, включив в него предложения **break**, **continue** и **goto**, как это показано ниже:

```
case '6':
    cout << "Предложение break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "Предложение continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "Предложение goto:\n\n";
    cout << "goto метка;\n";
    break;
```

5. Ниже приведен полный текст программы **Help3.cpp**:

```
/*
    Проект 3-3
    Законченная справочная система, обслуживающая
    повторные запросы.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    for(;;) {
        do {
            cout << "Справка по:\n";
            cout << " 1. if\n";
            cout << " 2. switch\n";
            cout << " 3. for\n";
            cout << " 4. while\n";
            cout << " 5. do-while\n";
            cout << " 6. break\n";
            cout << " 7. continue\n";
            cout << " 8. goto\n";
            cout << "Выберите один из пунктов (q для завершения): ";
            cin >> choice;
        } while( choice < '1' || choice > '8' && choice != 'q');
```

```
    if(choice == 'q') break;

cout << "\n\n";

switch(choice) {
    case '1':
        cout << "Предложение if:\n\n";
        cout << "if(условие) предложение;\n";
        cout << "else предложение;\n";
        break;
    case '2':
        cout << "Предложение switch:\n\n";
        cout << "switch(выражение) {\n";
        cout << " case константа:\n";
        cout << "  последовательность предложений\n";
        cout << "  break;\n";
        cout << " // ... \n";
        cout << "}\n";
        break;
    case '3':
        cout << "Цикл for:\n\n";
        cout << "for(инициализация; условие; приращение) ";
        cout << " предложение;\n";
        break;
    case '4':
        cout << "Цикл while:\n\n";
        cout << "while(условие) предложение;\n";
        break;
    case '5':
        cout << "Цикл do-while:\n\n";
        cout << "do {\n";
        cout << "  предложение;\n";
        cout << "} while (условие);\n";
        break;
    case '6':
        cout << "Предложение break:\n\n";
        cout << "break;\n";
        break;
    case '7':
        cout << "Предложение continue:\n\n";
        cout << "continue;\n";
        break;
    case '8':
        cout << "Предложение goto:\n\n";
        cout << "goto метка;\n";
        break;
```

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

6. Ниже приведен пример работы программы:

Справка по:

1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue
8. goto

Выберите один из пунктов (q для завершения): 1

Предложение if:

```
if(условие) предложение;  
else предложение;
```

Справка по:

1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue
8. goto

Выберите один из пунктов (q для завершения): 6

Предложение break:

```
break;
```

Справка по:

1. if
2. switch
3. for
4. while
5. do-while
6. break

7. continue

8. goto

Выберите один из пунктов (q для завершения): q

Цель

3.8. Вложенные циклы

Как вы уже видели в некоторых из приведенных выше примеров, один цикл может быть вложен внутрь другого. Вложенные циклы решают широкий спектр программистских задач и являются существенным элементом программирования. Поэтому перед тем, как завершить обсуждение циклов C++, давайте рассмотрим еще один пример вложенного цикла. В приводимой ниже программе вложенный цикл **for** используется для нахождения множителей чисел от 2 до 100:

```
/*
   Использование вложенных циклов для нахождения множителей
   чисел от 2 до 100.
*/

#include <iostream>
using namespace std;

int main() {

    for(int i=2; i <= 100; i++) {
        cout << "Множители числа " << i << ": ";

        for(int j = 2; j < i; j++)
            if((i%j) == 0) cout << j << " ";

        cout << "\n";
    }

    return 0;
}
```

Ниже приведена часть вывода программы:

```
Множители числа 2:
Множители числа 3:
Множители числа 4: 2
Множители числа 5:
Множители числа 6: 2 3
Множители числа 7:
```



```
Множители числа 8: 2 4
Множители числа 9: 3
Множители числа 10: 2 5
Множители числа 11:
Множители числа 12: 2 3 4 6
Множители числа 13:
Множители числа 14: 2 7
Множители числа 15: 3 5
Множители числа 16: 2 4 8
Множители числа 17:
Множители числа 18: 2 3 6 9
Множители числа 19:
Множители числа 20: 2 4 5 10
```

В этой программе внешний цикл проходит значения *i* от 2 до 100. Внутренний цикл последовательно анализирует все числа от 2 до *i*, выводя те, на которые *i* делится без остатка.

Цель

3.9. Использование предложения `goto`

Предложение C++ `goto` реализует безусловный переход. Когда это предложение встречается в программе, управление передается в заданную в нем точку. Предложение `goto` уже много лет как потеряло благосклонность программистов, так как его использование способствует образованию “макаронных” программ. Однако `goto` все еще время от времени используется и иногда весьма эффективно. Мы здесь не будем выносить приговор относительно ценности этого предложения в качестве средства управления программой. Следует, однако, заметить, что не существует программных ситуаций, которые требовали бы обязательного использования предложения `goto`; оно не является необходимым в законченном языке. Скорее его можно рассматривать, как некоторое дополнение, которое, если им пользоваться со смыслом, принесет пользу в определенных программных ситуациях. По этим причинам `goto` нигде, кроме настоящего раздела, не используется в примерах этой книги. Программисты недолюбливают `goto` главным образом потому, что оно имеет тенденцию запутывать программу, делая ее почти нечитаемой. Однако в некоторых случаях использование `goto` не усложняет программу, а, наоборот, проясняет ее структуру.

В предложении `goto` требуется указание метки. *Метка* — это допустимый идентификатор C++, сопровождаемый двоеточием. Метка

должна находиться в той же функции, что и использующее ее **goto**. Например, цикл от 1 до 100 можно организовать с помощью **goto** и метки:

```
x = 1;
loop1: ←—————|
  x++;
  if (x < 100) goto loop1; —————→
```

Выполнение передается в точку **loop1**.

Один из разумных примеров использования **goto** — выход из глубоко вложенного программного блока. Рассмотрим, например, такой фрагмент кода:

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

Устранение **goto** потребовало бы выполнения ряда дополнительных проверок. Простое предложение **break** здесь не будет работать, потому что оно обеспечивает лишь выход программы из самого внутреннего цикла.

✓ Вопросы для самопроверки

1. Напишите программу, которая вводит символы с клавиатуры до нажатия \$. Пусть программа считает число введенных точек. Перед завершением программы выведите на экран это число.
2. Может ли в предложении **switch** возникнуть ситуация, когда после выполнения одной ветви **case** продолжится выполнение следующей? Объясните.
3. Напишите общую форму цепочки **if-else-if**.
4. Рассмотрите следующий программный фрагмент:

```
if (x < 10)
  if (y > 100) {
    if (!done) x = z;
    else y = z;
  }
else cout << "ошибка"; // какое if?
```

К какому **if** относится последнее **else**?

5. Напишите предложение **for** для цикла, который проходит значения от 1000 до 0 с приращением -2 .
6. Допустим ли следующий фрагмент?

```
for(int i = 0; i < num; i++)  
    sum += i;
```

```
count = i;
```

7. Объясните, как выполняется предложение **break**.
8. Рассмотрите приведенный ниже программный фрагмент. Что будет выведено на экран после выполнения предложения **break**?

```
for(i = 0; i < 10; i++) {  
    while(running) {  
        if(x<y) break;  
        // ...  
    }  
    cout << "после while\n";  
}  
cout << "После for\n";
```

9. Что выведет на экран следующий программный фрагмент?

```
for(int i = 0; i<10; i++) {  
    cout << i << " ";  
    if(!(i%2)) continue;  
    cout << "\n";  
}
```

10. Выражение приращения в цикле **for** не обязательно должно изменять значение переменной управления циклом на фиксированную величину. Напротив, переменная управления циклом может изменяться произвольным образом. Используя это обстоятельство, напишите программу, которая с помощью цикла **for** вычисляет и выводит на экран геометрическую прогрессию 1, 2, 4, 8, 16, 32 и т. д.
11. Коды строчных латинских букв в таблице ASCII отличаются от кодов прописных букв на величину 32. Таким образом, для преобразования строчных букв в прописные из код буквы следует вычесть 32. Используя это обстоятельство, напишите программу, которая вводит символы с клавиатуры и преобразует строчные буквы в прописные, а прописные — в строчные, выводя результат на экран. Не изменяйте никакие другие символы. Программа должна завершиться при вводе символа точки, а перед завершением вывести число выполненных изменений.
12. Как выглядит предложение безусловного перехода в C++?

Модуль 4 Массивы, строки и указатели

Цели, достигаемые в этом модуле

- 4.1 Научиться использовать одномерные массивы
- 4.2 Познакомиться с двумерными массивами
- 4.3 Познакомиться с многомерными массивами
- 4.4 Разобраться с понятием строки
- 4.5 Изучить стандартные функции для обработки строк
- 4.6 Рассмотреть инициализацию массивов
- 4.7 Познакомиться с массивами строк
- 4.8 Освоить основы техники указателей
- 4.9 Научиться работать с операторами указателей
- 4.10 Начать использовать указатели в выражениях
- 4.11 Изучить взаимосвязь указателей и массивов
- 4.12 Познакомиться с понятием вложенной косвенности

В этом модуле обсуждаются массивы, строки и указатели. Хотя эти темы могут показаться несвязанными, в действительности это не так. В C++ эти понятия тесно переплетены, и чтобы разобраться в одном, необходимо изучить и остальные.

Массивом называется коллекция переменных одного типа, обращение к которым осуществляется по общему имени. Массивы могут быть одномерными или многомерными, хотя одномерные используются значительно чаще. Массивы предоставляют удобные средства создания списков взаимосвязанных переменных.

Возможно, чаще других вы будете сталкиваться с символьными массивами, потому что они используются для хранения символьных строк. Язык C++ не имеет встроенного типа строковых данных. Строки реализуются как массивы символов. Такой подход к строкам обеспечивает большие возможности и гибкость, чем это доступно в языках, использующих строковый тип данных.

Указатель — это объект, содержащий адрес памяти. Указатель, как правило, используется для того, чтобы получить доступ к значению другого объекта. Часто этим другим объектом является массив. Фактически указатели и массивы связаны гораздо теснее, чем это можно предположить.

Цель

4.1. Одномерные массивы

Одномерный массив представляет собой список взаимосвязанных переменных. Такие списки широко используются в программировании. Например, одномерный массив может использоваться для хранения учетных номеров активных пользователей сети. В другом массиве могут храниться спортивные результаты бейсбольной команды. Вычисляя среднее значение набора данных, вы обычно для хранения этих данных используете массив. Для современного программирования массивы являются фундаментальными объектами.

Общая форма объявления одномерного массива выглядит так:

```
тип имя[размер];
```

В этом объявлении *тип* задает базовый тип массива. Базовый тип определяет тип данных каждого элемента, из которых образуется массив. Число элементов, содержащихся в массиве, задается величиной *размер*. Приведенная ниже строка объявляет массив целых чисел типа *int*, имеющий имя *sample* и состоящий из 10 элементов:

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

Обращение к индивидуальным элементам массива осуществляется с помощью индексов. *Индекс* определяет позицию элемента внутри массива. В C++ все массивы используют ноль в качестве индекса своего первого элемента. Объявленный выше массив **sample** содержит 10 элементов с индексными значениями от 0 до 9. Доступ к элементу массива достигается путем индексации массива с помощью номер элемента. Для индексации массива укажите номер требуемого элемента, заключив его в квадратные скобки. Так, первый элемент массива **sample** называется **sample[0]**, последний — **sample[9]**. В качестве примера рассмотрим программу, которая заполняет массив **sample** числами от 0 до 9:

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

int main()
{
    int sample[10]; // резервирование места под 10 элементов
                    // типа int

    int t;

    // заполним массив
    for(t=0; t<10; ++t) sample[t] = t;

    // выведем массив
    for(t=0; t<10; ++t)
        cout << "Это sample[" << t << "]: " << sample[t] << "\n";

    return 0;
}
```

Вот вывод этой программы:

```
3To sample[0]: 0
3To sample[1]: 1
3To sample[2]: 2
3To sample[3]: 3
3To sample[4]: 4
3To sample[5]: 5
3To sample[6]: 6
3To sample[7]: 7
3To sample[8]: 8
3To sample[9]: 9
```

В C++ все массивы располагаются в последовательных ячейках памяти. (Другими словами, все элементы массива располагаются в памяти вплотную друг к другу) Наименьший адрес соответствует перво-

му элементу, а наибольший адрес — последнему. Например, после выполнения этого программного фрагмента:

```
int nums[5];
int i;

for(i=0; i<5; i++) nums[i] = i;
```

массив `nums` будет выглядеть таким образом:

| num[0] | num[1] | num[2] | num[3] | num[4] |
|--------|--------|--------|--------|--------|
| 0 | 1 | 2 | 3 | 4 |

Массивы так широко используются в программировании из-за того, что они позволяют без труда обрабатывать наборы взаимосвязанных переменных. Вот только один пример. Приводимая ниже программа создает массив из десяти элементов и присваивает каждому элементу значение. Затем программа вычисляет среднее из всех этих значений, а также находит минимальное и максимальное значения.

```
/*
    Вычисление среднего и нахождение минимального
    и максимального значений для набора значений.
*/

#include <iostream>
using namespace std;

int main()
{
    int i, avg, min_val, max_val;
    int nums[10];

    nums[0] = 10;
    nums[1] = 18;
    nums[2] = 75;
    nums[3] = 0;
    nums[4] = 1;
    nums[5] = 56;
    nums[6] = 100;
    nums[7] = 12;
    nums[8] = -19;
    nums[9] = 88;
```

```

// вычислим среднее
avg = 0;
for(i=1; i<10; i++)
    avg += nums[i]; ← Суммирование значений в nums.

avg /= 10; ← Вычислим среднее.

cout << "Среднее равно " << avg << '\n';

// найдем минимальное и максимальное значения
min_val = max_val = nums[0];
for(i=1; i<10; i++) {
    if(nums[i] < min_val) min_val = nums[i];
    if(nums[i] > max_val) max_val = nums[i];
}
cout << "Минимальное значение: " << min_val << '\n';
cout << "Максимальное значение: " << max_val << '\n'; ←
return 0;
}

```

Найдем минимальное и максимальные значения в **nums**

Ниже показан вывод этой программы:

```

Среднее равно 33
Минимальное значение: -19
Максимальное значение: 100

```

Обратите внимание на то, как программа работает с элементами массива **nums**. Хранение анализируемых значений в массиве заметно облегчает этот процесс. Переменная **i** управления циклом **for** используется одновременно в качестве индекса. Циклы такого рода широко используются при обработке массивов.

Работая с массивами, следует иметь в виду серьезное ограничение. В C++ недопустимо копировать один массив в другой с помощью операции присваивания. Например, следующий фрагмент неправилен:

```

int a[10], b[10];
// ...
a = b; // ошибка - недопустимая операция

```

Для передачи содержимого одного массива в другой вы должны скопировать каждое значение индивидуально, например, таким образом:

```

for(i=0; i < 10; i++) a[i] = b[i];

```


Границы не проверяются!

C++ не выполняет проверку границ массива. Это означает, что ничто не может вам помешать выйти за пределы массива. Другими словами, вы можете индексировать массив размера *N* за пределами *N* и не получите при этом никаких сообщений об ошибках, ни на этапе компиляции, ни при выполнении. В то же время, такое действие часто приводит к катастрофическим последствиям для вашей программы. Например, приведенный ниже пример будет откомпилирован и выполнен без каких-либо сообщений об ошибках, хотя мы явно вышли за пределы массива **crash**:

```
int crash[10], i;  
for(i=0; i<100; i++) crash[i]=i;
```

В этом случае цикл будет повторяться 100 раз, хотя в массиве **crash** только десять элементов! В результате будет затерта память, не принадлежащая массиву **crash**.

Если выход за пределы массива происходит в процессе операции присваивания, то скорее всего будет затерта память, выделенная для других целей, например, для хранения других переменных. Если выход за пределы массива происходит в процессе чтения, то неправильные данные нарушат нормальное выполнение программы. В любом случае ваша обязанность, как программиста, следить за тем, чтобы массивы имели достаточный размер для хранения того, что программа будет в них записывать, и в случае необходимости осуществлять проверку на выход за границы массива.

Минутная тренировка

1. Что такое одномерный массив?
 2. Индекс первого элемента массива всегда равен нулю. Справедливо ли это утверждение?
 3. Осуществляет ли C++ проверку на выход за границы массива?
 - 1 Одномерный массив представляет собой список взаимосвязанных переменных
 - 2 Справедливо. Первым индексом всегда является ноль
 - 3 Нет, C++ не выполняет проверку на выход за границы массива
-

Спросим у эксперта

Вопрос: Если выход за пределы массива может привести к катастрофическим последствиям, почему C++ не выполняет соответствующую проверку при операциях с массивами?

Ответ: C++ разрабатывался с целью дать профессиональным программистам средство для создания максимально быстрых и эффективных про-

грамм. Для достижения этой цели в C++ почти не предусмотрено проверок на ошибки во время выполнения, так как такая проверка замедляет (иногда драматически) выполнение программы. Наоборот, C++ ожидает от вас, программиста, достаточной ответственности, чтобы не допускать выхода за пределы массивов; если же по логике программы такое нарушение все же может иметь место, то вы сами должны добавить в программу требуемые средства проверки. К тому же, как вы узнаете позже, если ваша программа действительно нуждается в проверке на ошибки времени выполнения, то вы имеете возможность определять массивы собственных типов, которые будут выполнять такого рода проверку.

Цель

4.2. Двумерные массивы

C++ допускает возможность создания многомерных массивов. Простейшей формой такого массива является двумерный массив. *Двумерный массив* есть, в сущности, список, составленный из одномерных массивов. Для того, чтобы объявить двумерный массив **twoD** целых чисел размера 10×20, вы должны написать:

```
int twoD[10][20];
```

Обратите особое внимание на это объявление. В отличие от некоторых других компьютерных языков, которые для разделения размерностей массива используют запятые, в C++ каждый размер заключается в собственные квадратные скобки. Например, для обращения к элементу с номером 3,5 в массиве **twoD** вы должны использовать обозначение **twoD[3][5]**.

В приводимом ниже примере двумерный массив заполняется числами от 1 до 12:

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

int main()
{
    int t,i, nums[3][4];

    for(t=0; t < 3; ++t) {
        for(i=0; i < 4; ++i) {
            nums[t][i] = (t*4)+i+1;
            cout << nums[t][i] << ' ' ;
        }
        cout << '\n';
    }
}
```

← Индексация **nums** требует двух индексов

```
    return 0;
}
```

В этом примере элемент `nums[0][0]` будет иметь значение 1, `nums[0][1]` — значение 2, `nums[0][2]` — значение 3 и т. д. Значение последнего элемента `nums[2][3]` будет равно 12. Весь массив можно представить себе так, как это изображено на рисунке:

| | 0 | 1 | 2 | 3 | ← Правый индекс |
|---|---|----|----|----|-----------------|
| 0 | 1 | 2 | 3 | 4 | |
| 1 | 5 | 6 | 7 | 8 | |
| 2 | 9 | 10 | 11 | 12 | |

Левый индекс ↑

↑ `num[1][2]`

Двумерные массивы сохраняются в памяти в виде матрицы, состоящей из строк и столбцов, причем первый индекс определяет строку, а второй — столбец. Отсюда следует, что если элементы массива обрабатываются в том порядке, в каком они фактически располагаются в памяти, то правый индекс изменяется быстрее левого.

Следует иметь в виду, что память под все элементы массива выделяется во время компиляции. Далее, память, используемая для хранения массива, должна оставаться выделенной на все время существования массива. В случае двумерного массива мы можете определить, сколько байтов памяти он требует, по следующей формуле:

число байтов = число строк × число столбцов × число байтов в типе массива

Таким образом, для четырехбайтовых целых чисел массив с размерностью 10×5 займет в памяти $10 \times 5 \times 4 = 200$ байтов.

Цель

4.3. Многомерные массивы

C++ допускает создание массивов с числом измерений больше двух. Вот общая форма объявления многомерного массива:

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

Например, следующее объявление создает массив целых чисел с размерностью $4 \times 10 \times 3$:

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

Массивы с числом измерений три и больше используются не часто, отчасти из-за того, что они требуют для своего хранения значительных объемов памяти. Вспомним, что память для хранения всех элементов массива выделяется на все время жизни массива. Многомерные массивы могут потреблять значительные объемы памяти. Например, четырехмерный символьный массив с размерностью $10 \times 6 \times 9 \times 4$ потребует $10 \times 6 \times 9 \times 4$, или 2160 байтов. Если же каждый размер массива увеличить в 10 раз ($100 \times 60 \times 90 \times 40$), то память, необходимая для размещения такого массива, возрастет до 21600000 байтов! Таким образом, большие многомерные массивы могут быть причиной нехватки памяти для остальных частей вашей программы. Программа, создающая массивы с числом измерений больше, чем два или три, может быстро исчерпать всю наличную память!

Минутная тренировка

1. Каждое измерение многомерного массива задается с использованием собственной пары квадратных скобок. Справедливо ли это утверждение?
2. Покажите, как объявить двумерный массив целых чисел с именем `list` и размером 4×9 .
3. Покажите, как получить доступ к элементу с индексами 2,3 для массива из предыдущего вопроса.

1. Справедливо.
 2. `int list[4][9];`
 3. `list[2][3]`
-

Проект 4-1

Упорядочение массива

Поскольку в одномерном массиве данные организованы в виде индексированного линейного списка, такая структура оказывается чрезвычайно удобной для упорядочения данных. В этом проекте вы освоите простой способ упорядочения массива. Как вы, возможно, знаете, имеется целый ряд различных алгоритмов упорядочения. В частности, можно упомянуть быстрое упорядочение, упорядочение встряхиванием и метод Шелла. Однако наиболее известным, удобным и простым для понимания является алгоритм, носящий название метода всплывающего пузырька или просто пузырькового упорядочения. Хотя этот

метод не очень эффективен — фактически его нельзя использовать при работе с большими массивами — однако для упорядочения относительно коротких массивов он вполне годится.

Шаг за шагом

1. Создайте файл с именем **Bubble.cpp**.

2. Метод всплывающего пузырька получил свое название от способа, которым осуществляется операция упорядочения. Она использует повторные сравнения и, если необходимо, обмен местами соседних элементов массива. В ходе этой операции меньшие значения перемещаются к одному концу массива, а большие — к другому. Этот процесс напоминает всплывание пузырьков в баке с водой. Пузырьковое упорядочение требует выполнения нескольких проходов всего массива, причем в каждом проходе выполняется обмен местами расположенных не в том порядке элементов. Число проходов, требуемое для полного упорядочения всего массива, на единицу меньше числа элементов в массиве.

Ниже приведен код, формирующий ядро пузырькового упорядочения. Упорядочиваемый массив назван **nums**.

```
// Пузырьковое упорядочение.
for(a=1; a<size; a++)
    for(b=size-1; b>=a; b-) {
        if(nums[b-1] > nums[b]) { // если не в том порядке
            // элементы обмениваются местами
            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }
}
```

Обратите внимание на то, что упорядочение требует двух циклов **for**. Во внутреннем цикле попарно сравниваются соседние элементы массива в поисках тех, которые расположены не в том порядке. Если такая пара найдена, элементы в ней меняются местами. После каждого выполнения всех шагов внутреннего цикла минимальный из оставшихся элементов перемещается в упорядоченную позицию. Во внешнем цикле этот процесс повторяется до тех пор, пока все элементы массива не будут упорядочены.

3. Ниже приведен полный текст программы **Bubble.cpp**:

```
/*
Проект 4-1
Демонстрация пузырькового упорядочения.
*/
```

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

int main()
{
    int nums[10];
    int a, b, t;
    int size;

    size = 10; // число упорядочиваемых элементов

    // дадим элементам массива случайные начальные значения
    for(t=0; t<size; t++) nums[t] = rand();
    // выведем на экран исходный массив
    cout << "Исходный массив:\n  ";
    for(t=0; t<size; t++) cout << nums[t] << ' ';
    cout << '\n';

    // Это пузырьковое упорядочение.
    for(a=1; a<size; a++)
        for(b=size-1; b>=a; b--) {
            if(nums[b-1] > nums[b]) { // если не в том порядке
                // элементы обмениваются местами
                t = nums[b-1];
                nums[b-1] = nums[b];
                nums[b] = t;
            }
        }

    // выведем упорядоченный массив
    cout << "\nУпорядоченный массив:\n  ";
    for(t=0; t<size; t++) cout << nums[t] << ' ';

    return 0;
}
```

Ниже приведен вывод одного из прогонов программы:

Исходный массив:

41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Упорядоченный массив:

41 6334 11478 15724 18467 19169 24464 26500 26962 29358

4. Алгоритм пузырькового упорядочения, будучи вполне удовлетворительным для небольших массивов, оказывается неэффективным для

массивов большой длины. Наилучшим алгоритмом общего назначения считается алгоритм быстрого упорядочения. Однако этот алгоритм опирается на средства языка C++, которые вы еще не изучали. Кроме того, хотя вариант быстрого упорядочения и реализован в функции с именем `qsort()` из стандартной библиотеки C++, однако чтобы пользоваться этой функцией, вы должны изучить C++ в больших деталях.

Цель

4.4. Строки

Безусловно, наиболее важным применением одномерных массивов является создание символьных строк. C++ поддерживает строки двух видов. Первый, наиболее часто используемый, — это *строка с завершающим нулем* или, другими словами, массив символов, заканчивающийся нулем. Строка с завершающим нулем содержит символы, образующие эту строку, за которыми помещается ноль. Такие строки используются чрезвычайно широко, так как они обеспечивают высокий уровень эффективности и предоставляют программисту возможность выполнять разнообразные строковые операции. Когда программист на C++ использует термин “строка”, он (или она) обычно имеет в виду именно строку с завершающим нулем. Второй вид строк, определенный в C++ — это класс `string`, который входит в библиотеку классов C++. Таким образом, `string` не является встроенным типом. Класс `string` позволяет использовать объектно-ориентированный подход при обработке строк, однако он используется не так широко, как строки с завершающим нулем. Здесь мы рассмотрим первый вид строк.

Основы техники строк

Объявляя символьный массив, который будет содержать строку с завершающим нулем, вы должны задать ему длину на один символ больше, чем у самой длинной помещаемой в него строки. Если, например, вы хотите объявить массив `str`, в котором будет находиться 10-символьная строка, то вы должны написать следующее:

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

Задание величины 11 в качестве длины массива обеспечит в строке место для завершающего нуля.

Как вы уже знаете, C++ позволяет определять символьные константы. Символьная константа представляет собой список символов, заключенный в двойные кавычки. Вот несколько примеров:

```
"Привет!"      "I like C++"      "Mars"      ""
```

Добавлять ноль в конец строки вручную нет необходимости; компилятор C++ сделает это сам. В результате строка "Mars" будет выглядеть в память таким образом:

| | | | | |
|---|---|---|---|---|
| М | а | г | с | 0 |
|---|---|---|---|---|

Последняя строка в нашем примере выглядит как "". Такая строка называется *нулевой строкой*. Она содержит только завершающий ноль и больше ничего. Нулевые строки используются в программах в качестве пустых строк.

Ввод строки с клавиатуры

Самый простой способ прочесть в программу строку, вводимую с клавиатуры, заключается в использовании массива **char** и предложения **cin**. Приводимая ниже программа вводит строку, набираемую пользователем на клавиатуре:

// Использование cin для ввода строки с клавиатуры.

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

```
int main()
{
```

```
    char str[80];
```

```
    cout << "Вводите строку: ";
```

```
    cin >> str; // ввод строки с клавиатуры
```

```
    cout << "Вот ваша строка: ";
```

```
    cout << str;
```

```
    return 0;
```

```
}
```

Чтение строки с помощью cin



Пример прогона программы:

Вводите строку: Проверка

Вот ваша строка: Проверка

Хотя приведенная выше программа технически правильна, она не всегда будет работать так, как вы этого ожидаете. Для того, чтобы убедиться в этом, запустите программу и попробуйте ввести строку "Это новая проверка". Вот что получится:

Вводите строку: Это новая проверка
Вот ваша строка: Это

Когда программа повторяет на экране введенную вами строку, то вы видите только слово “Это”, а не всю строку. Так происходит потому, что система ввода-вывода C++ прекращает ввод с клавиатуры при получении пробельного символа. Пробельные символы включают в себя пробелы, символы табуляции и символы новой строки.

Для решения проблемы пробельных символов вы должны использовать другую библиотечную функцию C++, `gets()`. Общая форма вызова функции `gets()` выглядит так:

```
gets(имя-массива);
```

Если вам нужно ввести в программу строку с клавиатуры, вызовите функцию `gets()` с именем массива (без какого-либо индекса) в качестве аргумента. После возврата из `gets()` массив будет содержать строку, набранную на клавиатуре. Функция `gets()` вводит символы до тех пор, пока не встретится с символом возврата каретки. Для использования этой функции необходим заголовок `<cstdio>`.

Приводимый ниже вариант предыдущей программы использует функцию `gets()`, позволяющую вводить строки, содержащие пробелы:

```
// Использование gets() для ввода строки с клавиатуры.
```

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

```
int main()
{
```

```
    char str[80];
```

```
    cout << "Вводите строку: ";
```

```
    gets(str); // ввод строки с помощью gets() ←
```

```
    cout << "Вот ваша строка: ";
```

```
    cout << str;
```

```
    return 0;
```

```
}
```

Чтение строки с помощью `gets()`

Пример прогона программы:

Вводите строку: Это строка проверки
Вот ваша строка: Это строка проверки

Теперь пробельные символы читаются и включаются в строку. Еще одно замечание: обратите внимание на то, что в предложении `cout`

можно непосредственно использовать имя `str`. Вообще имя символьного массива, содержащего строку, можно использовать в любом месте, где допустимо применение символьной константы.

Не забывайте, что ни `cin`, ни `gets()` не выполняет проверку на выход за границы массива. Поэтому если пользователь введет строку, длина которой больше размера массива, то память за пределами массива будет затерта. Позже вы познакомитесь с альтернативой, которая помогает избежать этой неприятности.

Минутная тренировка

1. Что представляет собой строка с завершающим нулем?
 2. Какой размер должен иметь символьный массив, в котором будет храниться строка длиной 8 байт?
 3. Какую функцию можно использовать для ввода с клавиатуры строки, содержащей пробелы?
-
1. Строка с завершающим нулем представляет собой массив символов, который заканчивается нулем
 2. Если массив предназначен для хранения 8-символьной строки, он должен иметь размер по меньшей мере 9 символов.
 3. Для ввода с клавиатуры строки, содержащей пробелы, можно использовать функцию `gets()`.
-

Цель

4.5.

Некоторые библиотечные функции обработки строк

C++ поддерживает широкий диапазон функций для манипуляций со строками. Самые известные из них:

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

Все строковые функции используют один и тот же заголовок `<cstring>`. Рассмотрим эти функции подробнее.

`strcpy()`

Вызов этой функции выглядит таким образом:

```
strcpy(куда,откуда)
```

Функция **strcpy()** копирует содержимое строки *откуда* в строку *куда*. Не забывайте, что массив, образующий *куда*, должен иметь достаточный размер, чтобы в него поместилась строка *откуда*. Если места в нем не хватит, то произойдет выход за пределы массива *куда*, что, скорее всего, приведет к аварийному завершению программы.

strcat()

Вызов этой функции выглядит таким образом:

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

Функция **strcat()** присоединяет строку **s2** к концу **s1**; строка **s2** остается без изменений. Вы должны обеспечить достаточно большой размер строки **s1**, чтобы она могла вместить как свое исходное содержимое, так и содержимое **s2**.

strcmp()

Вызов этой функции выглядит таким образом:

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

Функция **strcmp()** сравнивает две строки и возвращает 0, если они равны. Если **s1** больше **s2** лексикографически (т. е. по порядку следования символов алфавита), возвращается положительное число; если она меньше **s2**, возвращается отрицательное число.

При использовании **strcmp()** важно иметь в виду, что при равенстве строк она возвращает **false**. Таким образом, если вы хотите, чтобы что-то произошло при равенстве строк, вам придется использовать оператор **!**. В следующем примере условие, управляющее предложением **if**, истинно, если **str** равна строке **"C++"**:

```
if(!strcmp(str, "C++") cout << "str = C++";
```

strlen()

Общая форма вызова функции **strlen()** такова:

```
strlen(s);
```

Здесь **s** — это строка. Функция **strlen()** возвращает длину строки, на которую указывает **s**.

Пример обработки строк

В приводимой ниже программе используются все четыре рассмотренные выше функции обработки строк:

```
// Демонстрация строковых функций.

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

int main()
{
    char s1[80], s2[80];

    strcpy(s1, "C++");
    strcpy(s2, " мощный язык.");

    cout << "длины: " << strlen(s1);
    cout << ' ' << strlen(s2) << '\n';

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

    strcat(s1, s2);
    cout << s1 << '\n';

    strcpy(s2, s1);
    cout << s1 << " и " << s2 << "\n";

    if(!strcmp(s1, s2))
        cout << "s1 и s2 теперь одинаковы.\n";

    return 0;
}
```

Ниже приведен вывод этой программы:

```
длины: 3 22
не равны
C++ мощный язык.
C++ мощный язык. и C++ мощный язык.
s1 и s2 теперь одинаковы.
```

Использование завершающего нуля

Наличие в конце строк завершающего нуля часто помогает упростить различные операции. Так, приводимая ниже программа преобразует строчные буквы строки в прописные:

```
// Преобразование строчных символов строки в прописные.

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

int main()
{
    char str[80];
    int i;

    strcpy(str, "this is a test");

    for(i=0; str[i]; i++) str[i] = toupper(str[i]);
    cout << str;
    return 0;
}
```

↑ Цикл завершается, когда индексируется завершающий ноль

Вывод этой программы:

THIS IS A TEST

Приведенная программа использует библиотечную функцию **`toupper()`**, которая возвращает прописной эквивалент своего символьного аргумента, чем и осуществляется преобразование всех символов строки. Функция **`toupper()`** использует заголовок **`<cctype>`**.

Обратите внимание на то, что в качестве условия для цикла **`for`** выступает просто массив, индексируемый управляющей переменной. Здесь используется то обстоятельство, что любое ненулевое значение является истинным. Вспомним, что все коды символов не равны 0, но любая строка завершается нулем. Поэтому наш цикл выполняется до тех пор, пока он не встретится с нулевым символом, когда **`str[i]`** оказывается равным 0. Поскольку завершающий ноль отмечает конец строки, цикл останавливается в точности там, где и требуется. В профессионально написанных C++-программах вы увидите массу примеров использования завершающего строку нуля аналогичным образом.

Минутная тренировка

1. Для чего предназначена функция `strcat()`?
2. Что возвращает функция `strcmp()`, когда она сравнивает две эквивалентные строки?
3. Покажите, как получить длину строки с именем `mystr`.
 - 1 Функция `strcat()` выполняет объединение (конкатенацию) двух строк.
 - 2 Функция `strcmp()` при сравнении двух эквивалентных строк возвращает 0.
 3. `strlen(mystr)`

Спросим у эксперта

Вопрос: Имеются ли в C++, помимо `toupper()`, другие функции манипуляции символами?

Ответ: Да. Стандартная библиотека C++ содержит несколько других функций манипуляции символами. Например, дополнением к `toupper()` служит функция `tolower()`, преобразующая прописные символы в их строчные эквиваленты. С помощью функции `isupper()` вы можете определить вид буквы (строчная или прописная); эта функция возвращает `true`, если буква прописная. Есть и функция `islower()`, которая возвращает `true` в случае строчной буквы. К другим символично-ориентированным функциям относятся `isalpha()`, `isdigit()`, `isspace()` и `ispunct()`. Все эти функции принимают в качестве аргумента символ и определяют его категорию. Например, `isalpha()` возвращает `true`, если аргумент является буквой алфавита.

Цель

4.6. Инициализация массивов

В C++ предусмотрены средства инициализации массивов. Общая форма инициализации массива сходна с аналогичным действием для других переменных, как это показано ниже:

тип имя-массива[размер] = {список-значений};

Здесь *список-значений* представляет собой список значений, разделенных запятыми и совместимых по типу с базовым типом массива. Первое значение помещается в первый элемент массива, второе — во второй элемент и т. д. Обратите внимание на то, что точка с запятой ставится после закрывающей скобки }.

В следующем примере массив из десяти целочисленных элементов инициализируется числами от 1 до 10:

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

В результате компиляции этого предложения `i[0]` будет иметь значение 1, а `i[9]` — 10.

Символьные массивы, предназначенные для хранения строк, допускают упрощенную инициализацию в такой форме:

```
char имя-массива[размер] = "строка";
```

Например, следующий программный фрагмент инициализирует `str` строкой "C++":

```
char str[4] = "C++";
```

Тот же результат можно было получить и иначе:

```
char str[4] = {'C', '+', '+', '\0'};
```

Поскольку строки в C++ заканчиваются нулем, вы должны объявить массив достаточно большой длины, предусмотрев место для этого нуля. Именно поэтому массив `str` имеет в этих примерах длину 4 символа, хотя в строке "C++" их только три. Если для инициализации массива используется строковая константа, компилятор автоматически записывает в массив завершающий ноль.

Многомерные массивы инициализируются так же, как и одномерные. Например, следующий фрагмент инициализирует массив `sqr` числами от 1 до 10 и их квадратами:

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

На рис. 4-1 показано, как массив `sqrs` располагается в памяти.

Инициализируя многомерный массив, вы можете добавить фигурные скобки вокруг инициализаторов каждого измерения. Это действие носит название *субагрегатной группировки*. Воспользовавшись этим приемом, объявление массива из предыдущего примера можно записать таким образом:

```
int sqrs[10][2] = {
    {1, 1},
```

```
{2, 4},  
{3, 9},  
{4, 16},  
{5, 25},  
{6, 36},  
{7, 49},  
{8, 64},  
{9, 81},  
{10, 100}  
};
```

Если, используя субагрегатную группировку, вы не укажете достаточное количество инициализаторов для данной группы, оставшимся членам будут автоматически присвоены нулевые значения.

| | 0 | 1 | ← Правый индекс |
|---|----|-----|-----------------|
| 0 | 1 | 1 | |
| 1 | 2 | 4 | |
| 2 | 3 | 9 | |
| 3 | 4 | 16 | |
| 4 | 5 | 25 | |
| 5 | 6 | 36 | |
| 6 | 7 | 49 | |
| 7 | 8 | 64 | |
| 8 | 9 | 81 | |
| 9 | 10 | 100 | |

↑ Левый индекс

Рис. 4-1. Инициализированный массив sqrs

Приведенная ниже программа использует массив `sqrs` для нахождения квадрата числа, введенного пользователем. Программа сначала ищет это число в массиве, а затем выводит соответствующее значение квадрата.

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

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

    cout << "Введите число от 1 до 10: ";
    cin >> i;

    // ищем i
    for(j=0; j<10; j++)
        if(sqrs[j][0]==i) break;
    cout << "Квадрат " << i << " равен ";
    cout << sqrs[j][1];

    return 0;
}
```

Вот пример прогона программы:

```
Введите число от 1 до 10: 4
Квадрат 4 равен 16
```

Инициализация массивов неопределенной длины

Объявляя инициализируемый массив, можно заставить C++ автоматически определять его размер. Для этого достаточно не указывать

размер массива в объявлении. В этом случае компилятор определит отсутствующий размер, пересчитав поставляемые инициализаторы, и создаст массив требуемой длины. Например,

```
int nums[] = { 1, 2, 3, 4 };
```

создает массив с именем `nums` из четырех элементов, которые содержат значения 1, 2, 3 и 4. Массив, при объявлении которого явным образом не указывается размер (как, например, массив `nums`), называют *массивом неопределенной длины*.

Такие массивы весьма полезны. Представьте себе, например, что вы, используя массивы с инициализацией, создаете таблицу Интернет-адресов:

```
char e1[16] = "www.osborne.com";  
char e2[16] = "www.weather.com";  
char e3[15] = "www.amazon.com";
```

Очевидно, что считать вручную количество символов в каждом адресе, чтобы определить длину соответствующего массива, слишком хлопотно. Кроме того, такой способ чреват ошибками, если вы случайно ошибетесь в счете и зададите неправильную длину массива. Гораздо лучше переложить на компилятор вычисление длины этих массивов, как это показано ниже:

```
char e1[] = "www.osborne.com";  
char e2[] = "www.weather.com";  
char e3[] = "www.amazon.com";
```

Помимо того, что такой метод избавляет вас от утомительной работы, он хорош еще и тем, что вы можете изменить любой адрес и не думать о соответствующем изменении размера массива.

Инициализация массивов неопределенной длины не ограничивается одномерными массивами. Для многомерного массива вы можете указать все размеры, кроме самого левого, что даст возможность правильно индексировать массив. С помощью такого метода вы можете создавать таблицы переменной длины, и компилятор будет автоматически выделять под них требуемую память. В следующем примере `sqrs` объявляется как массив неопределенной длины:

```
int sqrs[][2] = {  
    1, 1,  
    2, 4,  
    3, 9,  
    4, 16,  
    5, 25,
```

```
6, 36,  
7, 49,  
8, 64,  
9, 81,  
10, 100  
};
```

Преимуществом такого объявления перед объявлением с явным указанием всех размеров массива является возможность удлинить или укоротить эту таблицу, не внося при этом изменений в размеры массива.

Цель

4.7. Массивы строк

Особой формой двумерного массива является массив строк. Такие массивы широко используются в программировании. Например, внутренний обработчик базы данных может сравнивать команды пользователя с массивом строк допустимых команд. Для создания массива строк используется двумерный символьный массив, в котором левый индекс характеризует число строк, а правый — их максимальную длину. Например, приведенное ниже объявление создает массив из 30 строк, каждая из которых может иметь длину до 80 символов:

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

Обратиться к индивидуальной строке этого массива не составляет труда: вы просто указываете один левый индекс. Например, приведенное ниже предложение вызывает функцию `gets()` для заполнения третьей строки в `str_array`:

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

Чтобы получить доступ к индивидуальному символу в третьей строке, вы используете предложение вроде следующего:

```
cout << str_array[2][3];
```

В результате на экран выводится четвертый символ третьей строки.

Приводимая ниже программа демонстрирует работу с массивом строк на примере очень простого компьютеризованного телефонного справочника. Двумерный массив `numbers` содержит пары имя-телефонный номер. Для определения телефонного номера вы вводите имя; на экран выводится соответствующий телефонный номер.

```
// Простой компьютеризованный телефонный справочник.

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

int main()
{
    int i;
    char str[80];
    char numbers[10][80] = { ← Это массив из 10 строк, каждая из кото-
        "Том", "555-3322",      рых может содержать до 79 символов.
        "Мэри", "555-8976",
        "Джон", "555-1037",
        "Рейчел", "555-1400",
        "Шерри", "555-8873"
    };

    cout << "Введите имя: ";
    cin >> str;

    for(i=0; i < 10; i += 2)
        if(!strcmp(str, numbers[i])) {
            cout << "Телефон " << numbers[i+1] << "\n";
            break;
        }
    if(i == 10) cout << "Не найдено.\n";

    return 0;
}
```

Вот пример прогона программы:

```
Введите имя: Джон
Телефон 555-1037
```

Обратите внимание на то, как выполняется наращивание управляющей переменной цикла `for`: в каждом шаге к ней прибавляется 2. Так делается потому, что в массиве чередуются имена и телефонные номера.

Минутная тренировка

1. Покажите, как инициализировать четырехэлементный массив чисел `int` значениями 1, 2, 3 и 4.
2. Как по-другому можно записать это инициализирующее предложение:
`char str[7] = {"П", "Р", "и", "в", "е", "т", "\0"};`

3. Напишите следующее предложение в виде массива неопределенной длины:

```
int nums[4] = {44, 55, 66, 77};
```

1. `int list[4] = { 1, 2, 3, 4 };`

2. `char str[7] = "Привет";`

3. `int nums[] = {44, 55, 66, 77};`

Цель

4.8. Указатели

Указатели являются одним из наиболее мощных средств C++. Они же оказываются и самыми трудными в освоении. Будучи вероятным источником всяческих неприятностей, указатели тем не менее играют ключевую роль в программировании на C++. Именно посредством указателей C++ поддерживает такие средства, как связанные списки и динамическое выделение памяти. Указатели позволяют функциям изменять содержимое своих аргументов. Однако эти и другие возможные применения указателей будут обсуждаться нами в последующих модулях. Здесь вы только получите базовые сведения об указателях и научитесь их использовать.

В последующих обсуждениях нам понадобится ссылаться на размеры некоторых базовых типов данных C++. Для определенности этих обсуждений будем считать, что символы имеют длину 1 байт, целые (**int**) — четыре байта, переменные с плавающей точкой (**float**) — тоже четыре байта, а переменные **double** — восемь байтов. Другими словами, мы будем вести рассуждения применительно к типичной 32-разрядной среде.

Что такое указатели?

Указатель — это объект, содержащий адрес памяти. Очень часто этот адрес указывает на расположение в памяти другого объекта, например, переменной. Так, если *x* содержит адрес *y*, тогда говорят, что *x* “указывает” на *y*.

Переменные-указатели объявляются особым образом. Общая форма объявления переменной-указателя такова:

тип **имя-переменной*;

Здесь *тип* — это базовый тип указателя. *Базовый тип* определяет, на данные какого типа будет указывать этот указатель. *имя-переменной* — это имя переменной-указателя. Так, чтобы объявить *ip*, как указатель на *int*, вы должны написать:

```
int *ip;
```

Поскольку базовым типом `ip` указан `int`, этот указатель можно использовать только для указания на переменные типа `int`.

Теперь объявим указатель на `float`:

```
float *fp;
```

В этом случае базовым типом `fp` является `float`, и указатель `fp` может указывать только на переменные типа `float`.

Итак, в предложениях объявления указание перед именем переменной знака `*` делает эту переменную указателем.

Цель

4.9. Операторы указателей

Для использования с указателями предусмотрены два специальных оператора: `*` и `&`. Оператор `&` является унарным; он возвращает адрес памяти, в которой расположен его операнд. (Вспомним, что унарный оператор требует только одного операнда). Например, предложение

```
ptr = &total;
```

помещает в `ptr` адрес памяти, где находится переменная `total`. Этот адрес является адресом ячейки памяти, в которой хранится переменная `total`. Этот адрес не имеет *ничего общего со значением total*. Про операцию `&` можно сказать, что она возвращает *адрес* переменной, перед которой она указана. Таким образом, приведенное выше предложение присваивания можно передать словами, как “`ptr` получает адрес `total`”. Чтобы лучше разобраться в этом присваивании, предположим, что переменная `total` расположена по адресу 100. Тогда после выполнения присваивания `ptr` будет иметь значение 100.

Второй оператор, `*`, в некотором роде обратный по отношению к `&`. Этот унарный оператор возвращает значение переменной, расположенной по адресу, который указан в качестве операнда этого оператора. Продолжая тот же пример, если `ptr` содержит адрес переменной `total`, то предложение

```
val = *ptr;
```

поместит значение переменной `total` в `val`. Если, например, `total` первоначально имела значение 3200, то `val` будет теперь иметь то же значение 3200, потому что это значение, которое хранится в памяти по адресу 100, т. е. по адресу, который присвоен переменной `ptr`. Операцию `*` можно называть “по адресу”. Таким образом, последнее предложение можно передать словами, как “`val` получает значение, расположенное по адресу `ptr`”.

Приведенная ниже программа выполняет последовательность только что описанных операций:

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

int main()
{
    int total;
    int *ptr;
    int val;

    total = 3200; // присвоим total значение 3200

    ptr = &total; // получим адрес total

    val = *ptr; // получим значение по этому адресу

    cout << "total равно: " << val << '\n';

    return 0;
}
```

К сожалению, получилось так, что знак умножения и символ “по адресу” совпадают, что часто сбивает с толку начинающих изучать C++. Эти операторы не имеют никакого отношения друг к другу. Имейте в виду, что оба оператора указателей, `&` и `*`, имеют более высокий относительный приоритет, чем любой арифметический оператор за исключением унарного минуса, с которым они находятся на одном уровне приоритета.

Использование указателя часто называют *косвенной* операцией, потому что вы обращаетесь к одной переменной не непосредственно, а косвенно, через другую переменную.

Минутная тренировка

1. Что такое указатель?
 2. Покажите, как объявить указатель `valPtr` на переменную типа `long int`.
 3. Что выполняют операторы `*` и `&` применительно к указателям?
 1. Указатель — это объект, содержащий адрес памяти, в которой расположен другой объект.
 2. `long int *valPtr;`
 3. Оператор `*` позволяет получить значение, сохраненное по адресу, перед которым указан этот оператор. Оператор `&` позволяет получить адрес объекта, перед которым он указан.
-

Базовый тип указателя имеет большое значение

В предыдущем подразделе было показано, что переменной **val** можно присвоить значение переменной **total** косвенным образом, посредством указателя. Возможно, при чтении этого материала вы задались вопросом: каким образом C++ узнает, сколько байтов надо скопировать в переменную **val** из памяти с адресом, содержащимся в **ptr**? Или, в более общем виде, каким образом компилятор переносит правильное число байтов в любой операции присваивания, использующей указатель? Ответ заключается в том, что тип данных, над которыми осуществляется операция посредством указателя, определяется базовым типом указателя. В нашем случае **ptr** является указателем на **int**, и поэтому в **val** копируются четыре байта (в предположении, что используется 32-разрядная среда) из памяти с адресом, содержащимся в **ptr**. Если бы, однако, **ptr** был указателем на **double**, то скопировались бы 8 байтов.

Весьма важно следить за тем, чтобы переменные-указатели всегда указывали на переменные правильных типов. Если, например, вы объявляете указатель на тип **int**, то компилятор предполагает, что любой объект, на который указывает этот указатель, является целочисленной переменной. Если в действительности это переменная другого типа, вас наверняка ожидают неприятности. Например, приведенный ниже фрагмент неверен:

```
int *p;
double f;
// ...
p = &f; // ОШИБКА
```

Этот фрагмент неверен, потому что вы не можете присвоить указателю на целочисленную переменную значение указателя на переменную типа **double**. Действительно, операция **&f** создает указатель на **double**, однако **p** является указателем на **int**. Эти два типа несовместимы. (Фактически компилятор в этой точке сообщит об ошибке и ваша программа далее компилироваться не будет.)

Хотя два указателя должны иметь совместимые типы, чтобы значение одного можно было присвоить другому, вы можете преодолеть это ограничение (на свой страх и риск) с помощью операции приведения типа. Так, приведенный ниже фрагмент формально верен:

```
int *p ;
double f;
// ...
p = (int *) &f; // Теперь формально правильно
```

Приведение к типу **int *** преобразует указатель на **double** в указатель на **int**. Однако, использование в данном случае приведения типов

является сомнительной операцией. Причина заключается в том, что базовый тип указателя определяет, как именно компилятор будет трактовать данные, на которые этот указатель указывает. В рассматриваемом примере, хотя **p** фактически указывает на переменную с плавающей точкой, компилятор будет “думать”, что **p** указывает на **int** (потому что **p** является указателем на **int**).

Чтобы лучше разобраться в том, почему приведение типа в операциях присваивания указателей различающихся типов обычно не приводит к добру, рассмотрим такую короткую программу:

// Эта программа будет работать неправильно.

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

int main()
{
    double x, y;
    int *p;

    x = 123.23;
    p = (int *) &x; // используем приведение типа для присваивания
                    // значения double * переменной типа int *

    y = *p;        // К чему это приведет?
    cout << y;     // Что здесь будет выведено на экран?

    return 0;
}
```

Эти предложения не приведут к желаемому результату.

Вот реальный вывод этой программы:

1.37439e+009

Выведенное значение с очевидностью не равно 123.23! И вот почему. В программе переменной **p** (которая является указателем на **int**) было присвоено значение адреса переменной **x** (которая имеет тип **double**). Далее, когда переменной **y** присваивается значение, на которое указывает **p**, **y** принимает только четыре байта данных (а не восемь, как это требуется для значения типа **double**), потому что **p** является указателем на **int**. В результате предложение **cout** выводит не значение 123.23, а “мусор”.

Операции присваивания посредством указателя

Вы можете использовать указатель с левой стороны предложения присваивания для того, чтобы присвоить значение ячейке памяти, на

которую указывает указатель. Приняв, что **p** является указателем на **int**, следующее предложение присвоит значение 101 ячейке памяти, на которую указывает **p**:

```
*p = 101;
```

Это присваивание можно описать словами таким образом: “ячейке, на которую указывает **p**, присвоить значение 101”. Для того, чтобы увеличить или уменьшить значение по адресу, содержащемуся в **p**, можно использовать предложения вроде следующего:

```
(*p)++;
```

Здесь необходимы круглые скобки, потому что оператор ***** имеет более низкий приоритет, чем оператор **++**.

В приведенной ниже программе демонстрируется присваивание значения посредством указателя:

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

int main()
{
    int *p, num;

    p = &num;

    *p = 100; ← Присвоим num значение 100 посредством p
    cout << num << ' ';
    (*p)++; ← Инкремент num посредством p.
    cout << num << ' ';
    (*p)--; ← Декремент num посредством p.
    cout << num << '\n';

    return 0;
}
```

Вывод программы выглядит следующим образом:

```
100 101 100
```

Цель

4.10. Выражения с указателями

Указатели допустимо использовать в большинстве выражений C++. Однако для таких операций существуют особые правила. Кроме

того, в ряде случаев, чтобы получить правильный результат, вам придется окружать определенные части выражений с указателями круглыми скобками.

Арифметика указателей

С указателями можно использовать четыре арифметических оператора: `++`, `--`, `+` и `-`. Чтобы разобраться, как работает арифметика указателей, предположим, что `p1` является указателем на `int` с текущим значением 2000 (т. е. `p1` содержит адрес, равный 2000). В 32-разрядной среде после выполнения выражение

```
p1++;
```

содержимое `p1` станет равным 2004, а не 2001! Так происходит потому, что каждый раз, когда выполняется инкремент указателя `p1`, он начинает указывать на следующую переменную типа `int`. То же справедливо в отношении операции декремента. Приняв опять, что `p1` содержит значение 2000, после выполнения выражения

```
p1--;
```

`p1` примет значение 1996.

Обобщая предыдущие примеры, сформулируем правила арифметики указателей. Каждая операция инкремента указателя приводит к тому, что указатель теперь указывает на ячейку памяти со следующим элементом его базового типа. Каждый раз, когда выполняется декремент указателя, он указывает на ячейку с предыдущим элементом его базового типа. Для указателей на символы инкремент и декремент действуют на указатель по правилам “обычной” арифметики, потому что символы имеют размер один байт. Однако указатель на любой другой тип данного будет увеличиваться или уменьшаться на длину своего базового типа.

Арифметика не ограничивается операциями инкремента и декремента. Вы можете прибавлять к указателям значения или вычитать из них. Выражение

```
p1 = p1+9;
```

приведет к тому, что `p1` будет указывать на девятый элемент базового типа `p1` за тем, на который он указывает в настоящий момент.

Хотя указатели нельзя складывать, допустимо вычитать один указатель из другого (при условии, что они одного базового типа). Разность будет представлять число элементов базового типа, разделяющих эти два указателя.

Кроме сложения и вычитания указателя и целого числа или вычитания одного указателя из другого, никакие другие арифметические операции над указателями недопустимы. Нельзя, например, прибавлять к указателям или вычитать из них значения **float** или **double**.

Чтобы наглядно представить себе результаты арифметики указателей, выполните приведенную ниже короткую программу. Она создает указатели на **int** (**i**) и **double** (**f**). Затем к этим указателям прибавляются значения от 0 до 9 и результаты выводятся на экран. Обратите внимание на то, как по ходу выполнения цикла изменяется каждый адрес в соответствии с базовым типом указателя. (Для большинства 32-разрядных компиляторов **i** будет увеличиваться на 4, а **f** — на 8). Заметьте, что при использовании указателя в предложении **cout** его значение автоматически выводится в формате, соответствующем архитектуре центрального процессора и операционной среде.

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

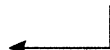
int main()
{
    int *i, j[10];
    double *f, g[10];
    int x;

    i = j;
    f = g;

    for(x=0; x<10; x++)
        cout << i+x << ' ' << f+x << '\n';

    return 0;
}
```

Вывод адресов, получаемых при прибавлении **x** к каждому указателю.



Вот пример пробного прогона (фактические значения могут отличаться от приведенных здесь).

```
0012FE5C 0012FE84
0012FE60 0012FE8C
0012FE64 0012FE94
0012FE68 0012FE9C
0012FE6C 0012FEA4
0012FE70 0012FEAC
0012FE74 0012FEB4
0012FE78 0012FEB8
0012FE7C 0012FEC4
0012FE80 0012FEC8
```

Сравнение указателей

Указатели можно сравнивать друг с другом, используя такие операторы отношения, как `==`, `<` и `>`. В общем случае результат операции сравнения указателей будет иметь смысл, только если оба указателя будут взаимосвязаны. Например, они могут указывать на разные элементы внутри одного и того же массива. (В Проекте 4-2 вы увидите пример такой ситуации.) Существует, однако, и другой вид сравнения указателей: любой указатель может сравниваться с нулевым указателем, который равен нулю.

Минутная тренировка

1. Вся арифметика указателей выполняется в соответствии с ____ ____ указателя.
 2. Считая, что тип **double** занимает 8 байт, на сколько увеличится значение указателя на **double**, если выполнить над ним операцию инкремента?
 3. При каких обстоятельствах (в общем случае) сравнение двух указателей имеет смысл?
 1. базовым типом
 2. Значение указателя будет увеличено на 8.
 3. В общем случае сравнение двух указателей имеет смысл, если они оба указывают на один и тот же объект, например, на один массив.
-

Цель

4.11. Указатели и массивы

В C++ имеется тесная взаимосвязь между указателями и массивами. Во многих случаях указатель и массив являются взаимозаменяемыми понятиями. Рассмотрим следующий фрагмент:

```
char str[80];  
char *p1;  
  
p1 = str;
```

Здесь **str** представляет собой массив из 80 символов, а **p1** — указатель на символы. Однако для нас более интересна третья строка. В ней **p1** присваивается адрес первого элемента массива **str**. (После присваивания **p1** будет указывать на **str[0]**.) И вот почему: в C++ использование имени массива без индекса образует указатель на первый элемент массива. Таким образом, операция

```
p1 = str;
```

присваивает указателю **p1** адрес элемента **str[0]**. Это очень важный момент: при использовании в выражении неиндексированного имени массива образуется указатель на первый элемент массива.

Поскольку после присваивания **p1** указывает на начало **str**, вы можете посредством **p1** получить доступ к элементам массива. Если, например, вы хотите обратиться к пятому элементу **str**, вы можете использовать обозначения

```
str[4]
```

или

```
*(p1+4)
```

Оба выражения дадут вам значение пятого элемента. Не забывайте, что индексы массива начинаются с нуля, поэтому при индексации **str** для доступа к пятому элементу используется индекс 4. Для обращения к пятому элементу посредством указателя **p1** к значению **p1** также добавляется 4, так как сам **p1** указывает на первый элемент **str**.

Круглые скобки, окружающие **p1+4** в последнем выражении, необходимы, потому что операция ***** имеет более высокий относительный приоритет, чем операция **+**. Если скобки опустить, выражение будет вычислять иначе: сначала будет найдено значение, на которое указывает **p1** (т. е. первый элемент массива), а затем к этому значению прибавится 4.

Фактически C++ предоставляет два метода обращения к элементам массива: с помощью *арифметики указателей* и с помощью *индексации массива*. Необходимо владеть обоими методами, так как иногда арифметика указателей работает быстрее индексации — особенно, если вы обращаетесь к элементам массива в строго последовательном порядке. Поскольку скорость часто является важнейшим критерием в программировании, обращение к элементам массивов посредством указателей весьма распространено в программах на C++. Кроме того, используя указатели вместо индексации массива, вы иногда можете написать более компактную программу.

Ниже приведен пример, демонстрирующий различия между индексацией массива и арифметикой указателей при обращении к элементам массива. Мы рассмотрим два варианта программы, которая будет выполнять преобразование букв в строке: прописных в строчные, а строчных — в прописные (обращение регистра). В первом варианте используется индексация. Второй вариант делает то же, но с помощью арифметики указателей. Первый вариант программы выглядит так:

```
// Обращение регистра посредством индексации массива.
```

```
#include <iostream>
```

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

int main()
{
    int i;
    char str[80] = "This Is A Test";

    cout << "Исходная строка: " << str << "\n";

    for(i = 0; str[i]; i++) {
        if(isupper(str[i]))
            str[i] = tolower(str[i]);
        else if(islower(str[i]))
            str[i] = toupper(str[i]);
    }

    cout << "Преобразованная строка: " << str;

    return 0;
}
```

Вывод программы выглядит так:

```
Исходная строка: This Is A Test
Преобразованная строка: tHIS iS a tEST
```

Обратите внимание на то, что программа с целью определения регистра использует библиотечные функции **isupper()** и **islower()**. Функция **isupper()** возвращает **true**, если ее аргумент является прописной буквой; функция **islower()** возвращает **true**, если ее аргумент – строчная буква. Внутри цикла **for** выполняется индексация массива **str**, и каждая буква анализируется и преобразуется. Цикл повторяется до тех пор, пока не встретится завершающий строку ноль. Поскольку ноль равнозначен **false**, цикл завершается.

Теперь рассмотрим вариант программы с арифметикой указателей:

```
// Обращение регистра посредством арифметики указателей.
```

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

int main()
{
    char *p;
```

```

char str[80] = "This Is A Test";

cout << "Исходная строка: " << str << "\n";

p = str; // присвоим p адрес начала массива

while(*p) {
    if(isupper(*p))
        *p = tolower(*p);
    else if(islower(*p)) ← Обращение к str посредством указателя.
        *p = toupper(*p);
    p++;
}

cout << " Преобразованная строка: " << str;

return 0;
}

```

В этом варианте программы **p** устанавливается на начало **str**. Затем внутри цикла **while** буква, на которую указывает **p**, анализируется и преобразуется, а затем выполняется инкремент **p**. Цикл завершается, когда **p** будет указывать на завершающий строку **str** ноль.

Правила, по которым компиляторы образуют объектный код, приводят к тому, что эти две программы могут различаться по производительности. В общем случае индексация массива требует большего числа машинных команд, чем выполнение арифметических преобразований с указателем. Соответственно, в профессионально написанной C++-программе вы чаще увидите вариант с указателями. Однако в начале обучения языку вы вполне можете использовать индексацию массивов, пока не приобретете необходимые навыки для работы с указателями.

Индексация указателя

Как вы только что видели, для обращения к массиву можно использовать арифметику указателей. Как это ни странно, но обратное тоже справедливо. В C++ возможно индексировать указатель, как если бы он был массивом. Вот пример индексации указателя. Это третий вариант программы обращения регистра:

```
// Индексация указателя, как если бы он был массивом.
```

```

#include <iostream>
#include <cctype>

```



```

using namespace std;

int main()
{
    char *p;
    int i;
    char str[80] = " This Is A Test";

    cout << "Исходная строка: " << str << "\n";

    p = str; // присвоим p адрес начала массива

    // теперь индексируем p
    for(i = 0; p[i]; i++) {
        if(isupper(p[i]))
            p[i] = tolower(p[i]);
        else if(islower(p[i])) ← Использование указателя p, как
                                если бы он был массивом.
            p[i] = toupper(p[i]);
    }

    cout << "Преобразованная строка: " << str;

    return 0;
}

```

Программа создает указатель на `char` с именем `p`, и присваивает затем этому указателю адрес первого элемента `str`. Внутри цикла `for` указатель `p` индексируется с использованием обычного синтаксиса индексации массива. Такая операция вполне допустима, так как в C++ выражение `p[i]` функционально идентично выражению `*(p+i)`. Это обстоятельство еще раз иллюстрирует тесную взаимосвязь между указателями и массивами.

Минутная тренировка

1. Можно ли к массиву обратиться посредством указателя?
 2. Можно ли индексировать указатель, как если бы он был массивом?
 3. Что обозначает имя массива, указанное без индекса?
 1. Да, к массиву можно обратиться посредством указателя.
 2. Да, указатель можно индексировать так же, как и массив.
 3. Имя массива, указанное без индекса, обозначает адрес первого элемента этого массива.
-

Спросим у эксперта

Вопрос: Можно ли сказать, что указатели и имена массивов взаимозаменяемы?

Ответ: Как было показано на предыдущих страницах, указатели и массивы тесно взаимосвязаны и их имена во многих случаях взаимозаменяемы. Например, имея указатель, указывающий на начало массива, можно обращаться к элементам этого массива как посредством арифметики указателя, так и с помощью индексации. Однако нельзя сказать, чтобы имена указателей и массивов были полностью взаимозаменяемы. Рассмотрим, например, следующий фрагмент:

```
int nums[10];
int i;

for(i=0; i<10; i++) {
    *nums = i; // это правильно
    nums++; // ОШИБКА - nums нельзя модифицировать
}
```

В этом примере **nums** представляет собой массив целых чисел. Как указано в комментариях, хотя вполне допустимо использовать с **nums** оператор ***** (операция с указателем), было бы грубой ошибкой модифицировать значение **nums**. Дело в том, что **nums** является константой, которая указывает на начало массива. Применить к ней операцию инкремента нельзя. Несмотря на то, что имя массива, указанное без индекса, действительно образует указатель на начало массива, изменение этого указателя недопустимо.

Хотя имя массива образует указатель-константу, его все же можно использовать в качестве части указательного выражения, если только не пытаться его модифицировать. Например, следующая строка представляет собой вполне допустимое предложение, которое присваивает элементу **nums[3]** значение 100:

```
*(nums+3) = 100; // Это правильно, поскольку nums не
                  // изменяется
```

Строковые константы

Посмотрим, как C++ работает со строковыми константами, вроде той, что используется в приведенном ниже предложении:

```
cout << strlen("Ксантиф");
```

Когда компилятор сталкивается со строковой константой, он сохраняет ее в *таблице строк* данной программы и создает указатель на эту строку. В результате “Ксантиф” образует указатель на начало этой фразы в таблице строк программы. Поэтому приведенная ниже программа составлена вполне разумно; она выводит фразу “Указатели увеличивают возможности C++”:

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

```
int main()
```

```
{
```

```
    char *ptr;
```

```
    ptr = " Указатели увеличивают возможности C++. \n";
```

```
    cout << ptr;
```

```
    return 0;
```

```
}
```

Указателю ptr присваивается адрес этой строковой константы.



В этой программе символы, образующие строковую константу, хранятся в таблице строк, и указателю `ptr` присваивается адрес нашей строки в этой таблице.

Поскольку указатель на таблицу строк программы создается автоматически при использовании строковой константы, у вас может появиться искушение воспользоваться этим указателем для модификации содержимого таблицы строк. Однако это рискованное занятие, так как многие компиляторы C++ создают оптимизированные таблицы строк, в которых одна строковая константа может использоваться в нескольких местах вашей программы. В этом случае изменение строки может привести к нежелательным побочным эффектам.

Проект 4-2 Переворачивание строки

Выше уже отмечалось, что сравнение одного указателя с другим имеет смысл, только если они оба указывают на один и тот же объект, например, массив. Теперь, когда мы рассмотрели взаимоотношение указателей и массивов, мы можем использовать сравнение указателей для рационализации некоторых типов алгоритмов. В этом проекте вы увидите пример такого рода.

Рассматриваемая здесь программа осуществляет переворачивание строки, оставляя ее на том же месте. Вместо копирования строки задом наперед в другой массив, программа обращает содержимое строки прямо внутри содержащего эту строку массива. Для выполнения этой операции в программе используются две переменных-указателя. Один указатель первоначально указывает на начало строки, второй — на ее последний символ. В программе также организован цикл, который выполняется до тех пор, пока начальный указатель меньше конечного.

В каждом шаге цикла символы, на которые указывают указатели, обмениваются местами, после чего указатели продвигаются (один вперед, а другой назад). Когда начальный указатель становится равным или большим, чем конечный, цикл завершается, а строка оказывается перевернутой.

Шаг за шагом

1. Создайте файл с именем **StrRev.cpp**.
2. Начните с включения в файл следующих строк:

```
/*
    Проект 4-2
    Переворачивание строки.
*/

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

int main()
{
    char str[] = "это проверка работы программы";
    char *start, *end;
    int len;
    char t;
```

Обращаемая строка содержится в **str**. Указатели **start** и **end** используются для обращения к символам строки.

3. Добавьте приведенные ниже строки, которые выводят исходную строку, определяют ее длину и устанавливают начальные значения указателей **start** и **end**:

```
cout << "Исходная: " << str << "\n";

len = strlen(str);

start = str;
end = &str[len-1];
```

Обратите внимание на то, что **end** указывает на последний символ строки, а не на завершающий ноль.

4. Добавьте фрагмент, обращающий строку:

```
while(start < end) {
    // обменяем символы
```

```
t = *start;
*start = *end;
*end = t;
// продвинем указатели
start++;
end--;
}
```

Обмен осуществляется следующим образом. Шаги цикла повторяются, пока адрес памяти, содержащийся в указателе **start**, меньше адреса, содержащегося в указателе **end**. Внутри цикла символы, на которые указывают **start** и **end**, обмениваются местами. Затем выполняется инкремент указателя **start** и декремент указателя **end**. Когда **start** становится больше или равным **end**, процесс прекращается, так как все символы строки поменялись местами. Поскольку оба указателя, и **start**, и **end**, указывают на один и тот же массив, их можно сравнивать друг с другом.

5. Ниже приведен полный текст программы **StrRev.cpp**:

```
/*
    Проект 4-2
    Переворачивание строки.
*/

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

int main()
{
    char str[] = "это проверка работы программы";
    char *start, *end;
    int len;
    char t;

    cout << " Исходная: " << str << "\n";

    len = strlen(str);

    start = str;
    end = &str[len-1];

    while(start < end) {
        // обменяем символы
        t = *start;
        *start = *end;
```

```
*end = t;  
  
// продвинем указатели  
start++;  
end--;  
}  
  
cout << "Перевернутая: " << str << "\n";  
return 0;  
}
```

Ниже приведен вывод программы:

Исходная: это проверка работы программы

Перевернутая: ыммаргорп ытобар акреворп отэ

Массивы указателей

Указатели, как и любые другие типы данных, могут образовывать массивы. Например, объявление массива из 10 указателей типа `int` выглядит так:

```
int *pi[10];
```

Здесь `pi` является именем массива из десяти указателей на целые числа.

Для присваивания адреса переменной типа `int` с именем `var` третьему элементу массива указателей, вы должны написать:

```
int var;  
pi[2] = &var;
```

Не забывайте, что `pi` является массивом указателей на `int`. Единственный вид объектов, которые могут содержаться в элементах этого массива — это адреса целочисленных переменных (не самих переменных!).

Чтобы найти значение `var`, вы должны написать:

```
*pi[2]
```

Как и в случае любых других массивов, массив указателей можно инициализировать. Обычное использование инициализированных массивов указателей — хранение указателей на строки. Вот пример, в котором используется двумерный массив указателей на символы для реализации небольшого толкового словаря:

```

// Использование двумерного массива указателей для создания
// словаря.

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

int main() {

    char *dictionary[][2] = {
        "карандаш", "Инструмент для рисования.",
        "клавиатура", "Устройство ввода.",
        "ружье", "Плечевое огнестрельное оружие.",
        "Самолет", "Воздушное судно с неподвижными крыльями.",
        "сеть", "Группа соединенных между собой компьютеров.",
        "", ""
    };
    char word[80];
    int i;

    cout << "Введите слово: ";
    cin >> word;

    for(i = 0; *dictionary[i][0]; i++) {
        if(!strcmp(dictionary[i][0], word)) {
            cout << dictionary[i][1] << "\n";
            break;
        }
    }

    if(!*dictionary[i][0])
        cout << word << " не найдено.\n";

    return 0;
}

```

Это двумерный массив указателей на **char**, который используется для адресации пар строк

Для поиска определения слова в **dictionary** ищется **word**. Если соответствие найдено, определение выводится на экран

Вот пример прогона программы:

Введите слово: сеть
Группа соединенных между собой компьютеров.

При создании массива **dictionary** он инициализируется набором слов и их определений. Вспомним, что C++ записывает все строковые константы в таблицу строк, связанную с вашей программой, поэтому массив будет содержать лишь указатели на строки. Программа сравнивает введенное пользователем слово со строками, хранящимися в

словаре. Если обнаруживается совпадение, на экран выводится определение соответствующего слова. Если совпадения не обнаруживается, выводится сообщение об ошибке.

Обратите внимание на то, что **dictionary** заканчивается двумя пустыми строками. Они отмечают конец массива. Вспомним, что пустая строка содержит только завершающий ноль. Цикл **for** выполняется до тех пор, пока первый символ строки не окажется нулем. Это условие обнаруживается с помощью выражения

```
*dictionary[i][0]
```

Индексы массива задают указатель на строку. Знак ***** позволяет получить символ, расположенный по этому адресу. Если этот символ равен нулю, то выражение ложно и цикл завершается. В противном случае выражение истинно и цикл продолжает свое выполнение.

Соглашение о нулевом указателе

После того, как указатель объявлен, но до того, как ему будет присвоен какой-то разумный адрес, он будет содержать произвольное значение. Попытка использовать указатель перед его инициализацией с большой вероятностью приведет к аварийному завершению программы. Поскольку нет надежного способа избежать использования неинициализированного указателя, программисты на C++ приняли процедуру, которая помогает предотвратить некоторые ошибки. По принятому соглашению, если указатель содержит нулевое значение, то считается, что он никуда не указывает. Таким образом, если всем неопределенным пока указателям присвоить нулевое значение, а программу написать так, чтобы она не использовала нулевые указатели, вы сможете избежать случайного обращения к неинициализированному указателю. Это весьма полезный прием, которым стоит пользоваться.

Указатель любого типа может быть инициализирован нулем при его объявлении. Например, следующее предложение инициализирует **p** нулем:

```
float *p = 0; //p теперь нулевой указатель
```

Для отбраковки нулевых указателей пользуйтесь предложениями **if** вроде следующих:

```
if(p) // далее выполнение, если p не ноль
```

```
if(!p) // далее выполнение, если p ноль
```


Минутная тренировка

1. Строковые константы, используемые в программе, хранятся в ____.
2. Что создает это объявление: `float *fpa[18];`
3. По соглашению указатель, содержащий ноль, считается неиспользуемым. Справедливо ли это утверждение?
 1. таблице строк
 2. Объявление создает массив из 18 указателей на `float`.
 3. Справедливо.

Цель

4.12. Указатель на указатель

Указатель на указатель представляет собой форму *вложенной косвенности*, или цепочки указателей. Рассмотрим рис. 4-2. Легко видеть, что в случае обычного указателя значение указателя представляет собой адрес переменной (или, можно сказать, адрес значения). В случае указателя на указатель первый указатель содержит адрес второго указателя, а тот уже указывает на ячейку памяти, содержащую требуемое значение.

Вложенная косвенность может иметь любую глубину вложенности, однако случаи, когда требуется иметь более глубокую вложенность, чем указатель на указатель, редки, да и вообще вряд ли стоит такие конструкции создавать. Излишне глубокая вложенность затрудняет понимание программы и легко может привести к принципиальным ошибкам.

Переменную, которая должна служить указателем на указатель, следует таковой и объявить. Это делается помещением дополнительной звездочки перед ее именем. Например, приведенное ниже объяв-

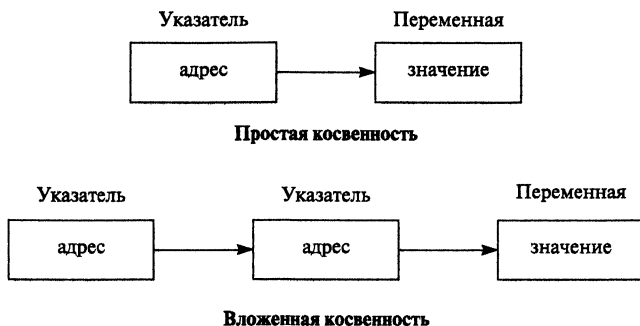


Рис. 4-2. Простая и вложенная косвенность

ление создает переменную **balance**, которая является указателем на указатель типа **int**:

```
int **balance;
```

Важно понимать, что **balance** является не указателем на целое, а указателем на указатель на целое.

Когда значение-мишень адресуется посредством указателя на указатель, обращение к этому значению требует применения оператора ***** дважды, как это показано в следующем коротком примере:

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

int main()
{
    int x, *p, **q;

    x = 10;

    p = &x; ←————— Присваивает p адрес x.

    q = &p; ←————— Присваивает q адрес p.

    cout << **q; // выводит значение x ←—————
    return 0;
}
```

Обращение к значению x посредством q
Заметьте, что требуются два знака *.

Здесь **p** объявляется, как указатель на целое, а **q** — как указатель на указатель на целое. Предложение **cout** выведет на экран число 10.

Спросим у эксперта

Вопрос: Учитывая большие возможности указателей, я могу предвидеть, что случайные ошибки в их использовании могут привести к серьезным нарушениям в работе программы. Можете ли вы порекомендовать, как избежать ошибок при работе с указателями?

Ответ: Во-первых, перед тем, как использовать переменные-указатели, убедитесь, что они инициализированы. Указатель должен на что-нибудь указывать, если вы хотите использовать его по назначению! Во-вторых, следите за тем, чтобы тип объекта, на который указывает указатель, совпадал с базовым типом указателя. В-третьих, не выполняйте никаких операций посредством нулевого указателя. Не забывайте: нулевой указатель означает, что указатель ни на что не указывает. Наконец, никогда не приво-

дите указатели к другому типу “просто чтобы программа нормально компилировалась”. Как правило, сообщения о несоответствии указателей говорят о том, что вы неправильно представляете себе какие-то элементы программы. Приведение одного типа указателя к другому обычно требуется только в необычных ситуациях.

✓ Вопросы для самопроверки

1. Покажите, как объявить массив **hightemps** типа **short int** размером 31 элемент.
2. В C++ индексы любого массива начинаются с _____.
3. Напишите программу, которая просматривает массив из 10 целых чисел в поисках одинаковых значений. Все найденные пары программа должна выводить на экран.
4. Что такое строка, завершающаяся нулем?
5. Напишите программу, которая запрашивает у пользователя две строки, а затем сравнивает эти строки, не обращая внимания на регистр букв. В этом случае “ok” и “OK” будут считаться равными.
6. Насколько велик должен быть массив-приемник при использовании функции **strcat()**?
7. Как задается каждый индекс в многомерном массиве?
8. Покажите, как инициализировать массив **nums** типа **int** значениями 5, 66 и 88.
9. В чем заключается принципиальное преимущество объявления массивов неопределенной длины?
10. Что такое указатель? Каковы два оператора указателей?
11. Допустимо ли индексировать указатель так, как это делается с массивом? Можно ли обратиться к массиву посредством указателя?
12. Напишите программу, которая подсчитывает число прописных букв в строке и выводит результат на экран.
13. Как называется ситуация, когда один указатель указывает на другой указатель?
14. Каково значение нулевого указателя в C++?

Модуль 5 Введение в функции

Цели, достигаемые в этом модуле

- 5.1 Познакомиться с общей формой определения функции
- 5.2 Научиться создавать собственные функции
- 5.3 Освоить использование аргументов функции
- 5.4 Понять, что такое возвращаемое функцией значение
- 5.5 Научиться использовать функции в выражениях
- 5.6 Узнать о локальной области видимости
- 5.7 Узнать о глобальной области видимости
- 5.8 Освоить передачу в функцию указателей и массивов
- 5.9 Освоить возврат из функции указателей
- 5.10 Рассмотреть передачу в функцию `main()` аргументов командной строки
- 5.11 Узнать о прототипах функций
- 5.12 Научиться создавать рекурсивные функции

В этом модуле мы начинаем серьезное изучение функций. Функции можно назвать строительными блоками C++, и отчетливое понимание механизма функций абсолютно необходимо для того, чтобы стать успешным программистом на C++. Здесь вы узнаете, как создать функцию. Вы также познакомитесь с передачей аргументов, возвратом из функции значения, локальными и глобальными переменными, прототипами функций и рекурсией.

Основы функций

Функцией называется подпрограмма, содержащая одно или больше предложений C++ и выполняющая конкретную задачу. До сих пор все программы, которые вы писали, использовали единственную функцию `main()`. Функции называются строительными блоками C++, потому что любая программа представляет собой набор функций. Все “предложения действий” вашей программы находятся внутри функций. Таким образом, функция содержит предложения, которые вы обычно рассматриваете, как выполняемую часть программы.

Хотя очень простые программы, вроде тех, что приводятся в настоящей книге в качестве примеров, могут содержать единственную функцию `main()`, большинство программ включает в себя несколько функций. Большие коммерческие программы могут определять сотни функций.

Цель

5.1. Общая форма определения функции

Все функции C++ имеют общую форму, приведенную ниже:

```
тип-возврата имя(список-параметров)
{
    // тело функции
}
```

Здесь *тип-возврата* определяет тип данного, возвращаемого функцией. Функция может возвращать любой тип за исключением массива. Если функция ничего не возвращает, то тип возврата должен быть `void`. Имя функции определяется элементом *имя*. В качестве имени может использоваться любой допустимый идентификатор, если он еще не занят. *список-параметров* представляет собой последовательность пар типов и идентификаторов, разделяемых запятыми. Параметры — это переменные, которые получают значения аргументов, передаваемых

функции при ее вызове. Если функция не требует параметров, список параметров будет пуст.

Фигурные скобки окружают тело функции. Тело функции состоит из предложений C++, определяющих, что именно эта функция делает. Когда по ходу этих предложений встречается закрывающая фигурная скобка, функция завершается, и управление передается вызывающему коду.

Цель

5.2. Создание функции

Создать функцию очень просто. Поскольку все функции имеют общую форму, они все структурно схожи с функцией `main()`, которую вы уже использовали. Начнем с простого примера, содержащего две функции: `main()` и `myfunc()`. Перед тем, как запустить эту программу (или ознакомиться с приведенным далее анализом ее работы), попробуйте сами сообразить, что эта функция выведет на экран.

```
// Эта программа содержит две функции: main() и myfunc().
```

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

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

← Это прототип функции `myfunc()`.

```
int main()
{
    cout << "В main()\n";

    myfunc(); // вызов myfunc()

    cout << "Снова в main()\n";

    return 0;
}
```

```
// Это определение функции.
```

```
void myfunc()
{
    cout << "Внутри myfunc()\n";
}
```

— Это функция с именем `myfunc()`.

Программа работает следующим образом. Начинается все с функции `main()`, в которой выполняется первое предложение `cout`. Затем `main()` вызывает функцию `myfunc()`. Посмотрите, как это делается:

указывается имя функции, за которым стоит пара круглых скобок. В данном случае вызов функции представляет собой предложение, и как и всякое предложение, оно должно заканчиваться точкой с запятой. Далее **myfunc()** выполняет предложение **cout** и, встретив закрывающую фигурную скобку **}**, возвращается в **main()**. В **main()** выполнение возобновляется на той строке кода, которая непосредственно следует за вызовом **myfunc()**. Наконец, **main()** выполняет второе предложение **cout** и завершается. Вот вывод этой программы:

```
В main()  
Внутри myfunc()  
Снова в main()
```

Способ, которым вызывается **myfunc()**, и способ, которым осуществляется возврат в **main()**, являются типичными представителями процедуры, приложимой ко всем функциям. Для вызова функции следует указать ее имя с парой круглых скобок. Когда вызывается функция, управление передается в эту функцию. Далее выполнение продолжается внутри функции, пока не встретится закрывающая фигурная скобка. Завершение работы функции приводит к передаче управления назад в вызывающий код на то предложение, которое стоит непосредственно за вызовом функции.

Обратите внимание на предложение в приведенной выше программе:

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

Как указано в комментарии, это *прототип* функции **myfunc()**. Прототипы будут детально рассмотрены ниже, а здесь только несколько слов. Прототип функции объявляет функцию перед ее определением. Прототип позволяет компилятору узнать тип возвращаемого функцией значения, а также число и типы параметров этой функции. Компилятор должен иметь всю эту информацию к тому моменту, когда он столкнется с первым вызовом функции. Именно поэтому прототип располагается перед **main()**. Единственная функция, которой не требуется прототип — это главная функция **main()**, прототип которой предопределен в C++.

Ключевое слово **void**, которое указывается как перед прототипом **myfunc()**, так и перед ее определением, формально устанавливает, что **myfunc()** не возвращает никакого значения. В C++ функции, не возвращающие значения, объявляются **void**.

Цель

5.3. Использование аргументов

В созданную вами функцию можно передать одно или несколько значений. Значение, передаваемое в вызываемую функцию, называет-

ся *аргументом*. Аргументы позволяют передать в функцию некоторую информацию.

Когда вы создаете функцию, которая принимает один или больше аргументов, переменные, которые будут принимать эти аргументы, так же должны быть объявлены. Эти переменные называются *параметрами* функции. Ниже приведен пример определения функции с именем **box()**, которая вычисляет объем коробки и выводит результат на экран. Она имеет три параметра, характеризующие длину (**length**), ширину (**width**) и высоту (**height**) коробки:

```
void box(int length, int width, int height)
{
    cout << "объем коробки равен " << length * width * height <<
        "\n";
}
```

Каждый раз, когда вызывается функция **box()**, она вычисляет объем коробки, умножая значения, передаваемые ей в переменные **length**, **width** и **height**.

Вызывая **box()**, вы должны задать три аргумента, например:

```
box(7, 20, 4);
box(50, 3, 2);
box(8, 6, 9);
```

Значения, указанные между круглыми скобками, представляют собой аргументы, передаваемые в **box ()**, причем значение каждого аргумента копируется в соответствующий ему параметр. Таким образом, при первом вызове **box ()** 7 копируется в **length**, 20 копируется в **width**, а 4 копируется в **height**. При втором вызове **box ()** 50 копируется в **length**, 3 копируется в **width**, а 2 копируется в **height**. При третьем вызове 8 копируется в **length**, 6 копируется в **width**, а 9 копируется в **height**.

Приведенная ниже программа демонстрирует использование функции **box()**:

```
// Простая программа, демонстрирующая использование box().

#include <iostream>
using namespace std;

void box(int length, int width, int height); // Прототип box()

int main()
{
    box(7, 20, 4); ←————— Аргументы, передаваемые в box( ).
```



```

    box(50, 3, 2);
    box(8, 6, 9);

    return 0;
}
// Вычисление объема коробки.
void box(int length, int width, int height)
{
    cout << "объем коробки равен " << length * width * height <<
    "\n";
}

```

Эти параметры принимают значения аргументов, передаваемых в `box()`.

Вывод программы:

```

объем коробки равен 560
объем коробки равен 300
объем коробки равен 432

```



Совет

Не забывайте, что термин *аргумент* относится к значению, которое используется в вызове функции. Переменная, принимающая значение аргумента, называется *параметром*. Между прочим, функции, принимающие при вызове аргументы, носят название *параметризованных функций*.

Минутная тренировка

1. Что происходит с выполнением программы при вызове функции?
2. Как различаются аргументы и параметры?
3. Если функция требует параметр, где он объявляется?
1. Когда вызывается функция, выполнение программы переходит в функцию. Когда функция завершается, выполнение программы возвращается в вызывающий код на предложение, непосредственно следующее за вызовом функции.
2. Аргументом называется значение, передаваемое функции. Параметр — это переменная, которая принимает это значение.
3. Параметры объявляются после имени функции, внутри круглых скобок.

Цель

5.4. Использование предложения return

В предыдущих примерах возврат из функции в вызывающий код осуществлялся, когда в тексте функции встречалась закрывающая фи-

гурная скобка. Хотя для многих функций это вполне допустимо, в ряде случаев такой метод возврата работать не будет. Часто требуется явным образом указать, как и когда функция должна завершить свою работу. Для этого используется предложение **return**.

Предложение **return** имеет две формы: одна позволяет вернуть значение, другая не делает этого. Мы начнем с варианта предложения **return**, который не возвращает значения. Если функция имеет тип возврата **void** (т. е. функция не возвращает никакого значения), используется следующая форма предложения **return**:

```
return;
```

Если в тексте функции встречается предложение **return**, выполнение немедленно возвращается в вызывающий код. Любые строки, остающиеся в функции, игнорируются. Рассмотрим, например, следующую программу:

```
// Использование предложения return.
```

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

```
void f();
```

```
int main()
{
    cout << "Перед вызовом\n";

    f();

    cout << "После вызова\n";

    return 0;
}
```

```
// функция void, использующая return.
```

```
void f()
{
    cout << "Внутри f()\n";

    return; // возврат в вызывающий код

    cout << "Это не будет выводиться.\n";
}
```

Это вызывает немедленный
возврат с обходом остав-
шегося предложения **cout**.

Ниже приведен вывод программы:

Перед вызовом
Внутри `f()`
После вызова

Как видно из вывода программы, `f()` возвращается в `main()`, как только в `f()` встречается предложение `return`. Второе предложение `cout` выполняться никогда не будет.

Вот более полезный пример использования предложения `return`. Функция `power()`, включенная в приведенную ниже программу, выводит результат возведения целого числа в целую положительную степень. Если показатель степени отрицателен, предложение `return` заставляет функцию завершиться, не пытаясь вычислить результат.

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

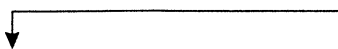
void power(int base, int exp);

int main()
{
    power(10, 2);
    power(10, -2);

    return 0;
}

// Возведение целого числа в целую положительную степень.
void power(int base, int exp)
{
    int i;
    if(exp < 0) return; /* Не умею работать с отрицательными
                        *показателями. */
    i = 1;
    for( ; exp; exp--) i = base * i;
    cout << "Ответ: " << i;
}
```

— Возврат, если `exp` отрицательно.



Вот вывод программы:

Ответ: 100

Если `exp` отрицательно (как это имеет место во втором вызове), `power()` возвращается, обходя весь остаток функции.

Функция может содержать несколько предложений `return`. Как только по ходу выполнения функции встречается одно из них, функция завершается. Например, приведенный ниже фрагмент вполне правилен:

```
void f()
{
    // ...

    switch(c) {
        case 'a': return;
        case 'b': // ...
        case 'c': return;
    }
    if(count < 100) return;
    // ...
}
```

Учтите, однако, что слишком много предложений **return** ухудшают структуру функции и затуманивают ее смысл. Использовать несколько предложений **return** имеет смысл лишь тогда, когда при этом текст функции становится более понятным.

Возвращаемые значения

Функция может вернуть значение в вызывающий ее код. Возвращаемое значение представляет собой способ получить информацию из функции. Для возврата значения следует использовать второй вариант предложения **return**:

`return значение;`

Здесь *значение* — это то значение, которое возвращается функцией в вызывающий ее код. Эта форма предложения **return** может использоваться только с функциями, не возвращающими **void**.

Функция, возвращающая значение, должна задать тип этого значения. Тип возврата должен быть совместим с типом данного, используемого в предложении **return**. Если это не так, во время компиляции будет зафиксирована ошибка. В объявлении функции может быть указан любой допустимый в C++ тип данного, за исключением того, что функция не может вернуть массив.

Для иллюстрации работы с функциями, возвращающими значения, перепишем функцию `box()` так, как это показано ниже. В этом варианте `box()` возвращает вычисленный объем. Обратите внимание на то, что помещение функции с правой стороны знака равенства присваивает возвращаемое значение переменной.

```
// Возврат значения.
```

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

```

int box(int length, int width, int height); // вернем объем

int main()
{
    int answer;
    answer = box(10, 11, 3); // присвоим возвращаемое значение
    cout << "Объем равен " << answer;

    return 0;
}

// Эта функция возвращает значение.
int box(int length, int width, int height)
{
    return length * width * height ;
}

```

Значение, возвращаемое из функции **box()**, присваивается переменной **answer**.

Вернем объем.

Вот вывод программы:

Объем равен 330

В этом примере **box()** возвращает значение **length * width * height** с помощью предложения **return**. Это значение присваивается переменной **answer**. Другими словами, значение, возвращаемое предложением **return**, становится значением функции **box()** в вызывающей программе.

Поскольку функция **box()** возвращает значение, ее объявление не предваряется ключевым словом **void**. (Вспомним, что **void** используется только если функция *не* возвращает значения.) Вместо этого **box()** объявлена, как возвращающая значение типа **int**. Обратите внимание на то, что тип возвращаемого функцией значения указывается перед ее именем как в прототипе, так и в определении.

Разумеется, **int** не является единственным типом, который может возвращаться функцией. Как уже отмечалось, функция может вернуть любой тип за исключением массива. Например, в приведенной ниже программе функция **box()** модифицирована так, что она принимает параметры типа **double** и возвращает значение типа **double**:

```

// Возврат значения типа double.

#include <iostream>
using namespace std;

// используем данные типа double

```

```
double box(double length, double width, double height);

int main()
{
    double answer;

    answer = box(10.1, 11.2, 3.3); // присвоим возвращаемое
                                   // значение
    cout << "Объем равен " << answer;

    return 0;
}

// Этот вариант box использует данные типа double.
double box(double length, double width, double height)
{
    return length * width * height ;
}.
```

Вот вывод этой программы:

Объем равен 373.296

Еще одно замечание: Если функция типа **не-void** осуществляет возврат, встретив закрывающую фигурную скобку, возвращается неопределенное (т. е. неизвестное) значение. В силу своеобразной особенности формального синтаксиса C++, в **не-void** функциях нет необходимости обязательно выполнять предложение **return**. Такое может случиться, если конец функции достигнут прежде, чем в ней встретилось предложение **return**. Поскольку, однако, функция объявлена как возвращающая значение, значение все же будет возвращено — несмотря на то, что оно является “мусором”. Разумеется, хороший стиль программирования требует, чтобы создаваемые вами **не-void** функции возвращали значение посредством явного выполнения предложения **return**.

Цель

5.5. Использование функций в выражениях

В предыдущем примере значение, возвращенное функцией **box()**, было присвоено переменной, а затем значение этой переменной выводилось на экран посредством предложения **cout**. Хотя такое построение программы нельзя назвать неправильным, однако более эффективно было бы использовать возвращаемое значение непо-

средственно в предложении **cout**. Так, функцию **main()** из предыдущей программы можно более эффективно переписать следующим образом:

```
int main()
{
    // используем возвращаемое из box( ) значение непосредственно
    cout << "Объем равен " << box(10.1, 11.2, 3.3);
    return 0;
}
```

Используем возвращаемое из **box()** значение непосредственно в предложении **cout**.

Когда выполняется предложение **cout**, автоматически вызывается функция **box()**, что дает возможность получить ее возвращаемое значение. Это значение и выводится на экран. Нет никакого смысла сначала присваивать это значение какой-то переменной.

Не-void функции допустимо использовать в любых выражениях. При вычислении выражения функция вызывается автоматически, и ее возвращаемое значение используется в выражении. Например, приведенная ниже программа суммирует объемы трех коробок и выводит на экран среднее значение объема:

```
// Использование box() в выражении.

#include <iostream>
using namespace std;

// используем данные типа double
double box(double length, double width, double height);
int main()
{
    double sum;

    sum = box(10.1, 11.2, 3.3) + box(5.5, 6.6, 7.7) +
          box(4.0, 5.0, 8.0);

    cout << "Сумма объемов равна " << sum << "\n";
    cout << "Среднее значение объема равно " << sum / 3.0 << "\n";

    return 0;
}

// Этот вариант box использует данные типа double.
double box(double length, double width, double height)
{
    return length * width * height ;
}
```

Вывод этой программы:

Сумма объемов равна 812.806

Среднее значение объема равно 270.935

Минутная тренировка

1. Приведите две формы предложения **return**.
 2. Может ли **void**-функция возвращать значение?
 3. Может ли функция быть частью выражения?
1. Вот две формы **return**:
`return;`
`return значение;`
 2. Нет, **void**-функция не может возвращать значений.
 3. Вызов не-**void** функции может быть использован в выражении. Когда это происходит, функция выполняется, и ее возвращаемое значение используется в выражении.
-

Правила видимости

До сих пор мы использовали переменные без строгого обсуждения, где они могут быть объявлены, как долго они существуют, и из каких частей программы они доступны. Эти атрибуты устанавливаются *правилами видимости*, определенными в C++.

В целом правила видимости языка управляют видимостью и временем жизни объекта. C++ определяет целую систему областей видимости (областей действия); важнейшими из них являются две: *локальная* и *глобальная*. Переменные можно объявлять и в той, и в другой области. В настоящем подразделе вы увидите, чем различаются переменные, объявленные в локальной области видимости, от переменных, объявленных в глобальной области, и как эти области связаны с функциями.

Цель

5.6. Локальная область видимости

Локальная область видимости создается посредством блока. (Вспомним, что блок начинается открывающей фигурной скобкой и заканчивается закрывающей фигурной скобкой.) Таким образом, каждый раз, когда вы начинаете новый блок, вы создаете новую область видимости. Переменную можно объявить в любом блоке. Переменная, объявленная внутри блока, называется *локальной переменной*.

Локальную переменную допустимо использовать только в предложениях, расположенных в том блоке, где она была объявлена. То же самое можно выразить иначе: локальные переменные неизвестны вне их собственного блока кода. Таким образом, предложения, расположенные вне блока, не могут обращаться к объектам, определенным внутри него. Суть дела заключается в том, что когда вы объявляете локальную переменную, вы локализуете эту переменную и защищаете ее от неавторизованного доступа или модификации. Заметим, что правила видимости создают предпосылки для инкапсуляции.

Одно из важнейших правил, касающихся локальных переменных, заключается в том, что они существуют только пока выполняется блок кода, в котором они были объявлены. Локальная переменная создается в тот момент, когда выполнение доходит до предложения объявления этой переменной, и уничтожается, когда блок завершает свое выполнение. Поскольку локальная переменная уничтожается при выходе из блока, ее значение теряется.

Наиболее распространенным программным блоком, в котором объявляются локальные переменные, является функция. Каждая функция определяет блок кода, который начинается с открывающей функцию фигурной скобки и заканчивается на закрывающей скобке, закрывающей функцию. И код, и данные, входящие в функцию, принадлежат этой функции, и к ним не может обратиться никакое предложение в любой другой функции, кроме как путем вызова этой функции. (Например, невозможно, используя предложение `goto`, перейти в середину другой функции.) Тело функции скрыто от остальной программы, и на него нельзя воздействовать из других частей программы, так же, как и оно не может воздействовать ни на какую другую часть программы. Таким образом, содержимое одной функции полностью отделено от содержимого другой. Можно сказать и по-другому: код и данные, определенные в одной функции, не могут взаимодействовать с кодом и данными, определенными в другой функции, потому что эти две функции имеют различные области видимости.

Из-за того, что каждая функция определяет свою собственную область видимости, переменные, объявленные внутри одной функции, никак не влияют на переменные, объявленные в другой функции, даже если у этих переменных одинаковые имена. Рассмотрим, например, такую программу:

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

```
void f1();
```

```
int main()
{
```

```
    int val = 10; ← Эта val локальна по отношению к main( ).
```

```

    cout << "val в main(): " << val << '\n';
    f1();
    cout << "val в main(): " << val << '\n';

    return 0;
}

void f1()
{
    int val = 88; ← Эта val локальна по отношению к f1().

    cout << "val в f1(): " << val << "\n";
}

```

Вот вывод программы:

```

val в main(): 10
val в f1(): 88
val в main(): 10

```

Переменная, названная **val**, объявлена дважды, один раз в **main()** и другой раз в **f1()**. **val** в **main()** никак не влияет на **val** в **f1()**, и вообще не имеет к ней никакого отношения. Причина этого в том, что каждая переменная **val** известна только в той функции, где она объявлена. Как видно из вывода программы, несмотря на то, что при вызове **f1()** объявленной в ней **val** присваивается значение 88, содержимое **val** в **main()** остается равным 10.

Из-за того, что локальная переменная создается и уничтожается при каждом входе в блок, где она объявлена, и выходе из него, локальная переменная не сохраняет свое значение между последовательными активациями ее блока. Это особенно важно применительно к вызовам функций. Когда вызывается функция, создаются ее локальные переменные. После возврата из функции все они уничтожаются. Это значит, что локальные переменные не сохраняют свои значения от одного вызова функции до другого.

Если объявление локальной переменной включает инициализатор, то эта переменная инициализируется заново при каждом входе в блок. Например:

```

/*
    Локальная переменная инициализируется каждый раз
    при входе в ее блок.
*/


#include <iostream>
using namespace std;

```

```
void f();

int main()
{
    for(int i=0; i < 3; i++) f();

    return 0;
}

// num инициализируется при каждом вызове f().
void f()
{
    int num = 99;  num присваивается значение 99 при каждом вызове f().

    cout << num << "\n";

    num++; // результат этого действия не сохраняется
}
```

Вывод программы подтверждает, что **num** инициализируется при каждом вызове **f()**:

```
99
99
99
```

Локальная переменная, которая не инициализируется при ее объявлении, будет иметь неопределенное значение до тех пор, пока ей не будет что-то присвоено.

Локальные переменные могут быть объявлены в любом блоке

Чаще всего все переменные, которые будут нужны в некоторой функции, объявляются в начале ее функционального блока. Так поступают главным образом для того, чтобы тот, кто будет читать этот код, мог легко определить, какие переменные будут в нем использоваться. Однако начало функционального блока не является единственным местом, где можно объявлять локальные переменные. Локальную переменную можно объявить где угодно, в любом блоке кода. Переменная, объявленная внутри блока, локальна по отношению к этому блоку. Это означает, что переменная не существует, пока не начал выполняться этот блок, и уничтожается вместе с завершением блока.

Никакие предложения вне этого блока – включая остальные строки той же функции – не могут обратиться к этой переменной. Для того, чтобы усвоить это ограничение, рассмотрим следующую программу:

```
// Переменные могут быть локальными по отношению к блоку.

#include <iostream>
using namespace std;

int main() {
    int x = 19; // x известна всюду.
    if(x == 19) {
        int y = 20; ←————— y локальна по отношению к блоку if.

        cout << "x + y равно " << x + y << "\n";
    }

    // y = 100; // Ошибка! y здесь неизвестна.

    return 0;
}
```

Переменная `x` объявлена в начале области видимости `main()` и доступна всему последующему коду, входящему в `main()`. Внутри блока `if` объявляется переменная `y`. Поскольку блок определяет область видимости, `y` видима только для остального кода внутри своего блока. Поэтому вне этого блока предложение

```
y = 100;
```

закомментировано. Если вы удалите ведущие символы комментария, при компиляции будет зафиксирована ошибка, поскольку `y` не видна вне ее блока. Внутри блока `if` можно использовать `x`, потому что код внутри блока имеет доступ к переменным, объявленным во внешнем блоке, содержащем в себе данный.

Хотя локальные переменные обычно объявляются в начале их блока, это не обязательно. Локальная переменная может быть объявлена в любом месте внутри блока, но, разумеется, перед тем, как она будет использоваться. Например, следующая программа вполне правильна:

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

int main()
{
    cout << "Введите число: ";
```

```

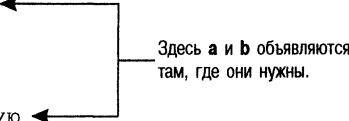
int a; // объявим одну переменную
cin >> a;

cout << "Введите второе число: ";
int b; // объявим другую переменную
cin >> b;

cout << "Произведение: " << a*b << '\n';

return 0;
}

```



Здесь **a** и **b** объявляются там, где они нужны.

В этом примере **a** и **b** не объявляются до тех пор, пока в них не возникает необходимость. Честно говоря, большинство программистов объявляют локальные переменные в начале использующей их функции, но это вопрос вкуса.

Соккрытие имен

Если локальная переменная, объявленная во внутреннем блоке, имеет то же имя, что и переменная, объявленная во внешнем блоке, то переменная, объявленная во внутреннем блоке, *скрывает* переменную из внешнего блока. Например:

```

#include <iostream>
using namespace std;

int main()
{
    int i;
    int j;


    i = 10;
    j = 100;

    if(j > 0) {
        int i; // эта i отличается от внешней i
        i = j / 2;
        cout << "внутренняя i: " << i << '\n';
    }

    cout << "внешняя i: " << i << '\n';

    return 0;
}

```



Эта **i** скрывает внешнюю **i**.

Вывод этой программы:

```
внутренняя i: 50  
внешняя i: 10
```

Переменная *i*, объявленная внутри блока *if*, скрывает внешнюю переменную *i*. Изменение значения внутренней *i* не оказывает влияния на внешнюю *i*. Более того, вне блока *if* внутренняя *i* неизвестна, и снова становится видимой внешняя *i*.

Параметры функции

Параметры функции находятся в области видимости функции. Таким образом, они локальны по отношению к функции. За исключением того, что они приобретают значения аргументов, параметры ведут себя точно так же, как и другие локальные переменные.

Цель

5.7. Глобальная область видимости

Поскольку локальные переменные известны только внутри той функции, где они объявлены, у вас может возникнуть вопрос: каким образом создать переменную, которая будет известна нескольким функциям? Ответом является объявление переменной в глобальной области видимости. *Глобальной областью видимости* является декларативный район, находящийся вне всех функций. Объявление переменной в глобальной области видимости создает *глобальную переменную*.

Спросим у эксперта

Вопрос: Что обозначает ключевое слово **auto**? Я слышал, что оно используется для объявления локальных переменных. Так ли это?

Ответ: Язык C++ содержит ключевое слово **auto**, которое можно использовать для объявления локальных переменных. Однако, поскольку все локальные переменные по умолчанию предполагаются принадлежащими классу **auto**, это слово фактически никогда не используется. Поэтому и в примерах этой книги вы его нигде не увидите. Если, однако, вы решите его использовать, располагайте его непосредственно перед типом переменной, как это показано здесь:

```
auto char ch;
```

Еще раз подчеркиваем, что слово **auto** не обязательно, и в этой книге оно не используется.

Глобальные переменные известны на протяжении всей программы. Их может использовать любой участок кода, и они сохраняют свои значения в течение всего времени выполнения программы. Таким образом, их область видимости простирается на всю программу. Вы можете создать глобальные переменные, объявив их вне всех функций. Поскольку эти переменные глобальны, к ним можно обращаться в любых выражениях, независимо от того, какие функции содержат эти выражения.

Приведенная ниже программа демонстрирует использование глобальной переменной. Переменная **count** объявлена вне всех функций. Ее объявление расположено перед функцией **main()**. Однако, его можно поместить в любом месте, но только не внутри функции. Поскольку, однако, вы должны объявлять переменную перед тем, как будете ее использовать, лучше всего располагать объявление глобальных переменных в самом начале программы.

```
// Использование глобальной переменной.
```

```
#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; ← Это обращение к глобальной count.
        func1();
    }
```

```
    return 0;
}
```

```
void func1()
{
    cout << "count: " << count; // Обращение к глобальной
                                // переменной
    ← Это также обращение к глобальной count.
```

```
cout << '\n'; // новая строка
func2();
}

void func2()
{
    int count; // это локальная переменная

    for(count=0; count<3; count++) cout << '.';
}
```

Эта **count** локальна в **func2()**.

Это обращение к локальной **count**.

Вывод программы выглядит так:

```
count: 0
...count: 2
...count: 4
...count: 6
...count: 8
...count: 10
...count: 12
...count: 14
...count: 16
...count: 18
...
```

Внимательно посмотрев на программу, легко заметить, что и **main()**, и **func1()** используют одну и ту же глобальную переменную **count**. В **func2()**, однако, объявлена локальная переменная **count**. Функция **func2()**, используя **count**, обращается к своей локальной переменной, а не к глобальной. Важно усвоить, что если глобальная и локальная переменные имеют одно и то же имя, все обращения к имени переменной внутри функции, в которой объявлена локальная переменная, будут относиться к локальной переменной, и никак не влиять на глобальную. Таким образом, локальная переменная скрывает глобальную с тем же именем.

Глобальные переменные инициализируются при запуске программы. Если объявление глобальной переменной имеет инициализатор, тогда переменная принимает значение этого инициализатора. Если глобальная переменная не имеет инициализатора, ее значение делается равным нулю.

Место под глобальные переменные выделяется в фиксированном районе памяти, выделенном вашей программой специально для этой цели. Глобальные переменные удобны в тех случаях, когда одно и то же данное используется несколькими функциями вашей программы, или когда переменная должна сохранять свое значение в течение всего времени выполнения программы. Однако следует избегать чрез-

мерного увлечения глобальными переменными. К этому имеются три причины:

- Глобальные переменные занимают память в течение всего времени выполнения программы, а не только когда они действительно нужны.
- Использование глобальной переменной в том месте, где годится локальная, снижает общность функции, так как она теперь зависит от объекта, который должен быть определен вне нее.
- Использование большого числа глобальных переменных может привести к программным ошибкам из-за неизвестных или нежелательных побочных эффектов. Основная проблема при разработке больших программ заключается в возможности случайной модификации значения переменной в силу ее использования в каком-то другом месте программы. Такое может случиться в C++, если в вашей программе слишком много глобальных переменных.

Минутная тренировка

1. В чем основные различия между локальными и глобальными переменными?
 2. Допустимо ли объявлять локальную переменную где угодно внутри блока?
 3. Сохраняет ли локальная переменная свое значение при повторных вызовах функции, в которой она объявлена?
1. Локальная переменная известна только внутри блока, в котором она объявлена. Она создается при входе в этот блок и уничтожается, когда блок завершает свое выполнение. Глобальная переменная объявляется вне всех функций. Она может использоваться всеми функциями и существует в течение всего времени выполнения программы.
 2. Да, локальная переменная может быть объявлена где угодно внутри блока, однако до того места, где она будет использоваться.
 3. Нет, локальные переменные уничтожаются при возврате из функции, в которой они объявлены.
-

Цель

5.8. Передача в функции указателей и массивов

В предыдущих примерах в качестве аргументов использовались простые значения типов `int` или `double`. Однако часто в качестве аргументов приходится использовать указатели и массивы. Хотя передача в функцию таких аргументов осуществляется так же, как и любых других, однако при их использовании возникают проблемы, которые следует рассмотреть особо.

Передача указателя

Для того, чтобы передать указатель в качестве аргумента, вы должны объявить параметр типа указателя. Вот пример:

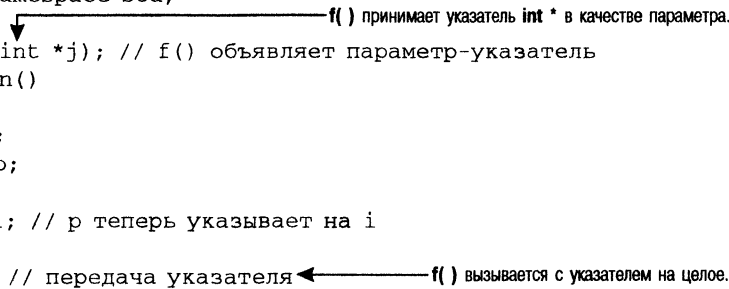
// Передача функции указателя.

```
#include <iostream>
using namespace std;
void f(int *j); // f() объявляет параметр-указатель
int main()
{
    int i;
    int *p;

    p = &i; // p теперь указывает на i

    f(p); // передача указателя
    cout << i; // i теперь равно 100

    return 0;
}
// f() получает указатель на int.
void f(int *j)
{
    *j = 100; // переменной, на которую указывает j,
              // теперь присвоено значение 100
}
```



Рассмотрим внимательно эту программу. Мы видим, что функция `f()` принимает один параметр: указатель на `int`. Внутри `main()` указателю `p` присваивается значение адреса переменной `i`. Далее вызываете `f()` с `p` в качестве аргумента. Когда параметр-указатель `j` получает значение `p`, он так же указывает на `i` внутри `main()`. Таким образом, предложение

```
*j = 100;
```

присваивает значение 100 переменной `i`. В общем случае `f()` запишет значение 100 по любому адресу, который будет ей передан при вызове.

В предыдущем примере фактически не было необходимости использовать переменную-указатель `p`. Достаточно при вызове `f()` указать в качестве аргумента `i`, предварив это имя значком `&`. В результате

в функцию `f()` будет передан адрес `i`. Модифицированная программа имеет такой вид:

```
// Передача функции указателя.

#include <iostream>
using namespace std;

void f(int *j);

int main()
{
    int i;

    f(&i); ← Нет необходимости в p. В f( ) непосредственно
              передается адрес i.

    cout << i;

    return 0;
}

void f(int *j)
{
    *j = 100; // переменной, на которую указывает j,
              // теперь присвоено значение 100
}
```

Существенно, чтобы вы усвоили важное обстоятельство, связанное с передачей в функции указателей: когда вы выполняете внутри функции операцию, использующую указатель, вы фактически работаете с переменной, на которую этот указатель указывает. Это дает возможность функции изменять значение объекта, на который указывает аргумент-указатель.

Передача массива

Когда в качестве аргумента функции выступает массив, в функцию передается не копия всего массива, а адрес его первого элемента. (Вспомним, что имя массива без индекса является указателем на первый элемент массива.) Соответственно, объявление параметра в функции должно быть совместимым с этим адресом по типу. Имеются три способа объявить параметр, который должен будет принять указатель на массив. Первый способ заключается в том, что этот параметр объявляется как массив того же типа и размера, что и массив, передаваемый в функцию при ее вызове. Этот способ использован в приведенном ниже 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); // передача в функцию массива t

    return 0;
}

// Выведем несколько чисел.
void display(int num[10]) ← Параметр объявлен как массив определенной длины.
{
    int i;

    for(i=0; i < 10; i++) cout << num[i] << ' ' ;
}
```

Несмотря на то, что параметр **num** объявлен как целочисленный массив из 10 элементов, компилятор C++ автоматически преобразует его в указатель на **int**. Это преобразование необходимо, потому что никакой параметр не может принять целый массив. Поскольку в функцию будет передан указатель на массив, в функции должен быть такой же параметр-указатель для его получения.

Второй способ объявить параметр-указатель состоит в задании параметра в виде массива неопределенной длины, как это показано ниже:

```
void display(int num[]) ← Параметр объявлен как массив неопределенной длины.
{
    int i;

    for(i=0; i < 10; i++) cout << num[i] << ' ' ;
}
```

Здесь параметр **num** объявлен как массив неопределенной длины. Поскольку C++ не выполняет проверку на выход за границы массива, фактический размер массива не существенен для параметра (но, разумеется, не для программы). При таком способе объявления компилятор так же автоматически преобразует параметр в указатель на **int**.

Наконец, `num` можно объявить как указатель. Этот метод чаще других используется в профессионально написанных программах. Вот пример такого объявления:

```
void display(int *num) ←———— Параметр объявлен как указатель.
{
    int i;
    for(i=0; i < 10; i++) cout << num[i] << ' ';
}
```

Возможность объявления параметра как указателя проистекает из того, что любой указатель можно индексировать как массив, используя квадратные скобки `[]`. Заметьте себе, что все три метода объявления массива в качестве параметра дают один и тот же результат: указатель.

Важно понимать, что когда массив используется в качестве аргумента функции, в функцию передается его адрес. Это означает, что код внутри функции будет воздействовать и, возможно, изменять фактическое содержимое этого массива. В качестве примера рассмотрим входящую в приведенную ниже программу функцию `cube()`, которая преобразует значение каждого элемента массива в его куб. При вызове функции `cube()` ей в качестве первого аргумента передается адрес массива, а в качестве второго — его размер:

// Измерение содержимого массива с помощью функции.

```
#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: ";
```

```
    for(i=0; i < 10; i++) cout << nums[i] << ' ';
    cout << '\n';
```

```
    cube(nums, 10); // вычисление кубов
```

←———— Передадим адрес `nums`
функции `cube()`.

```
    cout << "Измененное содержимое: ";
```

```
    for(i=0; i<10; i++) cout << nums[i] << ' ';
```

```
    return 0;
```

```
}
```

```
// Возведение в куб элементов массива.
void cube(int *n, int num)
{
    while(num) {
        *n = *n * *n * *n; ← Здесь изменяется значение элемента
        num--;               массива, на который указывает n.
        n++;
    }
}
```

Ниже приведен вывод этой программы:

Исходное содержимое: 1 2 3 4 5 6 7 8 9 10

Измененное содержимое: 1 8 27 64 125 216 343 512 729 1000

Мы видим, что после вызова функции **cube()** массив **nums** в **main()** содержит кубы первоначальных значений его элементов. Таким образом, значения элементов **nums** были модифицированы программными предложениями внутри функции **cube()**. Это стало возможным потому, что параметр функции **n** указывает на **nums**.

Передача строк

Строка представляет собой обычный массив символов с нулем в конце, и когда вы передаете строку в функцию, фактически передается только указатель на начало строки. Этот указатель имеет тип **char ***. Рассмотрим в качестве примера следующую программу. В ней определена функция **strInvertCase()**, которая преобразует строчные буквы строки в прописные и наоборот:

```
// Передача в функцию строки.

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

void strInvertCase(char *str);

int main()
{
    char str[80];

    strcpy(str, "This Is A test");

    strInvertCase(str);
}
```

```
cout << str; // вывод модифицированной строки

return 0;
}

// Преобразование регистра букв внутри строки.
void strInvertCase(char *str)
{
    while(*str) {

        // преобразуем регистр
        if(isupper(*str)) *str = tolower(*str);
        else if(islower(*str)) *str = toupper(*str);

        str++; // сместимся к следующему символу
    }
}
```

Вот вывод этой программы:

```
tHIS iS a tEST
```

Минутная тренировка

1. Покажите, как объявляется **void**-функция **count**, имеющая единственный параметр-указатель на **long int**, названный **ptr**.
2. Если в функцию передается указатель, может ли функция изменить содержимое объекта, на который указывает этот указатель?
3. Можно ли передать функции массив? Поясните ваш ответ.

1. `void count(long int *ptr);`
 2. Да, объект, на который указывает аргумент-указатель, может быть изменен кодом внутри функции.
 3. Массивы нельзя передавать в функцию. Однако указатель на массив передать можно. При этом изменения, вносимые в массив внутри функции, отображаются в массиве, используемом как аргумент функции.
-

Цель

5.9. Возврат указателей

Функции могут возвращать указатели. Указатели возвращаются, как и данные любых других типов, и не вызывают никаких особых проблем. Если, однако, учесть, что указатели являются одним из наиболее сложных средств C++, краткое обсуждение этой темы вполне уместно.

В приводимой ниже программе демонстрируется использование указателя в качестве возвращаемого значения. Функция `get_substr()` просматривает строку в поисках заданной подстроки. Функция возвращает указатель на первую найденную подстроку. Если заданной подстроки найти не удалось, возвращается нулевой указатель. Например, если анализируется строка “Я люблю С++”, а в качестве искомой задана подстрока “люблю”, то функция вернет указатель на первую букву *л* в слове “люблю”.

```
// Возврат указателя.

#include <iostream>
using namespace std;

char *get_substr(char *sub, char *str);

int main()
{
    char *substr;

    substr = get_substr("три", "один два три четыре");

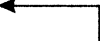
    cout << "подстрока найдена: " << substr;

    return 0;
}

// Возвращает указатель на подстроку или нулевой указатель,
// если подстрока не найдена.
char *get_substr(char *sub, char *str)
{
    int t;
    char *p, *p2, *start;

    for(t=0; str[t]; t++) {
        p = &str[t]; // начальная установка указателей
        start = p;
        p2 = sub;
        while(*p2 && *p2==*p) { // проверка на наличие подстроки
            p++;
            p2++;
        }

        /* Если p2 указывает на завершающий 0 (т. е. на
           конец подстроки), значит, вся подстрока найдена. */
        if(!*p2)
```



get_substr() возвращает указатель на char.


```
        return start; // вернуть указатель на начало подстроки
    }

    return 0; // подстрока не найдена
}
```

Вот вывод этой программы:

подстрока найдена: три четыре

Функция `main()`

Как вы уже знаете, функция `main()` выделяется среди всех остальных, поскольку это первая функция, которая вызывается, когда вы запускаете свою программу. Она обозначает начало программы. В отличие от некоторых других языков программирования, которые всегда начинают выполнение с самого “верха” программы, C++ начинает любую программу с вызова функции `main()`, независимо от того, в каком месте исходного текста программы она расположена. (Хотя, как правило, функцию `main()` помещают в самое начало программы, чтобы было легче ее найти.)

В программе может быть только одна функция `main()`. Если вы попытаетесь включить вторую, программа не будет знать, где начать выполнение. Большинство компиляторов обнаружат эту ошибку и сообщат о ней. Как уже отмечалось, функция `main()` предопределена в C++ и не нуждается в прототипе.

Цель

5.10. `argc` и `argv`: аргументы функции `main()`

Иногда при запуске программы требуется передать в нее некоторую информацию. Обычно это делается посредством передачи в `main()` *аргументов командной строки*. Аргументами командной строки называются данные, которые вы вводите вслед за именем программы на командной строке операционной системы. (В системе Windows команда Пуск также использует командную строку.) Например, когда вы компилируете C++-программу, вы вводите на командной строке что-то вроде этого:

```
cl имя-программы
```

Здесь *имя-программы* обозначает программу, которую вы хотите откомпилировать. Имя программы передается компилятору C++ как аргумент командной строки.

В C++ определены два встроенных (но необязательных) параметра функции `main()`. Они называются `argc` и `argv` и служат для приема аргументов командной строки. Функция `main()` допускает только два параметра. Однако ваша конкретная операционная среда может поддерживать и другие параметры, так что вам стоит свериться с документацией к используемому вами компилятору. Теперь рассмотрим `argc` и `argv` более детально.



Замечание

Формально имена параметров командной строки могут быть какими угодно – вы можете назвать их как вам заблагорассудится. Однако в течение ряда лет для их обозначения использовались именно эти имена – `argc` и `argv`, и мы рекомендуем вам не отходить от этой практики, чтобы любой, читающий вашу программу, мог идентифицировать их как параметры командной строки.

Параметр `argc` является целым числом, обозначающим число аргументов, введенных на командной строке. Это число будет по меньшей мере равно 1, так как имя программы рассматривается как первый аргумент.

Параметр `argv` является указателем на массив символьных указателей. Каждый указатель в массиве `argv` указывает на строку, содержащую аргумент командной строки. На имя программы указывает `argv[0]`; `argv[1]` будет указывать на первый аргумент, `argv[2]` на второй и т. д. Все аргументы командной строки передаются в программу как символьные строки, поэтому числовые аргументы должны быть преобразованы вашей программой в соответствующий числовой формат.

Важно правильно объявить параметр `argv`. Чаше всего используют такое обозначение:

```
char *arg[];
```

Для обращения к отдельным аргументам вы можете индексировать `argv`. В приведенной ниже программе демонстрируется обращение к аргументам командной строки. Программа выводит на экран все аргументы командной строки, которые вы вводите с клавиатуры при ее запуске.

```
// Вывод на экран аргументов командной строки.
```

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

```
int main(int argc, char *argv[])
{
    for(int i = 0; i < argc; i++) {
        cout << argv[i] << "\n";
    }

    return 0;
}
```

Предположим, что программа имеет имя ComLine. Тогда, если запустить ее таким образом:

```
C>ComLine один два три
```

программа выведет на экран следующее:

```
ComLine
один
два
три
```

C++ не оговаривает точный формат аргументов командной строки, потому что операционные системы, управляющие работой компьютеров, в этом отношении могут значительно различаться. Однако чаще всего при вводе аргументов командной строки их разделяют пробелами или символами табуляции. При этом обычно запятые или символы точки с запятой не могут использоваться в качестве разделителей аргументов. Например, комбинация аргументов

```
один, два, и три
```

образует четыре строки, в то время как в комбинации

```
один,два,и три
```

только две строки — запятая не является допустимым разделителем.

Если вам требуется передать программе аргумент командной строки, который содержит пробелы, вы должны заключить его в кавычки. Например, следующая фраза будет рассматриваться как один аргумент командной строки:

```
"это все есть один аргумент"
```

Имейте в виду, что приводимые здесь примеры годятся для широкого диапазона операционных сред, но все же не для всех.

Обычно вы используете **argc** и **argv** для ввода в программу начальных режимов или значений (например, имен файлов). В C++ вы можете вводить столько аргументов командной строки, сколько допускает ваша операционная система. Использование аргументов командной строки придаст вашей программе профессиональный вид и позволит запускать ее из командных файлов.

Передача числовых аргументов командной строки

Если вы передаете в программу через аргументы командной строки числовые данные, они будут приняты программой в форме символьных строк. Ваша программа должна в этом случае преобразовать их во внутренний двоичный формат с помощью одной из функций стандартной библиотеки, поддерживаемой C++. Три наиболее часто используемые для этого функции приведены ниже:

| | |
|----------------|---|
| atof() | Преобразует строку в значение double и возвращает результат. |
| atol() | Преобразует строку в значение long int и возвращает результат. |
| atoi() | Преобразует строку в значение int и возвращает результат. |

Все эти функции принимают в качестве аргумента строку, содержащую числовое значение; все они требуют наличия заголовка **<cstdlib>**.

В приведенной ниже программе демонстрируется преобразование числового аргумента командной строки в его двоичный эквивалент. Программа вычисляет сумму двух чисел, вводимых на командной строке вслед за именем программы. Для преобразования числового аргумента в его внутреннее представление используется функция **atof()**.

```
/*
   Эта программа выводит 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]); // преобразуем первый аргумент командной  
    // строки  
b = atof(argv[2]); // преобразуем второй аргумент командной  
    // строки  
  
cout << a + b;  
  
return 0;  
}
```

Используйте **atof()** для преобразования числового аргумента командной строки в тип **double**.

Для сложения двух чисел используйте следующую форму командной строки (в предположении, что программа имеет имя *add*):

C:>add 100.2 231

Минутная тренировка

1. Как обычно называют два параметра функции **main()**? Объясните, что содержит каждый из них.
 2. Чему всегда соответствует первый аргумент командной строки?
 3. Числовой аргумент командной строки поступает в программу в виде строки. Правильно ли это?
 1. Два параметра функции **main()** обычно называются **argc** и **argv**. Параметр **argc** задает число наличных аргументов командой строки, а **argv** является указателем на массив строк, содержащих сами аргументы.
 2. Первым аргументом командной строки всегда является имя данной программы.
 3. Правильно.
-

Цель

5.11. Прототипы функций

В начале этого модуля был кратко затронут вопрос о прототипах функций. Теперь наступил момент, когда мы должны рассмотреть эту тему более подробно. В C++ все функции перед тем, как их использовать, должны быть объявлены. Объявление функции выполняется с помощью ее прототипа. В прототипе задаются три характеристики функции:

- Тип возвращаемого значения.
- Типы параметров функции.
- Число параметров функции.

Прототипы позволяют компилятору выполнить три важных операции:

- Они говорят компилятору, с помощью какого рода кода должна вызываться функция. Различные типы возвращаемых значений должны по-разному обрабатываться компилятором.
- Они позволяют компилятору выявить недопустимые преобразования типов аргументов, используемых при вызове функции, в объявленные типы параметров функции. При обнаружении несоответствий типов компилятор выдает сообщение об ошибке.
- Они позволяют компилятору обнаружить несоответствие числа аргументов, используемых при вызове функции, и числа ее параметров.

Общая форма прототипа функции показан ниже. Она совпадает с определением функции за исключением отсутствия тела функции:

```
тип имя-функции(тип имя-параметра1, тип имя-параметра2, ..., тип имя-параметраN);
```

Использование имен параметров необязательно. Однако их использование позволяет компилятору при обнаружении расхождений в типах аргументов и параметров включить в сообщение об ошибке имя спорного параметра, что облегчает отладку. Поэтому разумно включать в прототипы имена параметров.

Для того, чтобы продемонстрировать полезность прототипов функций, рассмотрим следующую программу. Если вы попытаетесь ее оттранслировать, будет выдано сообщение об ошибке, потому что программа пытается вызвать функцию `sqr_it()` с целочисленным аргументом вместо требуемого указателя на целое. (Компилятор не выполняет автоматическое преобразование целого числа в указатель.)

```
/*  
  Эта программа использует прототип функции для  
  активизации строгой проверки типов.  
*/
```

```
void sqr_it(int *i); // прототип
```

```
int main()
```

```
{
```

```
  int x;
```

```
  x = 10;
```

```
  sqr_it(x); // Ошибка! Несоответствие типов!
```

```
  return 0;
```

```
}
```

Прототип предотвращает несоответствие типов аргументов и параметров. **sqr_it()** ожидает получить указатель, а вызывается с аргументом типа **int**.



```
void sqr_it(int *i)
{
    *i = *i * *i;
}
```

Определение функции может одновременно служить ее прототипом, если оно расположено в программе до первого вызова этой функции. Например, это вполне правильная программа:

```
// Использование определения функции в качестве ее прототипа.

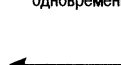
#include <iostream>
using namespace std;

// Определим, является ли число четным.
bool isEven(int num) {
    if(! (num %2)) return true; // num четно
    return false;
}

int main()
{
    if(isEven(4)) cout << "4 четно\n";
    if(isEven(3)) cout << "это не будет выведено";

    return 0;
}
```

Поскольку функция **isEven()** определена перед ее использованием, ее определение служит одновременно и ее прототипом.



В этой программе функция **isEven()** определена перед ее использованием в **main()**. В этом случае определение функции может служить и ее прототипом, и в отдельном прототипе нет необходимости.

В действительности обычно оказывается проще и лучше определить прототипы всех функций, используемых программой, чем следить за тем, чтобы определения всех функций располагались в программе до их вызовов. Это особенно справедливо для больших программ, в которых не всегда легко проследить, из каких функций какие другие функции вызываются. Более того, вполне возможно иметь две функции, которые вызывают друг друга. В этом случае прототипы функций необходимы.

Заголовки содержат прототипы

В самом начале этой книги вы столкнулись со стандартными заголовками C++. Вы узнали, что эти заголовки содержат информацию, необходимую для работы программ. Хотя это неполное объяснение и

справедливо, в нем сказано далеко не все. Заголовки C++ содержат прототипы функций стандартной библиотеки. (В них также можно найти различные значения и определения, используемые этими функциями.) Как и функции, которые вы пишете сами, функции стандартной библиотеки должны иметь прототипы, и эти прототипы следует включить в программу до вызова самих функций. По этой причине любая программа, использующая библиотечную функцию, должна также подключать к программе заголовок, содержащий прототип этой функции.

Чтобы выяснить, какой заголовок требует та или иная библиотечная функция, изучите документацию библиотеки вашего компилятора. Вместе с описанием каждой функции вы найдете название заголовка, который следует включить в программу для использования этой функции.

Минутная тренировка

1. Что такое прототип функции? В чем его назначение?
 2. Должны ли все функции, кроме `main()`, иметь свои прототипы?
 3. Почему вы должны подключать заголовок при использовании в программе функции из стандартной библиотеки?
 1. Прототип объявляет имя функции, тип возвращаемого значения и типы параметров. Прототип указывает компилятору, какой код он должен создать для вызова функции, а также обеспечивает правильность вызова.
 2. Да, все функции, за исключением `main()`, должны иметь свои прототипы.
 3. Помимо другой информации, заголовок включает прототипы функций стандартной библиотеки.
-

Цель

5.12. Рекурсия

Последняя тема, которую мы рассмотрим в этом модуле, посвящена *рекурсии*. Рекурсия, иногда называемая *циклическим определением*, представляет собой процесс определения чего-то с помощью самого себя. В приложении к программированию рекурсия — это процесс вызова функцией самой себя. Функцию, которая вызывает саму себя, называют *рекурсивной*.

Классическим примером рекурсии может служить функция `factr()`, которая вычисляет факториал целого числа. Факториалом числа N называют произведение всех целых чисел между 1 и N . Например, факториал числа 3 (или, как говорят, “три факториал”) вычисляется как $1 \times 2 \times 3$, что равно 6. Ниже приведены как функция `factr()`, так и ее итеративный эквивалент.


```
// Демонстрация рекурсии.

#include <iostream>
using namespace std;

int factr(int n);
int fact(int n);

int main()
{
    // используем рекурсивный вариант
    cout << "4 факториал равен " << factr(4);
    cout << '\n';
    // используем итеративный вариант

    cout << "4 факториал равен " << fact(4);
    cout << '\n';

    return 0;
}

// Рекурсивный вариант.
int factr(int n)
{
    int answer;

    if(n==1) return(1);
    answer = factr(n-1)*n; ←————— Выполнить рекурсивный вызов factr( ).
    return(answer);
}

// Итеративный вариант.
int fact(int n)
{
    int t, answer;

    answer = 1;
    for(t=1; t<=n; t++) answer = answer*(t);
    return(answer);
}
```

Действие нерекурсивного варианта **fact()** вполне очевидно. Он использует цикл, начинающийся с 1, в котором каждое следующее число умножается на последовательно увеличивающееся произведение.

Действие рекурсивной функции **factr()** несколько сложнее. При вызове **factr()** с аргументом 1 функция возвращает 1; в противном случае она возвращает произведение **factr(n-1)*n**. Для вычисления этого выражения **factr()** вызывается с аргументом **(n-1)**. Это происходит до тех пор, пока **n** не станет равным 1, и вызов функции приведет к возврату из нее. Например, когда вычисляется факториал 2, первый вызов **factr()** породит второй вызов с аргументом 1. Этот вызов вернет 1, которая умножится на 2 (первоначальное значение **n**). Переменная **answer** будет тогда равна 2. Вы можете ради любопытства включить в **factr()** предложения **cout**, которые покажут, на каком уровне находится вызов и каков промежуточный результат (переменная **answer**).

Когда функция вызывает саму себя, для новых локальных переменных и параметров выделяется память (обычно на системном стеке), и код функции выполняется с самого начала с использованием этих новых переменных. Рекурсивный вызов не создает новой копии функции; только ее аргументы будут новыми. При возврате из каждого рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение продолжается с точки вызова функции изнутри функции. Можно сказать, что рекурсивные функции действуют подобно раздвижной подзорной трубе, сначала раскладываясь, а затем складываясь.

Имейте в виду, что большинство рекурсивных функций не уменьшают размер кода в заметной степени. Кроме того, рекурсивные варианты большинства алгоритмов выполняются несколько более медленно, чем их итеративные эквиваленты, из-за добавления издержек на дополнительные функциональные вызовы. Слишком много рекурсивных обращений к функции может вызвать переполнение стека. Из-за того, что параметры и локальные переменные функции размещаются на стеке, и каждый новый вызов создает новую копию этих переменных, стек может истощиться. Если это произойдет, то могут быть разрушены другие данные. Реально, однако, вряд ли вам придется столкнуться с такой ситуацией, если только рекурсивная функция не заиклится.

Основное преимущество рекурсивных функций заключается в том, что их можно использовать для создания вариантов некоторых алгоритмов, которые оказываются проще и нагляднее, чем их итеративные аналоги. Например, алгоритм быстрого упорядочения довольно затруднительно реализовать в виде итеративных вызовов. Некоторые задачи, особенно в области искусственного интеллекта, кажутся прямо приспособленными для рекурсивного решения.

Составляя рекурсивную функцию, вы должны где-то в ней включить условное предложение, например, **if**, чтобы заставить функцию осуществить возврат без выполнения рекурсивного вызова. Если вы пренебрежете таким условным предложением, то вы-

звав однажды эту функцию, вы никогда из нее не вернетесь. Это довольно распространенная ошибка. Разрабатывая программы с рекурсивными функциями, не стесняйтесь включать в них предложения `cout`, которые позволят вам наблюдать, что происходит в программе, и аварийно завершить выполнение, если вы обнаруживаете ошибку.

Ниже приведен другой пример рекурсивной функции, названной `reverse()`. Она выводит на экран строку задом наперед.

```
// Вывод строки задом наперед посредством рекурсии.

#include <iostream>
using namespace std;

void reverse(char *s);

int main()
{
    char str[] = "это просто проверка";

    reverse(str);

    return 0;
}

// Вывод строки задом наперед.
void reverse(char *s)
{
    if(*s)
        reverse(s+1);
    else
        return;

    cout << *s;
}
```

Функция `reverse()` прежде всего проверяет, не передан ли ей указатель на завершающий ноль строки. Если нет, то `reverse()` вызывает себя с передачей в качестве аргумента указателя на следующий символ строки. Когда наконец будет найден завершающий ноль, вызовы начинают “раскручиваться”, и символы появляются на экране в обратном порядке.

Еще одно замечание. Понятие рекурсии часто оказывается сложным для начинающих. Не отчаивайтесь, если этот материал показался вам слишком запутанным. С течением времени вы освоите и его.

Проект 5-1 Быстрое упорядочение

В Модуле 4 вы познакомились с простым методом упорядочения, носящим название пузырькового. Там же мы упомянули, что имеются существенно лучшие алгоритмы упорядочения. В этом проекте вы реализуете один из лучших вариантов: Quicksort (быстрое упорядочение). Метод Quicksort, изобретенный Хором (С.А.Р. Hoare), который и дал ему это имя, является лучшим алгоритмом упорядочения общего назначения из имеющихся в настоящее время. Причина, по которой он не мог быть продемонстрирован в Модуле 4, заключается в том, что реализация Quicksort опирается на рекурсию. Вариант, который мы рассмотрим, будет упорядочивать символьный массив, хотя логику алгоритма можно приспособить для упорядочения объектов любого типа.

Quicksort построен на идее разделов. Общая процедура заключается в том, что выбирается значение, называемое *компарандом*, а затем массив разбивается на две секции. Все элементы, большие или равные компаранда, перемещаются на одну сторону, а все меньшие — на другую. Этот процесс затем повторяется для каждой оставшейся секции, пока весь массив не окажется упорядоченным. Пусть, например, дан массив **fedacb**, и значение **d** используется в качестве компаранда. Первый проход Quicksort изменит последовательность символов в массиве следующим образом:

| | |
|--------------------|-------------|
| Исходное состояние | f e d a c b |
| Проход 1 | b c a d e f |

Этот процесс повторяется затем для каждой секции — т. е. **bca** и **def**. Легко видеть, что процесс по самой своей природе существенно рекурсивен, и, очевидно, лучше всего реализовать Quicksort в виде рекурсивной функции.

Найти значение, которое будет выступать в качестве компаранда, можно двумя способами. Вы можете выбрать компаранд случайным образом, или можно получить его, усреднив небольшой набор значений, взятых из массива. Для получения оптимального упорядочения вы должны выбрать такое значение, которое будет точно в середине диапазона данных, составляющих упорядочиваемый массив. Однако, для большинства коллекций данных это сделать непросто. В худшем случае выбранное значение окажется с одного края массива. Даже в этом случае Quicksort работает правильно. Вариант Quicksort, который мы реализуем, выбирает в качестве компаранда средний элемент массива.

Еще одно замечание: библиотека C++ включает функцию с именем **qsort()**, которая также реализует алгоритм Quicksort. Вам будет интересно сравнить ее с вариантом, приведенным здесь.

Шаг за шагом

1. Создайте файл с именем **QSDemo.cpp**.
2. Алгоритм Quicksort будет реализован с помощью двух функций. Первая, с именем **quicksort()**, предоставляет удобный интерфейс для пользователя и организует вызов **qs()**, которая осуществляет фактическое упорядочение. Прежде всего, создайте функцию **quicksort()**, как это показано ниже:

```
// Организуем вызов функции упорядочения.
void quicksort(char *items, int len)
{
    qs(items, 0, len-1);
}
```

Здесь **items** указывает на упорядочиваемый массив, а **len** задает число элементов в этом массиве. Как показано на следующем шаге, **qs()** требует определения начального раздела, что и осуществляет **quicksort()**. Преимущество в использовании **quicksort()** заключается в том, что ее можно вызывать с указанием только двух очевидных аргументов: указателя на упорядочиваемый массив и числа элементов в массиве. Она затем предоставляет начальный и конечный индексы для упорядочиваемого раздела.

3. Добавьте собственно упорядочивающую функцию с именем **qs()**, как это показано ниже:

```
// Рекурсивный вариант алгоритма Quicksort для упорядочения
// символов.
void qs(char *items, int left, int right)
{
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[( left+right) / 2 ];

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);
}
```

```
if(left < j) qs(items, left, j);  
if(i < right) qs(items, i, right);  
}
```

Эту функцию следует вызывать с указанием индексов упорядочиваемого раздела. Левый параметр должен соответствовать началу (левой границе) раздела. Правый параметр должен соответствовать концу (правой границе) раздела. При первом вызове раздел представляет собой весь массив. Каждый рекурсивный вызов последовательно упорядочивает все меньшие и меньшие разделы.

4. Чтобы активизировать алгоритм Quicksort, просто вызовите **quicksort()** с именем упорядочиваемого массива и его длины. После возврата из функции массив будет упорядочен. Не забудьте, что этот вариант годится только для символьных массивов, но вы можете адаптировать его логику для упорядочения нужных вам типов массивов.

Вот программа, демонстрирующая алгоритм Quicksort:

```
/*  
    Проект 5-1  
    Вариант алгоритма Quicksort для упорядочения символов.  
*/  
  
#include <iostream>  
#include <cstring>  
  
using namespace std;  
  
void quicksort(char *items, int len);  
  
void qs(char *items, int left, int right);  
  
int main() {  
  
    char str[] = "jfmckldoelazlkper";  
    int i;  
  
    cout << "Исходный массив: " << str << "\n";  
  
    quicksort(str, strlen(str));  
  
    cout << "Упорядоченный массив: " << str << "\n";  
  
    return 0;  
}
```

```
// Вызовем функцию, выполняющую фактическое упорядочение.
void quicksort(char *items, int len)
{
    qs(items, 0, len-1);
}

// Рекурсивный вариант алгоритма Quicksort
// для упорядочения символов.
void qs(char *items, int left, int right)
{
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left+right) / 2 ];

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}
```

Ниже приведен вывод программы:

Исходный массив: jfmckldoelazlkper

Упорядоченный массив: acdeefjkklllmoprz

Спросим у эксперта

Вопрос: Я слышал что-то о правиле “по умолчанию int”. Что это такое и применимо ли это правило к C++?

Ответ: В исходном языке C, а также в ранних версиях C++, если в объявлении отсутствовал спецификатор типа, по умолчанию предполагался тип **int**. Например, с использованием старого стиля приведенная ниже функция будет написана правильно; она вернет результат типа **int**:

```
f() { // по умолчанию тип возврата устанавливается int
{
    int x;
    // ...
    return x;
}
```

В этом фрагменте для значения, возвращаемого функцией `f()`, по умолчанию устанавливается тип `int`, поскольку никакой другой тип возврата не указан. Однако, правило “по умолчанию `int`” (которое также называют правилом “неявного `int`”) не поддерживается современными версиями C++. Хотя большинство компиляторов продолжает поддерживать правило “по умолчанию `int`” ради обратной совместимости, вам следует явным образом специфицировать тип возврата каждой создаваемой вами функции. Поскольку старые программы нередко использовали тип возврата по умолчанию, все сказанное следует иметь в виду, если вам придется модифицировать унаследованные программы.

✓ Вопросы для самопроверки

1. Приведите общую форму определения функции.
2. Создайте функцию `hypot()`, которая вычисляет гипотенузу прямоугольного треугольника по двум его катетам. Продемонстрируйте использование этой функции в программе. Вам понадобится библиотечная функция `sqrt()`, которая возвращает квадратный корень из его аргумента. Прототип этой функции таков:

```
double sqrt(double значение);
```

Эта функция использует заголовок `<cmath>`.

3. Может ли функция вернуть указатель? Может ли функция вернуть массив?
4. Создайте собственный вариант функции `strlen()` стандартной библиотеки.
5. Сохраняет ли локальная переменная свое значение при последовательных вызовах функции, в которой она объявлена?
6. Приведите одно преимущество глобальных переменных. Приведите один их недостаток.
7. Создайте функцию `byThrees()`, которая возвращает последовательность чисел, каждое из которых на 3 больше предыдущего. Начните последовательность с 0. Тогда первые пять чисел, возвращенных функцией `byThrees()`, будут 0, 3, 6, 9 и 12. Создайте другую функцию, `reset()`, которая заставит `byThrees()` начать создавать новую

последовательность снова с 0. Продемонстрируйте использование этих функции в программе.

Подсказка: Вам понадобится глобальная переменная.

8. Напишите программу, которая требует ввод пароля, указываемого на командной строке. Ваша программа может не выполнять никакой полезной работы, а только выводить сообщение, правильный ли пароль введен.
9. Прототип предотвращает вызов функции с неправильным числом аргументов. Справедливо ли это утверждение?
10. Напишите рекурсивную функцию, которая выводит числа от 1 до 10. Продемонстрируйте ее использование в программе.

Модуль 6 Подробнее о функциях

Цели, достигаемые в этом модуле

- 6.1 Осознать два подхода к передаче аргументов
- 6.2 Понять, как C++ передает аргументы в функции
- 6.3 Узнать, как с помощью указателя создается вызов по ссылке
- 6.4 Научиться передавать ссылки в функции
- 6.5 Рассмотреть возврат ссылок
- 6.6 Освоить независимые ссылки
- 6.7 Познакомиться с понятием перегрузки функций
- 6.8 Начать использовать аргументы функции по умолчанию
- 6.9 Узнать, как избежать неоднозначности при перегрузке функций

В этом модуле мы продолжим изучение функций. Здесь будут рассмотрены три наиболее важные темы, связанные с функциями: ссылки, перегрузка функций и аргументы по умолчанию. Эти средства существенно расширяют возможности функций. Ссылка представляет собой неявный указатель. Перегрузка функций позволяет реализовывать одну и ту же функцию разными способами, каждый из которых предназначен для решения своей задачи. Перегрузка функций является одним из проявлений полиморфизма в C++. При использовании аргументов по умолчанию возникает возможность задать для параметра значение, которое будет автоматически присвоено параметру, если соответствующий аргумент не указан.

Мы начнем с рассмотрения двух способов передачи в функции аргументов и с того, какие возможности возникают при их применении. Понимание механизма передачи аргументов необходимо для освоения понятия ссылки.

Цель

6.1.

Два подхода к передаче аргументов

В принципе в компьютерном языке существуют два способа передачи подпрограмме аргумента. Первый называется *передачей по значению* (или *вызовом по значению*). Он заключается в копировании значения аргумента в параметр подпрограммы. Очевидно, что в этом случае, если подпрограмма изменит свои параметры, это никак не повлияет на аргументы, с которыми она вызывалась.

Передача по ссылке (или *вызов по ссылке*) является вторым способом передачи аргументов подпрограмме. В этом случае в параметр копируется адрес аргумента (не его значение). Внутри подпрограммы этот адрес используется для обращения к фактическому аргументу, указанному в вызове. Это значит, что изменения параметра *повлияют* на аргумент, использованный при вызове подпрограммы.

Цель

6.2.

Как C++ передает аргументы

По умолчанию C++ использует при вызове функции передачу аргументов по значению. Это значит, что код внутри функции не может изменить аргументы, указанные при ее вызове. В этой книге все про-

граммы до сих пор использовали метод передачи по значению. Рассмотрим в качестве примера функцию **reciprocal()** в приведенной ниже программе:

```
/* Изменение переданного по значению параметра
   не изменяет аргумент. */

#include <iostream>
using namespace std;

double reciprocal(double x);

int main()
{
    double t = 10.0;

    cout << "Обратное значение от 10.0 составляет " <<
        reciprocal(t) << "\n";

    cout << "Значение t все еще равно: " << t << "\n";

    return 0;
}

// Возврат обратного значения.
double reciprocal(double x)
{
    x = 1 / x; // создадим обратное значение
    return x;
}
```

← Это не изменяет значения **t** внутри **main()**.

Вывод этой программы выглядит следующим образом:

```
Обратное значение от 10.0 составляет 0.1
Значение t все еще равно: 10
```

Как видно из вывода программы, когда внутри **reciprocal()** выполняется присваивание

```
x = 1/x;
```

единственное, что изменяется — это локальная переменная **x**. Переменная **t**, использованная как аргумент, будет все еще иметь значение 10; операции внутри функции никак на нее не повлияют.

Цель

6.3. Использование указателя для создания вызова по ссылке

Несмотря на то, что в C++ по умолчанию используется вызов по значению, можно вручную создать вызов по ссылке, передав в функцию адрес аргумента (т. е. указатель). В этом случае возникает возможность изменения значения аргумента вне функции. Вы видели пример такой операции в предыдущем модуле, где обсуждался вопрос о передаче в функцию указателей. Как вы знаете, указатели передаются в функцию в принципе точно так же, как и любые другие значения. Разумеется, необходимо объявить параметры функции как указатели.

Чтобы убедиться в том, что передача указателя позволяет вручную создать вызов по ссылке, рассмотрим функцию, названную **swap()**, которая обменивает значения двух переменных, на которые указывают ее аргументы. Один из способов реализации функции приведен ниже:

// Обмен значений переменных, на которые указывают x и y.

```
void swap(int *x, int *y)
{
    int temp;

    temp = *x; // сохраним значение с адресом x
    *x = *y; // отправим значение из y в x
    *y = temp; // отправим значение из x в y
}
```

Обмен значений переменных, на которые указывают x и y.

Функция **swap()** объявляет два параметра-указателя **x** и **y**. Она использует эти параметры для обмена значений переменных, на которые указывают аргументы, передаваемые функции. Вспомним, что ***x** и ***y** обозначают переменные, находящиеся по адресам **x** и **y**. Таким образом, предложение

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

помещает объект, на который указывает **y**, в объект, на который указывает **x**. В результате, когда функция завершается, значения переменных, использованных при вызове функции, меняются местами.

Поскольку **swap()** ожидает получения двух указателей, вы должны вызывать **swap()** с указанием *адресов* переменных, у которых надо

обменять значения. Как это делается, показано в приведенной ниже программе:

// Демонстрация варианта swap() с указателями.

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

```
// Объявим, что swap() использует указатели.
void swap(int *x, int *y);
```

```
int main()
{
    int i, j;
```

```
    i = 10;
    j = 20;
```

```
    cout << "Исходные значения i и j: ";
    cout << i << ' ' << j << '\n';
```

Вызов **swap()** с адресами переменных, у которых мы хотим обменять значения.

```
    swap(&j, &i); // вызов swap() с адресами i и j
```

```
    cout << "Новые значения i и j: ";
    cout << i << ' ' << j << '\n';
```

```
    return 0;
}
```

// Обменяем значения, на которые указывают x и y.

```
void swap(int *x, int *y)
```

```
{
    int temp;
```

Обменяем значения переменных, на которые указывают x и y, посредством явных операций с указателями.

```
    temp = *x; // сохраним значение с адресом x
    *x = *y;   // поместим значение с адресом y
              // в переменную с адресом x
    *y = temp; // поместим значение с адресом x
              // в переменную с адресом y
}
```

В функции **main()** переменной **i** присваивается значение 10, а переменной **j** – 20. Затем вызывается **swap()** с адресами **i** и **j** в качестве аргументов. Унарный оператор **&** позволяет получить адреса переменных. В результате в **swap()** передаются не значения переменных, а их адреса. После возврата из **swap()** значения **i** и **j** поме-

няются местами, как это видно из приведенного ниже вывода программы:

Начальные значения *i* и *j*: 10 20

Новые значения *i* и *j*: 20 10

Минутная тренировка

1. Объясните понятие вызова по значению.
 2. Объясните понятие вызова по ссылке.
 3. Какой механизм передачи параметров используется в C++ по умолчанию?
1. При вызове по значению в параметры подпрограммы копируются значения аргументов.
 2. При вызове по ссылке в параметры подпрограммы копируются адреса аргументов.
 3. По умолчанию C++ использует вызов по значению.
-

Цель

6.4. Параметры-ссылки

Как мы видели, с помощью операций над указателями можно вручную организовать передачу по ссылке, однако такой подход несколько неуклюж. Во-первых, вам приходится выполнять все операции посредством указателей. Во-вторых, вы должны помнить, что при вызове функции ей следует передавать не значения переменных, а их адреса. К счастью, в C++ имеется возможность заставить компилятор автоматически использовать для одного или нескольких параметров конкретной функции передачу по ссылке вместо передачи по значению. Это достигается с помощью *параметра-ссылки*. Если вы используете параметр-ссылку, функции автоматически передается адрес (не значение) аргумента. Внутри функции при операциях над параметром-ссылкой автоматически выполняется снятие ссылки, в результате чего исчезает необходимость в использовании указателей.

Параметр-ссылка объявляется помещением перед именем параметра в объявлении функции знака **&**. Операции с использованием параметра-ссылки фактически выполняются над аргументом, указанным при вызове функции, а не над самим параметром-ссылкой.

Чтобы освоить использование параметров-ссылок, начнем с простого примера. В приведенной ниже программе функция `f()` требует один параметр-ссылку типа `int`:

```
// Использование параметра-ссылки.
```

```
#include <iostream>
```

```
using namespace std;

void f(int &i); // здесь i является параметром-ссылкой

int main()
{
    int val = 1;

    cout << "Старое значение val: " << val << '\n';

    f(val); // передадим f() адрес val

    cout << "Новое значение val: " << val << '\n';

    return 0;
}

void f(int &i)
{
    i = 10; // это модифицирует передаваемый аргумент
}
```

Объявим параметр-ссылку.

Эта программа выведет на экран следующее:

```
Старое значение val: 1
Новое значение val: 10
```

Обратите особое внимание на определение `f()`:

```
void f(int &i)
{
    i = 10; // это модифицирует передаваемый аргумент
}
```

Присвоим значение переменной, на которую ссылается `i`.

Посмотрите, как объявляется `i`. Перед параметром помещается знак `&`, который превращает этот параметр в параметр-ссылку. (Такое же объявление используется и в прототипе функции.) Внутри функции предложение

```
i = 10;
```

не присваивает `i` значение 10. Напротив, это значение присваивается переменной, на которую ссылается `i` (в данном случае это переменная `val`). Заметьте, что в этом предложении не используется оператор указателя `*`. В тех случаях, когда вы используете параметр-ссылку, компилятор C++ понимает, что вы имеете в виду адрес переменной, и при обращении к аргументу сам снимает ссылку. Указание здесь оператора `*` было бы серьезной ошибкой.

Поскольку `i` объявлена как параметр-ссылка, компилятор автоматически передаст функции `f()` *адрес* любого аргумента, с которым она вызывается. Так, в функции `main()` предложение

```
f(val); // передадим f() адрес val
```

передает в функцию `f()` адрес `val` (не значение этой переменной). Здесь нет необходимости предварять `val` оператором `&`. (Указание этого оператора было бы серьезной ошибкой.) Поскольку `f()` получает адрес `val` (в форме ссылки), она может модифицировать значение `val`.

Чтобы показать применение параметров-ссылок (и для демонстрации их преимуществ) в приведенную ниже программу включен вариант функции `swap()`, использующей ссылки. Обратите особое внимание на объявление и вызов функции `swap()`:

```
// Использование функции swap() с параметрами-ссылками.
```

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

```
// Объявим swap() с использованием параметров-ссылок.
void swap(int &x, int &y);
```

```
int main()
{
    int i, j;
```

```
    i = 10;
    j = 20;
```

```
    cout << " Начальные значения i и j: ";
    cout << i << ' ' << j << '\n';
```

```
    swap(j, i);
```

Здесь в функцию `swap()` автоматически передаются адреса `i` и `j`.

```
    cout << "Новые значения i и j: ";
    cout << i << ' ' << j << '\n';
```

```
    return 0;
}
```

```
/* Здесь swap() определена как использующая передачу по ссылке,
   а не передачу по значению. В результате функция может обменять
   значения двух аргументов, указанных при ее вызове.
```

```
*/
void swap(int &x, int &y)
```

```

{
    int temp;

    // используем ссылки для обмена значений аргументов
    temp = x;
    x = y;
    y = temp;
}

```

Теперь обмен значений происходит
автоматически посредством ссылок.

Вывод этой программ будет такой же, как в предыдущем варианте. Еще раз обратите внимание на то, что при обращении к параметрам `x` и `y` с целью обмена их значений нет необходимости указывать оператор `*`. Компилятор автоматически определяет адреса аргументов, используемых при вызове `swap()`, и автоматически снимает ссылки с `x` и `y`.

Подытожим сказанное. Когда вы создаете параметр-ссылку, этот параметр автоматически ссылается на аргумент, используемый при вызове функции (т. е. явным образом указывает на него). Далее, отпадает необходимость указывать при аргументе оператор `&`. Так же и внутри функции параметр-ссылка используется непосредственно; оператор `*` не нужен. Все операции с параметром-ссылкой автоматически воздействуют на аргумент, указанный при вызове функции. Наконец, когда вы присваиваете параметру-ссылке некоторое значение, фактически вы присваиваете это значение переменной, на которую указывает ссылка. Применительно к функциям это будет та переменная, которая указана в качестве аргумента при вызове функции.

Последнее замечание. Язык `C` не поддерживает ссылки. Таким образом, организовать передачу параметра по ссылке в `C` можно только с помощью указателя, как это было показано в первом варианте функции `swap()`. Переводя программу с языка `C` на язык `C++`, вы можете, где это будет оправдано, преобразовать параметры-указатели в параметры-ссылки.

Спросим у эксперта

Вопрос: В некоторых программах на `C++` я сталкивался с объявлениями, в которых знак `&` указывается при имени типа, как в этом предложении:

```
int& i;
```

а не при имени переменной, как здесь:

```
int &i;
```

Есть ли разница в этих объявлениях?

Ответ: Короткий ответ будет отрицательным: никакой разницы в этих объявлениях нет. Например, прототип функции `swap()` можно записать таким образом:

```
void swap(int& x, int& y);
```

Здесь знак `&` примыкает вплотную к `int`, и не к `x`. Более того, некоторые программисты, объявляя указатели, предпочитают помещать знак `*` при типе, а не при переменной, например, так:

```
float* p;
```

Такой стиль объявлений отражает желание некоторых программистов, чтобы C++ содержал отдельные типы ссылок или указателей. Однако, при отнесении знаков `&` и `*` к типу, а не к переменной, возникает некоторое неудобство, связанное с тем, что согласно формальному синтаксису C++ оба эти оператора не обладают свойством дистрибутивности при использовании в списке переменных, и это может привести к неправильным объявлениям. Например, в следующем объявлении создается один, а не два указателя на `int`:

```
int* a, b;
```

Здесь `b` объявлено, как целое (а не указатель на целое), потому что, в соответствии с синтаксисом C++, при использовании в объявлениях знаки `*` и `&` связываются с индивидуальной переменной, перед которой они указаны, а не с типом, указываемым перед ними.

С другой стороны, с точки зрения компилятора не имеет никакого значения, будете ли вы писать `int *p` или `int* p`. Поэтому если вам больше нравится относить знаки `*` и `&` к типу, а не к переменной, вы вполне можете так и делать. Однако, чтобы избежать неясностей, в этой книге мы всегда будем связывать знаки `*` и `&` с именем переменной, которую они модифицируют, а не именем типа.

Минутная тренировка

1. Как объявляется параметр-ссылка?
 2. Если вы вызываете функцию, использующую параметр-ссылку, должны ли вы предварять аргумент знаком `&`?
 3. Если функция принимает параметр-ссылку, следует ли внутри функции в операциях над этим параметром указывать знаки `*` или `&`?
 1. Параметр-ссылка объявляется указанием перед именем параметра знака `&`.
 2. Нет, если в функции используется параметр-ссылка, аргумент автоматически передается по ссылке.
 3. Нет, операции с параметром осуществляются, как со значением. Однако изменения параметра отражаются на аргументе, указанном при вызове функции.
-

Цель

6.5. Возврат ссылок

Функции могут возвращать ссылки. При программировании на C++ для возврата ссылок можно найти несколько применений. С некоторыми вы познакомитесь позже в этой книге. Однако ссылка в качестве значения возврата имеет и другие важные применения, которые вы можете использовать уже сейчас.

Когда функция возвращает ссылку, она фактически возвращает указатель на возвращаемое значение. Это приводит к совершенно удивительной возможности: функцию можно использовать с левой стороны предложения присваивания! Рассмотрим, например, такую программу:

```
// Возврат ссылки.

#include <iostream>
using namespace std;

double &f(); // функция возвращает ссылку. ← Здесь f() возвращает
                                                    ссылку на double.

double val = 100.0;

int main()
{
    double x;

    cout << f() << '\n'; // выведем значение val

    x = f(); // присвоим переменной x значение val
    cout << x << '\n'; // выведем значение x

    f() = 99.1; // изменим значение val
    cout << f() << '\n'; // выведем новое значение val

    return 0;
}
// Эта функция возвращает ссылку на double.
double &f()
{
    return val; // возврат ссылки на val ← Это предложение возвращает
                                                    ссылку на глобальную
                                                    переменную val.
```

Вывод этой программы выглядит таким образом:

```
100
100
99.1
```

Рассмотрим ход выполнения программы. В начале программы объявлена функция `f()`, которая возвращает ссылку на тип **double**, а также глобальная переменная **val**, инициализируемая значением 100. В функции `main()` следующее предложение выводит на экран исходное значение **val**:

```
cout << f() << '\n'; // выведем значение val
```

Когда вызывается `f()`, она возвращает ссылку на **val** (не значение **val**!). Эта ссылка затем используется в предложении `cout` для вывода на экран значения **val**.

В строке

```
x = f(); // присвоим переменной x значение val
```

ссылка на **val**, возвращенная функцией `f()`, присваивает значение **val** переменной **x**.

Самая интересная строка в программе приведена ниже:

```
f() = 99.1; // изменим значение val
```

Это предложение приводит к изменению значения **val** на 99.1. Вот почему: поскольку `f()` возвращает ссылку на **val**, эта ссылка становится мишенью для предложения присваивания. В результате значение 99.1 присваивается переменной **val** косвенным образом, через ссылку на эту переменную, возвращаемую функцией `f()`.

Вот другой пример простой программы, использующей возврат ссылки:

```
// Возврат ссылки на элемент массива.

#include <iostream>
using namespace std;

double &change_it(int i); // 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; // изменим 2-й элемент
change_it(3) = -98.8; // изменим 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]; // возвращает ссылку на i-й элемент
}

```

Эта программа изменяет значения второго и четвертого элементов массива **vals**. Программа выводит следующее:

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

Посмотрим, как эта программа работает.

Функция **change_it()** объявлена как возвращающая ссылку на **double**. Конкретно она возвращает ссылку на элемент массива **vals**, задаваемый параметром функции **i**. Ссылка, возвращаемая функцией **change_it()**, затем используется в функции **main()** для присваивания значения этому элементу.

Используя ссылку в качестве возвращаемого значения, следите за тем, чтобы объект, на который указывает эта ссылка, находился в области видимости. Например, рассмотрим такую функцию:

```

// Ошибка, нельзя вернуть ссылку на локальную переменную.
int &f()
{
    int i = 10;

    return i;
}

```

Ошибка! После возврата из функции **f()** переменная **i** будет вне области видимости.

Когда произойдет возврат из функции **f()**, переменная **i** будет вне области видимости. В результате ссылка на **i**, возвращаемая **f()**, будет не определена. Некоторые компиляторы вообще не будут компилировать приведенный выше вариант **f()**. Однако такого рода ошибка может возникнуть косвенным образом, поэтому внимательно следите за тем, на какой объект вы возвращаете ссылку.

Цель

6.6. Независимые ссылочные переменные

Несмотря на то, что ссылки включены в C++ главным образом для поддержки передачи параметров по ссылке при вызове функций, а также для использования в качестве возвращаемого функцией значения, имеется возможность объявить отдельную переменную ссылочного типа. Такие переменные называются *независимыми ссылочными переменными*. Следует, правда, сразу сказать, что такие переменные используются относительно редко, так как они усложняют программу и делают ее трудно читаемой. Имея в виду эту оговорку, рассмотрим кратко использование переменных такого типа.

Независимая ссылочная переменная должна указывать на какой-то объект, поэтому ее необходимо инициализировать при объявлении. В общем случае это означает, что ей должен быть присвоен адрес предварительно объявленной переменной. После того, как это сделано, имя ссылочной переменной может быть использовано всюду, где допустимо использовать переменную, на которую она указывает. Фактически эти переменные нельзя различить. Рассмотрим, например, такую программу:

// Использование независимой переменной ссылочного типа.

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

int main()
{
    int j, k;
    int &i = j; // независимая ссылочная переменная ←
    j = 10;

    cout << j << " " << i; // выводит 10 10

    k = 121;
    i = k; // копирует значение k в j (не адрес k)

    cout << "\n" << j; // выводит 121

    return 0;
}
```

Программа выводит следующее:

```
10 10
121
```

Адрес, на который указывает ссылочная переменная, фиксирован; его нельзя изменять. Поэтому, когда выполняется предложение

`i = k;`

в переменную `j` (на которую указывает `i`) копируется значение `k` (не адрес `k`).

Как уже отмечалось ранее, обычно использование независимых переменных ссылочного типа нельзя назвать разумным, потому что в них нет необходимости и они только замусоривают вашу программу. Наличие двух имен для одной и той же переменной не приведет ни к чему, кроме путаницы.

Несколько ограничений при использовании ссылочных переменных

В использовании ссылочных переменных необходимо иметь в виду следующие ограничения:

- Нельзя ссылаться на ссылочную переменную.
- Недопустимо создавать массивы ссылочных переменных.
- Нельзя создать указатель на ссылку. Другими словами, к ссылке нельзя приложить оператор `&`.

Минутная тренировка

1. Может ли функция вернуть ссылку?
2. Что такое независимая ссылочная переменная?
3. Можно ли создать ссылку на ссылку?

- 1 Да, функция может вернуть ссылку
 - 2 Независимая ссылочная переменная — это просто другое имя того же объекта
 - 3 Нет, вы не можете создать ссылку на ссылку
-

Цель

6.7. Перегрузка функций

В этом подразделе вы познакомитесь с одной из самых захватывающих средств C++: перегрузкой функций. В C++ две или даже несколько функций могут иметь одно и то же имя при условии, что различаются объявления их параметров. Такие функции называются *перегруженными*, а само это средство называют *перегрузкой функций*. Перегрузка функций является одним из способов реализации полиморфизма в C++.

Для того, чтобы перегрузить функцию, достаточно объявить другой вариант функции с тем же именем. Компилятор возьмет на себя все остальное. Вы только должны соблюсти одно важное условие: типы или число параметров (или и то, и другое) каждой из перегруженных функций должны различаться. Для перегруженных функций недостаточно различия только в типе возвращаемых значений. Они должны различаться типом или числом их параметров. (Типы возврата не во всех случаях предоставляют достаточную информацию для C++, чтобы тот мог решить, какую из функций использовать.) Разумеется, перегруженные функции *могут* различаться также и типами возвращаемых значений. Когда вызывается перегруженная функция, реально выполняется тот вариант, у которого параметры соответствуют аргументам вызова.

Начнем с простого программного примера:

// Перегрузим функцию трижды.

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

```
void f(int i);           // параметр типа int
void f(int i, int j);    // два параметра типа int
void f(double k);        // один параметр типа double
```

Для каждого варианта перегруженной функции требуется отдельный прототип

```
int main()
{
    f(10); // вызов f(int)

    f(10, 20); // вызов f(int, int)

    f(12.23); // вызов f(double)

    return 0;
}
```

```
void f(int i)
{
    cout << "В f(int) i равно " << i << '\n';
}
```

```
void f(int i, int j)
{
    cout << "В f(int, int) i равно " << i;
    cout << ", j равно " << j << '\n';
}
```

```
void f(double k)
```

Здесь определены три различающихся реализации f()

```
{
    cout << "В f(double) k равно " << k << '\n';
}
```

Эта программа выведет на экран следующее:

```
В f(int) i равно 10
В f(int, int) i равно 10, j равно 20
В f(double) k равно 12.23
```

Итак, **f()** перегружена трижды. Первый вариант требует один целочисленный параметр, второй вариант требует два целочисленных параметра, и третий вариант требует один параметр типа **double**. Из-за того, что списки параметров каждой функции различаются, компилятор имеет возможность выбрать требуемый вариант функции исходя из типа аргументов, указанных при вызове функции.

Для того, чтобы осознать ценность перегрузки функций, рассмотрим функцию, названную **neg()**, которая возвращает отрицание своего аргумента. Например, при вызове с числом -10 функция возвращает 10 . При вызове с числом 9 функция возвращает -9 . Не имея средства перегрузки функций, вы должны были бы для данных с типами **int**, **double** и **long** разработать три отдельные функции с различающимися именами, например, **ineg()**, **lneg()** и **fneg()**. Благодаря перегрузке мы можете для всех функций, возвращающих отрицание своего аргумента, использовать одно имя, например, **neg()**. Таким образом, перегрузка поддерживает концепцию полиморфизма “один интерфейс, много методов”. Приведенная ниже программа демонстрирует эту концепцию:

```
// Создание различных вариантов функции neg().
```

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

int neg(int n);          // neg() для int.
double neg(double n);    // neg() для double.
long neg(long n);        // neg() для long.

int main()
{
    cout << "neg(-10): " << neg(-10) << "\n";
    cout << "neg(9L): " << neg(9L) << "\n";
}
```

```
cout << "neg(11.23): " << neg(11.23) << "\n";

return 0;
}

// neg() для int.
int neg(int n)
{
    return -n;
}

// neg() для double.
double neg(double n)
{
    return -n;
}

// neg() для long.
long neg(long n)
{
    return -n;
}
```

Вот вывод этой программы:

```
neg(-10): 10
neg(9L): -9
neg(11.23): -11.23
```

В программе предусмотрены три схожие, но все же разные функции с именем **neg**, каждая из которых возвращает обратное значение своего аргумента. Компилятор определяет, какую функцию вызывать в каждой ситуации, по типу аргумента вызова.

Ценность перегрузки заключается в том, что она позволяет обращаться к набору функций с помощью одного имени. В результате имя **neg** описывает *обобщенное действие*. Обязанность выбрать *конкретный* вариант для *конкретной* ситуации возлагается на компилятор. Вам, программисту, нужно только запомнить имя обобщенного действия. Таким образом, благодаря применению полиморфизма число объектов, о которых надо помнить, сокращается с трех до одного. Хотя приведенный пример крайне прост, вы можете, развив эту концепцию, представить, как перегрузка помогает справляться с возрастающей сложностью.

Другое преимущество перегрузки функций заключается в возможности определять слегка различающиеся варианты одной и той же функции, каждый из которых предназначен для определенного типа данных. В качестве примера рассмотрим функцию с

именем `min()`, которая находит меньшее из двух значений. Не трудно написать варианты `min()`, которые будут выполняться по-разному для данных различных типов. Сравнивая два целых числа, `min()` вернет меньшее из них. Сравнивая два символа, `min()` может вернуть букву, стоящую в алфавите ранее другой, независимо от того, прописные эти буквы или строчные. В таблице ASCII прописные буквы имеют значения, на 32 меньшие, чем соответствующие строчные. Таким образом, игнорирование регистра букв может быть полезным при упорядочении по алфавиту. Сравнивая два указателя, можно заставить `min()` сравнивать значения, на которые указывают эти указатели, и возвращать указатель на меньшее из них. Ниже приведена программа, реализующая все эти варианты `min()`:

```
// Создадим различные варианты min().

#include <iostream>
using namespace std;

int min(int a, int b);    // min() для int
char min(char a, char b); // min() для char
int * min(int *a, int *b); // min() для int*

int main()
{
    int i=10, j=22;

    cout << "min('X', 'a'): " << min('X', 'a') << "\n";
    cout << "min(9, 3): " << min(9, 3) << "\n";
    cout << "min(&i, &j): " << min(&i, &j) << "\n";

    return 0;
}

// min() для int. Возвращает меньшее значение.
int min(int a, int b) ←————— Каждый вариант min() может иметь различия
{                                     Этот вариант возвращает меньшее из
    if(a < b) return a;                двух значений int.
    else return b;
}

// min() для char - игнорирование регистра букв.
char min(char a, char b) ←————— Этот вариант min() игнорирует регистр букв
{
    if(tolower(a) < tolower(b)) return a;
    else return b;
}
```

```

/*
    min() для указателей на int.
    Сравнивает значения и возвращает указатель на меньшее
    значение.
*/

int * min(int *a, int *b) ← Этот вариант min( ) возвращает указатель
                           на меньшее значение
{
    if(*a < *b) return a;
    else return b;
}

```

Вот вывод этой программы:

```

min('X', 'a'): a
min(9, 3): 3
*min(&i, &j): 10

```

Когда вы перегружаете функцию, каждый вариант этой функции может выполнять любые нужные вам действия. Нет никаких правил, устанавливающих, что перегруженные функции должны быть похожи друг на друга. Однако с точки зрения стиля перегрузка функций предполагает их взаимосвязь. Поэтому, хотя вы и можете дать одно и то же имя перегруженным функциям, выполняющим совсем разные действия, этого делать не следует. Например, вы можете выбрать имя `sqr()` для функций, одна из которых возвращает *квадрат int*, а другая — *квадратный корень* из значения `double`. Однако эти две операции фундаментально различаются, и использование для них понятия перегрузки функций противоречит исходной цели этого средства. (Такой способ программирования считается исключительно дурным стилем!) В действительности перегружать следует только тесно связанные операции.

Автоматическое преобразование типов и перегрузка

Как уже отмечалось в Модуле 2, C++ в определенных случаях выполняет автоматическое преобразование типов. Эти преобразования также применимы к параметрам перегруженных функций. Рассмотрим следующий пример:

```

/*
    Автоматическое преобразование типа может влиять
    на выбор перегруженной функции.
*/

```

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

void f(int x);

void f(double x);

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;


    f(i); // вызывается f(int)
    f(d); // вызывается f(double)

    f(s); // вызывается f(int) - преобразование типа
    f(r); // вызывается f(double) - преобразование типа

    return 0;
}

void f(int x) {
    cout << "Внутри f(int): " << x << "\n";
}

void f(double x) {
    cout << "Внутри f(double): " << x << "\n";
}
```



Здесь осуществляется автоматическое преобразование типов

Вывод программ выглядит следующим образом:

```
Внутри f(int): 10
Внутри f(double): 10.1
Внутри f(int): 99
Внутри f(double): 11.5
```

В этом примере определены только два варианта функции **f()**: один, принимающий параметр типа **int**, и другой для параметра типа **double**. Однако вполне возможно передавать **f()** значения типа **short** или **float**. Параметр типа **short** C++ автоматически преобразует в **int** и вызовет **f(int)**. Значение типа **float** будет автоматически преобразовано в **double**, что приведет к вызову **f(double)**.

Важно, однако отдавать себе отчет в том, что автоматическое преобразование действует лишь в тех случаях, когда компилятор не находит перегруженной функции с точным соответствием типов параметра

и аргумента. Ниже приведен предыдущий программный пример с добавлением варианта `f()` для параметра типа `short`:

```
// Теперь добавим f(short).
```

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

void f(int x);
void f(short x); ← Добавлена функция f(short)
void f(double x);

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;

    f(i); // вызывается f(int)
    f(d); // вызывается f(double)

    f(s); // теперь вызывается f(short) ← Теперь преобразование
                                                типа не происходит, так
                                                как имеется f(short)

    f(r); // вызывается f(double) - преобразование типа
}

void f(int x) {
    cout << "Внутри f(int): " << x << "\n";
}

void f(short x) {
    cout << "Внутри f(short): " << x << "\n";
}

void f(double x) {
    cout << "Внутри f(double): " << x << "\n";
}
```

Теперь, запустив программу, вы получите на экране следующее:

```
Внутри f(int): 10
Внутри f(double): 10.1
Внутри f(short): 99
Внутри f(double): 11.5
```

В этом варианте, поскольку здесь добавлена перегруженная функция `f()`, принимающая аргумент типа `short`, при вызове `f()` со значением `short` будет активизирована `f(short)`, и автоматического преобразования `short` в `int` не происходит.

Минутная тренировка

1. Какое условие необходимо выполнить при перегрузке функции?
 2. Почему перегруженные функции должны выполнять схожие действия?
 3. Участвует ли тип возвращаемого функцией значения в выборе компилятором варианта перегруженной функции?
-
1. Перегруженные функции имеют одно имя, но объявления их параметров различаются.
 2. Перегруженные функции должны выполнять схожие действия, потому что перегрузка функций предполагает взаимосвязь между функциями
 3. Нет, тип возвращаемого значения перегруженных функций может различаться, однако он не влияет на выбор компилятором конкретного варианта функции
-

Проект 6-1

Создание перегруженных функций для вывода на экран

В этом проекте вы создадите набор перегруженных функций, выводящих на экран данные различных типов. Хотя предложение `cout` достаточно удобно, такой набор функций вывода предоставит альтернативу, которая может прельстить некоторых программистов. В действительности и Java, и C++ используют не операторы вывода, а функции вывода. Создав перегруженные функции вывода, вы сможете пользоваться обоими методами. Кроме того, вы сможете придать вашим функциям вывода конкретные характеристики, удобные для решения ваших задач, например, при выводе булевых переменных выводить не 1 и 0, а слова “true” и “false”.

Вам надо будет создать два набора функций с именами `println()` и `print()`. Функция `println()` будет отображать свой аргумент и вслед за тем переводить курсор на следующую строку. Функция `print()` будет просто выводить свой аргумент, без перевода строки. Например, фрагмент:

```
print(1);
println('X');
print("Перегрузка функция - мощное средство. ");
print(18.22);
```

выведет на экран следующее:

```
1X
Перегрузка функция - мощное средство. 18.22
```


В этом проекте `println()` и `print()` будут перегружены для данных типов `bool`, `char`, `int`, `long`, `char*` и `double`, однако вы сможете добавить перегруженные функции для ваших собственных типов.

Шаг за шагом

1. Создайте файл с именем **Print.cpp**.
2. Начните проект со следующих строк:

```
/*
    Проект 6-1
    Создание перегруженных функций print() и println(),
    которые выводят на экран данные разных типов.
*/

include <iostream>
using namespace std;
```

3. Добавьте прототипы для функций `println()` и `print()`, как это показано ниже:

```
// Эти функции осуществляют перевод строки.
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// Эти функции не переводят строку.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);
void print(double d);
```

4. Реализуйте функции `println()`, как показано ниже:

```
// Вот набор функций println().
void println(bool b)
{
    if(b) cout << "true\n";
    else cout << "false\n";
}
```

```
void println(int i)
{
    cout << i << "\n";
}

void println(long i)
{
    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

void println(char *str)
{
    cout << str << "\n";
}

void println(double d)
{
    cout << d << "\n";
}
```

Заметьте, что каждая функция присоединяет к выводу символ новой строки. Обратите внимание также на то, что **println(bool)**, отображая значения булевых переменных, выводит на экран слова “true” или “false”. Это иллюстрация того, как легко можно настроить форму вывода, исходя из ваших потребностей и вкусов.

5. Реализуйте функции `print()`, как показано ниже:

```
// Вот набор функций print()
void print(bool b)
{
    if(b) cout << "true";
    else cout << "false";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{

```

```
    cout << i;
}

void print(char ch)
{
    cout << ch;
}

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}
```

Эти функции практически совпадают со своими аналогами из набора **println()**, за исключением того, что они не переводят строку. В результате последующий вывод появляется на той же строке.

6. Ниже приведен полный текст программы **Print.cpp**:

```
/*
    Проект 6-1
    Создание перегруженных функций print() и println(),
    которые выводят на экран данные разных типов.
*/
#include <iostream>
using namespace std;

// Эти функции осуществляют перевод строки.
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// Эти функции не переводят строку.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);
void print(double d);
```

```
int main()
{
    println(true);
    println(10);
    println("Это проверка");
    println('x');
    println(99L);
    println(123.23);

    print("Вот некоторые значения: ");
    print(false);
    print(' ');
    print(88);
    print(' ');
    print(100000L);
    print(' ');
    print(100.01);

    println(" Все!");

    return 0;
}
// Вот набор функций println().
void println(bool b)
{
    if(b) cout << "true\n";
    else cout << "false\n";
}

void println(int i)
{
    cout << i << "\n";
}

void println(long i)
{
    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

void println(char *str)
{

```

```
cout << str << "\n";
}

void println(double d)
{
    cout< d << "\n";
}

// Вот набор функций print().
void print(bool b)
{
    if(b) cout << "true";
    else cout << "false";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{
    cout << i;
}

void print(char ch)
{
    cout << ch;
}

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}
```

Ниже показан вывод программы:

```
true
10
Это проверка
x
```

99

123.23

Вот некоторые значения: false 88 100000 100.01 Все!

Цель

6.8. Аргументы функций с инициализацией по умолчанию

Следующее относящееся к функциям средство, которые мы здесь рассмотрим – это *аргументы с инициализацией по умолчанию*. В C++ вы можете задать параметру значение по умолчанию; оно будет автоматически использоваться, когда в вызове функции отсутствует аргумент, соответствующий этому параметру. Такие аргументы с инициализацией по умолчанию можно использовать для упрощения вызова сложных функций. Они также иногда могут выполнять те же задачи, что и перегрузка функций, но более простыми средствами.

Аргумент с инициализацией по умолчанию указывается синтаксически так же, как и переменная с инициализацией. Рассмотрим следующий пример, в котором объявляется функция `myfunc()`, принимающая два аргумента типа `int`. Первому аргументу по умолчанию дается значение 0, второму – 100:

```
void myfunc(int x = 0, int y = 100);
```

Теперь `myfunc()` может быть вызвана любым из трех приведенных ниже способов:

```
myfunc(1, 2); // оба значения передаются явным образом
```

```
myfunc(10);   // для x передается значение, y использует
               // умолчание
```

```
myfunc();     // пусть и x, и y используют умолчание
```

В первом вызове функции параметру `x` передается значение 1, а параметру `y` – 2. Во втором вызове `x` получает значение 10, а `y` будет иметь значение 100 по умолчанию. В третьем вызове оба параметра получают значения по умолчанию. Следующая программа демонстрирует этот процесс:

```
// Демонстрация аргументов с инициализацией по умолчанию.
```

```
#include <iostream>
```

```
using namespace std;
```

```
void myfunc(int x = 0, int y = 100);
```

В **myfunc()** заданы аргументы с инициализацией по умолчанию для обоих параметров

```
int main()
```

```
{
```

```
    myfunc(1, 2);
```

```
    myfunc(10);
```

```
    myfunc();
```

```
    return 0;
```

```
}
```

```
void myfunc(int x, int y)
```

```
{
```

```
    cout << "x: " << x << ", y: " << y << "\n";
```

```
}
```

Приведенный здесь вывод программы подтверждает использование умолчания для аргументов:

```
x: 1, y: 2
```

```
x: 10, y: 100
```

```
x: 0, y: 100
```

При создании функции с инициализацией аргументов по умолчанию значения по умолчанию должны указываться только один раз, и это надо сделать при первом объявлении функции в файле. В предыдущем примере умолчание для аргументов было задано в прототипе **myfunc()**. Если вы попытаетесь задать новые (или даже те же самые) значения по умолчанию в определении **myfunc()**, компилятор сообщит вам об ошибке и не будет компилировать программу.

Несмотря на то, что значения аргументов по умолчанию не могут быть переопределены внутри программы, вы можете задать различные значения по умолчанию для каждого варианта перегруженной функции; другими словами, различные варианты перегруженной функции могут иметь различные умолчания для своих аргументов.

Важно понимать, что все параметры, принимающие значения по умолчанию, должны находиться справа от тех, для которых умолчание не используется. Так, следующий прототип неверен:

```
// Неверно!
```

```
void f(int a = 1, int b);
```

Начав определять параметры, принимающие значения по умолчанию, вы уже не можете указать параметр без умолчания. Объявление вроде следующего так же неверно и компилироваться не будет.

```
int myfunc(float f, char *str, int i=10, int j); // Неверно!
```

Поскольку параметру *i* было дано значение по умолчанию, параметр *j* требует того же.

Одна из причин, по которой в C++ включена инициализация по умолчанию для аргументов функций, заключается в том, что она упрощает задачу программиста при работе со сложными программами. Очень часто функция, ради того, чтобы ее можно было использовать в широком диапазоне ситуаций, имеет больше параметров, чем это требуется для ее вызова в наиболее типичных и распространенных случаях. Тогда при наличии у нее аргументов с умолчанием вам нужно помнить и указывать только те аргументы, которые требуются в конкретной ситуации, а не все те, которые покрывают самый общий случай.

Аргументы с инициализацией по умолчанию или перегрузка?

Одним из приложений аргументов с инициализацией по умолчанию является создание упрощенного аналога перегруженных функций. Чтобы выяснить, почему это так, представьте себе, что вы хотите создать два специфических варианта стандартной функции `strcat()`. Один вариант будет действовать наподобие `strcat()` и присоединять все содержимое одной строки к концу другой. Во втором варианте функция будет требовать еще один (третий) аргумент, указывающий число присоединяемых символов. Таким образом, второй вариант будет присоединять только указанное число символов одной строки к концу другой. Наши специфические функции (назовем их `mystrcat()`) будут иметь следующие прототипы:

```
void mystrcat(char *s1, char *s2, int len);  
void mystrcat(char *s1, char *s2);
```

Первый вариант будет копировать *len* символов из *s2* в конец *s1*. Второй вариант скопирует всю строку с адресом *s2* в конец строки с адресом *s1* и, таким образом, будет действовать аналогично `strcat()`.

Хотя не будет ошибкой реализовать два варианта `mystrcat()` для решения поставленной задачи, имеется и более простой способ. Используя аргумент с инициализацией по умолчанию, вы можете создать

только один вариант `mystrcat()`, который будет выполнять обе операции. Следующая программа демонстрирует эту возможность:

```
// Использование специфического варианта strcat().

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

void mystrcat(char *s1, char *s2, int len = 0);

int main()
{
    char str1[80] = "Это проверка";
    char str2[80] = "0123456789";

    mystrcat(str1, str2, 5); // присоединить 5 символов
    cout << str1 << '\n';

    strcpy(str1, "это проверка"); // повторная инициализация str1

    mystrcat(str1, str2); // присоединить всю строку
    cout << str1 << '\n';

    return 0;
}

// Специфический вариант strcat().
void mystrcat(char *s1, char *s2, int len)
{
    // найдем конец s1
    while(*s1) s1++;

    if(len==0) len = strlen(s2);

    while(*s2 && len) {
        *s1 = *s2; // копируем символы
        s1++;
        s2++;
        len--;
    }

    *s1 = '\0'; // завершающий 0 для s1
}
```

Здесь `len` по умолчанию приравнивается нулю

Здесь `len` указано, и копируются только 5 символов

Здесь `len` по умолчанию равно нулю и присоединяется вся строка

Вывод программы выглядит следующим образом:

```
Это проверка01234  
это проверка0123456789
```

Как видно из текста программы, `mystrcat()` присоединяет от 1 до `len` символов строки, на которую указывает `s2`, к концу строки, на которую указывает `s1`. Однако, если `len` равно 0, что произойдет, если `mystrcat()` вызывается без последнего аргумента, который в этом случае получит значение по умолчанию, `mystrcat()` присоединит всю строку `s2` к концу `s1`. (Таким образом, при `len = 0` функция действует так же, как и стандартная функция `strcat()`.) Используя инициализацию `len` по умолчанию, становится возможным скомбинировать обе операции в одной функции. Как показывает этот пример, иногда использование аргументов с инициализацией по умолчанию позволяет получить упрощенный аналог перегруженных функций.

Правильное использование аргументов с инициализацией по умолчанию

Хотя аргументы с инициализацией по умолчанию являются при их правильном использовании чрезвычайно удобным средством, они могут привести и к неприятностям. Назначение аргументов с инициализацией по умолчанию заключается в обеспечении эффективности и удобства при работе с функцией при сохранении значительной гибкости. С этой целью аргументы с умолчанием должны отражать способ наиболее частого использования функции или предоставлять способы разумного альтернативного использования. Если для данного параметра не существует определенного наиболее часто используемого значения, то нет причин объявлять умолчание для соответствующего аргумента. Фактически объявление умолчания для аргументов, когда для этого нет достаточных оснований, только ухудшает программу, поскольку умолчания будут лишь сбивать с толку любого, кто будет читать вашу программу. Кроме того, инициализация аргумента по умолчанию не должна нарушать работу программы. Нельзя допускать, чтобы случайное использование умолчания для аргумента имело необратимые отрицательные последствия. Представьте себе, что вы забыли указать аргумент, а в результате программа стерла файл с важным данными!

Минутная тренировка

1. Покажите, как объявить `void`-функцию с именем `count`, которая требует двух параметров типа `int` с именами `a` и `b`, причем каждому дается значение по умолчанию, равное 0.
2. Можно ли аргументы с инициализацией по умолчанию объявить и в прототипе функции, и в ее определении?

3. Правильно ли следующее объявление? Если нет, то почему?

```
int f(int x=10, double b);
```

- 1 void count(int a=0, int b=0);
- 2 Нет, аргументы с инициализацией по умолчанию должны быть объявлены в первом объявлении функции, которое обычно является ее прототипом
- 3 Объявление неправильно, поскольку параметр без умолчания не может следовать за параметром с умолчанием

Цель

6.9. Перегрузка функций и неоднозначность

Перед тем, как завершить этот модуль, необходимо обсудить вид ошибки, специфический для языка C++: *неоднозначность*. Возможно создать ситуацию, в который компилятор не может выбрать между двумя (или большим числом) правильно перегруженных функций. Если такое случается, то говорят, что возникла *неоднозначность*. Неоднозначные предложения являются ошибочными, и программа, содержащая неоднозначность, компилироваться не будет.

Наиболее часто причиной неоднозначности оказывается автоматическое преобразование типов, выполняемое C++. Как известно, C++ автоматически пытается преобразовать типы аргументов, используемых при вызове функции, в типы параметров, определенные этой функцией. Вот пример этого:

```
int myfunc(double d);
// ...
cout << myfunc('c'); // не ошибка, выполняется преобразование
```

Как показывает комментарий, такое предложение не является ошибочным, так как C++ автоматически преобразует символ `c` в его эквивалент типа **double**. Фактически в C++ лишь очень немногие виды преобразований такого рода запрещены. Автоматическое преобразование типов является очень удобным средством, однако оно также оказывается основной причиной возникновения неоднозначности. Рассмотрим следующую программу:

```
// Неоднозначность при перегрузке.

#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);
```

```
int main()
{
    // однозначно, вызов myfunc(double)
    cout << myfunc(10.1) << " ";

    // неоднозначно
    cout << myfunc(10); // Ошибка! ← Какой вариант myfunc( )
                                следует использовать?

    return 0;
}

float myfunc(float i)
{
    return i;
}

double myfunc(double i)
{
    return -i;
}
```

В приведенном примере функция **myfunc()** перегружена так, что она может принимать аргумент как типа **float**, так и типа **double**. В однозначной строке вызывается **myfunc(double)**, потому что всем константам с плавающей точкой в C++ автоматически назначается тип **double** (если только они не объявлены явным образом типа **float**). Однако, когда **myfunc()** вызывается с целочисленным аргументом 10, возникает неоднозначность, так как компилятор не может определить, в какой тип следует преобразовать константу: **float** или **double**. И то, и другое преобразование в данном случае допустимо. Такая ситуация вызывает сообщение об ошибке и предотвращает компиляцию программы.

Следует особо подчеркнуть, что неоднозначность в предыдущем примере не является следствием перегрузки **myfunc()** относительно типов **double** и **float**. Неясность возникла в результате конкретного вызова **myfunc()** с использованием аргумента не определенного типа. Иначе говоря, ошибкой является не перегрузка **myfunc()**, а вид ее конкретного вызова.

Вот другой пример неоднозначности, вызванной автоматическим преобразованием в C++:

```
// Другой пример неоднозначности при перегрузке.

#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) << " "; // Ошибка, неоднозначность!
    return 0;
}

char myfunc(unsigned char ch)
{
    return ch-1;
}

char myfunc(char ch)
{
    return ch+1;
}

```

Следует ли преобразовать 88 в **char** или в **unsigned char**?

В C++ **unsigned char** и **char** не являются внутренне неоднозначными. (Это разные типы.) Однако, когда **myfunc()** вызывается с числом 88 в качестве аргумента, компилятор не знает, какую функцию вызвать. Следует ли преобразовать 88 в **char** или в **unsigned char**? Оба преобразования вполне допустимы.

Другая ситуация, которая может привести к неоднозначности, возникает при использовании в перегруженной функции аргумента с инициализацией по умолчанию. Чтобы убедиться в этом, рассмотрим такую программу:

```

// Еще пример неоднозначности при перегрузке функций.

#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
    cout << myfunc(4, 5) << " "; // однозначно
    cout << myfunc(10); // Ошибка, неоднозначность!
    return 0;
}

int myfunc(int i)
{

```

Предполагается ли j по умолчанию, или вызывается вариант **myfunc()** с одним параметром?

```
    return i;
}

int myfunc(int i, int j)
{
    return i*j;
}
```

В первом вызове `myfunc()` указаны два аргумента; следовательно, никакой неоднозначности нет, вызывается `myfunc(int i, int j)`. Однако второй вызов `myfunc()` приводит к неоднозначности, поскольку компилятор не знает, вызвать ли вариант `myfunc()` с одним аргументом, или использовать умолчание в варианте с двумя аргументами.

В своих программах на C++ вы наверняка будете сталкиваться с ошибками неоднозначности. При этом, пока вы не приобретете достаточный опыт, таких ошибок в ваших программах будет много, так как допустить их легко, а обнаружить заранее — трудно.

✓ Вопросы для самопроверки

1. Какими двумя способами можно передать аргумент в подпрограмму?
2. Что называется ссылкой в C++? Как создается параметр-ссылка?
3. Дан фрагмент программы:

```
int f(char &c, int *i);
// ...
char ch = 'x';
int i = 10;
```

- Покажите, как вызвать `f()` для операций над переменными `ch` и `i`.
4. Создайте `void`-функцию с именем `round()`, которая округляет значение своего аргумента типа `double` до ближайшего целого значения. Пусть `round()` использует параметр-ссылку и возвращает округленный результат через этот параметр. Продемонстрируйте вызов `round()` в программе. Для решения этой задачи вам понадобится функция `modf()` из стандартной библиотеки. Она имеет такой протип:

```
double modf(double num, double *i);
```

Функция `modf()` разлагает `num` на целую и дробную части. Она возвращает дробную часть и помещает целую часть в переменную, на которую указывает `i`. Функция требует заголовков `<cmath>`.

5. Модифицируйте вариант `swap()` со ссылкой, чтобы помимо обмена значений ее аргументов, она еще возвращала ссылку на меньший аргумент. Назовите эту функцию `min_swap()`.
6. Почему функция не может вернуть ссылку на локальную переменную?
7. Чем должны различаться списки параметров двух перегруженных функций?
8. В Проекте 6-1 вы создали набор функций `print()` и `println()`. Добавьте в эти функции второй параметр, который будет задавать величину отступа. Например, при вызове `print()` таким образом:

```
print("проба",18);
```

строка “проба” будет выведена на экран после 18 пробелов. Назначьте параметру отступа значение 0 по умолчанию, чтобы при его отсутствии текст выводился без отступа.

9. Покажите, какими способами можно вызвать `myfunc()`, имеющую такой прототип:

```
bool myfunc(char ch, int a=10, int b=20);
```

10. Кратко объясните, почему перегрузка функций может привести к неоднозначности.

Модуль 7 Подробнее о типах данных и операторах

Цели, достигаемые в этом модуле

- 7.1 Познакомиться с описателями `const` и `volatile`
- 7.2 Узнать об описателях классов памяти
- 7.3 Начать работать со статическими переменными
- 7.4 Научиться повышать производительность с помощью регистровых переменных
- 7.5 Познакомиться с перечислимыми типами
- 7.6 Рассмотреть оператор `typedef`
- 7.7 Освоить побитовые операции
- 7.8 Начать использовать операции сдвига
- 7.9 Познакомиться с оператором `?`
- 7.10 Понять, как используется оператор-запятая
- 7.11 Освоить операции составного присваивания

В этом модуле мы возвращаемся к понятиям типов данных и операторов. В дополнение к типам данных, которые вы уже использовали, C++ поддерживает целый ряд других. Некоторые из них создаются с помощью модификаторов, добавляемых к типам, о которых вы уже знаете. Кроме того, имеются еще перечислимые типы данных, а с помощью оператора **typedef** можно определять собственные типы. C++ также предоставляет несколько дополнительных операторов, существенно расширяющих диапазон программистских задач, которые удобно решать с помощью этого языка. К ним относятся операторы побитовые и сдвигов, а также операторы **?** и **sizeof**.

Цель

7.1. Описатели **const** и **volatile**

В C++ имеются два типа описателей, влияющих на способы доступа к данным и их модификации. Это описатели **const** и **volatile**. Вместе они называются *cv-описателями*, и в предложении объявления переменной они указываются перед базовым типом.

const

Если переменная объявлена с описателем **const**, ее значение не может быть изменено в процессе выполнения программы. Таким образом, **const**-“переменная” в действительности совсем не является переменной! Однако вы можете назначить переменной, объявленной как **const**, начальное значение. Например, предложение

```
const int max_users = 9;
```

создает переменную **max_users** типа **int**, которая содержит значение 9. Эту переменную можно использовать в выражениях, как и любую другую переменную, однако ее значение не может быть модифицировано вашей программой.

Обычно **const** используется для создания *именованной константы*. Часто программы используют одно и то же значение для многих целей. Например, в программе может быть объявлено несколько различных массивов одного размера. В этом случае вы можете задать размеры всех массивов с помощью **const**-переменной. Достоинство такого приема заключается в том, что если позже вам потребуется изменить размеры этих массивов, то достаточно только задать новое значение **const**-переменной и перекомпилировать программу. Вам не надо будет изменять размер каждого массива в его объявлении. Такой подход помогает избежать ошибок и, кроме того, он проще и нагляднее обычно-

го объявления. Приведенный ниже пример иллюстрирует это применение описателя **`const`**:

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

const int num_employees = 100;

int main()
{
    int empNums[num_employees];
    double salary[num_employees];
    char *names[num_employees];
    // ...
}
```

Здесь создается именованная константа с именем **`num_employees`**, имеющая значение 100.

`num_employees` используется здесь для задания размеров массивов

В этом примере, если вам понадобится изменить размер массивов, вам надо только изменить объявление **`num_employees`** и перекомпилировать программу. Все три массива автоматически изменяют свои размеры.

Другое важное применение описателя **`const`** заключается в защите объекта от модификации посредством указателя. Например, вы хотите предотвратить изменение значения объекта, на который указывает параметр-указатель некоторой функции. Для этого достаточно объявить этот параметр-указатель с описателем **`const`**. После этого функция не сможет модифицировать объект, адресуемый через этот указатель. Другими словами, если параметр-указатель предворяется описателем **`const`**, никакое предложение в функции не сможет модифицировать переменную, на которую указывает этот параметр. Например, в приведенной ниже программе функция **`negate()`** возвращает отрицание значения, на которое указывает ее параметр. Использование **`const`** в объявлении параметра запрещает коду внутри функции как-либо изменять значение, на которой указывает параметр.

// Использование `const` с параметром-указателем.

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

int negate(const int *val);

int main()
{
    int result;
    int v = 10;

    result = negate(&v);
}
```

```

    cout << v << " отрицание: " << result;
    cout << "\n";

    return 0;
}

int negate(const int *val) ← Здесь параметр val объявлен
                             как const-указатель
{
    return - *val;
}

```

Поскольку параметр **val** объявлен, как **const**-указатель, функция не может как-либо изменить значение, на которое указывает **val**. Поскольку приведенная выше функция **negate()** не пытается изменить **val**, программа будет компилироваться и выполняться правильно. Если, однако, **negate()** написана так, как показано в следующем примере, компилятор сообщит об ошибке:

```

// Это работать не будет!
int negate(const int *val)
{
    *val = - *val; // Ошибка, изменение недопустимо
    return *val;
}

```

В этом случае программа пытается изменить значение переменной, на которую указывает **val**, что недопустимо, так как **val** объявлена, как **const**.

Описатель **const** может также использоваться с параметром-ссылкой, чтобы предотвратить модификацию функцией объекта, на который ссылается этот параметр. Например, приведенный ниже вариант функции **negate()** неверен, потому что в нем делается попытка модифицировать переменную, на которую ссылается **val**:

```

// И это работать не будет!
int negate(const int &val)
{
    val = -val; // Ошибка, изменение недопустимо
    return val;
}

```

volatile

Описатель **volatile** сообщает компилятору, что значение переменной может изменяться, хотя в программе нет предложений, явным образом модифицирующих эту переменную. Например, программа обработки

прерываний от таймера может, получив адрес глобальной переменной, обновлять ее с каждым тактом таймера. В этой ситуации содержимое переменной изменяется, хотя в программе нет предложения присваивания этой переменной каких-либо значений. Такое внешнее изменение переменной может привести к неприятным последствиям, потому что компилятору C++ разрешено автоматически оптимизировать определенные выражения в предположении, что содержимое переменной остается неизменным, если ее имя не встречается в левой части выражений присваивания. Если, однако, факторы за пределами программы изменяют значение переменной, могут возникнуть проблемы. Для их предотвращения вы должны объявить такую переменную с описателем **volatile** (изменяемая), как это показано ниже:

```
volatile int current_users;
```

Поскольку переменная **current_users** объявлена как **volatile**, ее значение будет считываться из памяти при каждом обращении к ней.

Минутная тренировка

1. Может ли программа изменить значение **const**-переменной?
 2. Если значение переменной изменяется какими-либо событиями за пределами программы, как следует объявить такую переменную?:
 1. Нет, значение **const**-переменной программа изменить не может.
 2. Если значение переменной изменяется событиями за пределами программы, она должна быть объявлена с модификатором **volatile**
-

Описатели классов памяти

Всего имеется пять описателей классов памяти, поддерживаемых C++:

auto
extern
register
static
mutable

Эти описатели сообщают компилятору, как следует хранить переменную в памяти. Описатели классов памяти должны предшествовать остальной части объявления переменной.

Описатель **mutable** приложим только к объектам *классов*, которые будут обсуждаться в последующих модулях этой книги. Остальные описатели рассматриваются в этом модуле.

auto

Описатель **auto** объявляет локальную переменную. Он, однако, используется редко (если вообще используется), потому что локальным переменным описатель **auto** назначается по умолчанию. Почти невозможно найти программу, в которой использовалось бы это ключевое слово. Оно является пережитком, оставшимся от языка C.

Цель

7.2. extern

Все программы, с которыми вы до сих пор работали, были невелики по размеру. Однако реальные компьютерные программы обычно оказываются очень большими. По мере того, как файл с программой увеличивает свой размер, время компиляции становится настолько большим, что это вызывает раздражение. Если это происходит, вы должны разбить свою программу на два или даже несколько файлов. Тогда изменения в каком-то одном файле не потребуют перекомпиляции всей программы. Вы можете просто перекомпилировать измененный файл, а затем скомпоновать (“слинковать”) его с существующим объектным кодом, содержащим результат компиляции остальных файлов. Такой “многофайловый” подход может дать существенную экономию времени при работе с большими программными проектами. Ключевое слово **extern** помогает реализовать этот подход. Посмотрим, как это делается.

В программах, состоящих из двух или большего числа файлов, каждый файл должен знать имена и типы глобальных переменных, используемых программой. Однако вы не можете просто объявить копии глобальных переменных в каждом файле. Каждая глобальная переменная в программе должна быть только в одном экземпляре. Поэтому если вы попытаетесь объявить глобальные переменные во всех файлах, на этапе компоновки возникнет ошибка. Компоновщик (программа, имеющая обычно имя LINK или TLINK) обнаружит дубликаты глобальных переменных и откажется компоновать программу. Решением этой проблемы служит объявление всех глобальных переменных в одном файле и их использование в других повторных объявлениях с ключевым словом **extern**, как это показано на рис. 7-1.

Первый файл объявляет переменные **x**, **y** и **ch**. Во втором файле имеется копия списка глобальных переменных из первого файла, но к объявлениям добавлен описатель **extern**. Этот описатель делает переменную известной данному модулю, однако фактически не создает ее. Другими словами, **extern** дает компилятору узнать имена и типы глобальных переменных без фактического (повторного) выделения под них памяти. Когда компоновщик будет компоновать оба модуля вместе, все ссылки на внешние переменные оказываются разрешенными.

| <u>Первый файл</u> | <u>Второй файл</u> |
|--|---|
| <pre>int x, y; char ch; int main() { // ... } void func() { x = 123; }</pre> | <pre>extern int x, y; extern char ch; void func22() { x = y/10; } void func23() { y = 10; }</pre> |

Рис. 7-1. Использование глобальных переменных в отдельно компилируемых модулях

До сих пор нас не заботил вопрос о различии между объявлением и определением переменной, однако здесь он приобретает важность. *Объявление* объявляет имя и тип переменной. *Определение* выделяет под переменную память. В большинстве случаев объявление переменной одновременно является и ее определением. Однако предварив имя переменной описателем **extern**, вы можете объявить переменную без ее определения.

Спецификация компоновки extern

Ключевое слово **extern** предоставляет возможность задавать *спецификацию компоновки*, которая представляет собой инструкцию для компилятора относительно того, как функция должна обрабатываться компоновщиком. По умолчанию все функции компонуются как C++-функции, однако спецификация компоновки позволяет вам скомпоновать функцию по правилам другого языка. Общая форма спецификатора компоновки выглядит таким образом:

extern "язык" *прототип-функции*

Здесь *язык* определяет требуемый в данном случае язык программирования. Например, следующее предложение указывает, что функция **myCfunc()** должна быть скомпонована как C-функция:

```
extern "C" void myCfunc();
```

Все компиляторы C++ поддерживают оба языка, и C, и C++. Некоторые также допускают использование спецификаторов компоновки для языков FORTRAN, Pascal или BASIC. (Сверьтесь с документацией к вашему компилятору.) Вы можете задать способ компоновки

сразу для нескольких функций, используя такую форму спецификации компоновки:

```
extern "язык" {  
    прототипы  
}
```

Для большинства программистских задач вам не потребуется использовать спецификацию компоновки.

Цель

7.3. Статические переменные

Переменные типа **static** — это постоянно существующие переменные, действующие внутри своей функции или файла. От глобальных переменных они отличаются тем, что вне своей функции или файла они неизвестны. Описатель **static** по-разному влияет на локальные и глобальные переменные, поэтому те и другие будут здесь рассмотрены отдельно.

Локальные статические переменные

Когда локальной переменной придается описатель **static**, для нее выделяется постоянная память практически так же, как и для глобальных переменных. Это позволяет статической переменной сохранять свое значение при повторных обращениях из функций. (Другими словами, ее значение не теряется при возврате из функции, в отличие от обычной локальной переменной.) Основным отличием локальной переменной класса **static** от глобальной переменной является то, что статическая локальная переменная известна только в том блоке, в котором она объявлена.

Чтобы объявить переменную класса **static**, следует предварить ее тип ключевым словом **static**. Например, следующее предложение объявляет **count** как статическую переменную:

```
static int count;
```

Статической переменной можно присвоить начальное значение. Например, в следующем предложении статической переменной **count** дается начальное значение 200:

```
static int count = 200;
```

Локальная статическая переменная инициализируется только один раз, когда начинается выполнение программы, а не каждый раз при входе в блок, где она была объявлена.

Статические локальные переменные используются в тех случаях, когда требуется сохранять новое значение переменной при повторных вызовах функции. Если бы статических переменных не было, во всех таких случаях пришлось бы использовать глобальные переменные, что чревато возможными побочными эффектами.

Пример статической переменной можно увидеть в приведенной ниже программе. Она вычисляет текущее среднее значение для чисел, вводимых пользователем с клавиатуры:

```
// Вычисление текущего среднего для чисел, вводимых
// пользователем.

#include <iostream>
using namespace std;

int running_avg(int i);

int main()
{
    int num;

    do {
        cout << "Вводите числа (-1 для завершения): ";
        cin >> num;
        if(num != -1)
            cout << "Текущее среднее равно: " << running_avg(num);
        cout << '\n';
    } while(num > -1);

    return 0;
}

int running_avg(int i)
{
    static int sum = 0, count = 0;
    sum = sum + i;

    count++;

    return sum / count;
}
```

Поскольку переменные **sum** и **count** являются статическими, они сохраняют свои значения при повторных вызовах функции **running_avg()**

В этой программе локальные переменные **sum** и **count** объявлены с ключевым словом **static** и инициализированы значением 0. Вспомним, что переменные класса **static** инициализируются только один раз, а не

при каждом входе в функцию. Программа использует переменную **running_avg()** для подсчета и вывода на экран текущего среднего из всех чисел, введенных к этому моменту пользователем. Поскольку переменные **sum** и **count** являются статическими, они сохраняют свои значения при повторных вызовах **running_avg()**, позволяя функции накапливать текущую сумму и количество введенных чисел. Чтобы убедиться в необходимости для этих переменных описателя **static**, попробуйте удалить его из программы и запустить ее снова. Вы увидите, что программа теперь работает неправильно, так как текущая сумма теряется при каждом выходе из функции **running_avg()**.

Глобальные статические переменные

Когда описатель **static** придается глобальной переменной, компилятор создает глобальную переменную, известную только в том файле, в котором эта переменная объявлена. Таким образом, даже несмотря на то, что эта переменная глобальная, другие функции в других файлах ничего о ней не знают и не могут изменить ее значение. Тем самым устраняются возможные побочные эффекты. Для решения частной задачи, если локальная статическая переменная вас не устраивает, вы можете создать небольшой файл, содержащий только функции, которым нужна глобальная статическая переменная, отдельно скомпилировать этот файл и использовать его затем, не опасаясь побочных эффектов.

В качестве примера использования глобальной статической переменной мы модифицируем программу вычисления текущего среднего из предыдущего подраздела. В новом варианте программа разбита на два файла, тексты которых приведены ниже. Программа также дополнена функцией **reset()**, которая сбрасывает в 0 среднее значение.

```
// ----- Первый файл -----  
#include <iostream>  
using namespace std;  
  
int running_avg(int i);  
void reset();  
  
int main()  
{  
    int num;  
    do {  
        cout << "Вводите числа (-1 для завершения, -2 для сброса): ";  
        cin >> num;  
        if(num == -2) {
```

```
        reset();
        continue;
    }
    if(num != -1)
        cout << "Текущее среднее равно: " << running_avg(num);
    cout << '\n';
} while(num != -1);

return 0;
}

// ----- Второй файл -----
static int sum = 0, count = 0; ← Эти переменные известны только в
                               файле, в котором они объявлены

int running_avg(int i)
{
    sum = sum + i;
    count++;
    return sum / count;
}

void reset()
{
    sum = 0;
    count = 0;
}
```

Теперь **sum** и **count** стали глобальными статическими переменными, область видимости которых ограничена вторым файлом. К ним допустимо обращаться из функций **running_avg()** и **reset()** из второго файла, но больше ниоткуда. Их, таким образом, можно сбросить вызовом **reset()** и начать усреднять второй набор чисел. (Запустив программу, вы можете сбросить текущее среднее, введя **-2**.) Однако, никакая функция вне второго файла не может обратиться к этим переменным. Если вы попытаетесь выполнять какие-либо операции над **sum** и **count** из первого файла, вы получите сообщение об ошибке.

Подытожим: имя локальной статической переменной известно только в функции или блоке кода, где она объявлена, а имя глобальной статической переменной известно только в файле, где она находится. В сущности, описатель **static** позволяет создавать переменные, видимые лишь в такой области, в которой они используются, что уменьшает возможность возникновения побочных эффектов. Переменные типа **static** позволяют вам, программисту, скрывать одни области вашей программы от других областей. Эта возможность может иметь огромное значение, если вы работаете с очень большими и сложными программами.

Спросим у эксперта

Вопрос: Я слышал, что некоторые программисты не используют статические глобальные переменные. Это так?

Ответ: Хотя статические глобальные переменные вполне допустимы и широко используются в C++-программах, стандарт C++ не рекомендует их применение. Вместо них предлагается другой метод управления доступом к глобальным переменным, именно, использование пространств имен, которые также будут описаны в этой книге. Однако статические глобальные переменные широко используются программистами на C, потому что C не поддерживает пространства имен. По этой причине вы еще долго будете сталкиваться со статическими глобальными переменными.

Цель

7.4. Регистровые переменные

Возможно, наиболее широко используемым описателем класса памяти является ключевое слово **register**. Этот описатель требует от компилятора сохранить переменную таким образом, чтобы к ней можно было обращаться с максимальной скоростью. Как правило, это означает, что переменная будет храниться либо в регистре процессора, либо в кеш-памяти. Как вы, возможно, знаете, регистры (и кеш-память) работают существенно быстрее, чем основная память компьютера. Поэтому, если переменная хранится в регистре, обращение к ней потребует гораздо меньше времени, чем если она находилась в памяти. Время обращения к переменным оказывает огромное влияние на скорость выполнения вашей программы, поэтому продуманное использование класса **register** является важным программистским приемом.

Строго говоря, описатель **register** только ставит перед компилятором запрос, который компилятор вполне может проигнорировать. Причину этого понять легко: в процессоре имеется лишь ограниченное число регистров (и ограниченный объем памяти с быстрым доступом), причем это число зависит от вычислительной среды. Таким образом, если компилятор уже использовал всю наличную быструю память, он просто сохранит переменную обычным образом. Как правило, это не приводит к каким-либо отрицательным последствиям, но, разумеется, преимущества класса **register** оказываются неиспользованными. Как правило, вы можете рассчитывать на возможность оптимизации по скорости по меньшей мере двух переменных.

Поскольку реально быстрый доступ может быть предоставлен только ограниченному числу переменных, важно тщательно выбрать те, которым вы назначите класс **register**. (Только правильно выбрав переменные, вы добьетесь ощутимого эффекта в производительности.) В

общем можно сказать, что чем чаще переменная используется в программе, тем больше будет выигрыш в скорости при назначении ее регистровой переменной. Так, например, переменные, управляющие циклами или используемые внутри циклов, являются разумными кандидатами на включение их в класс **register**.

Ниже приведен пример использования регистровой переменной для повышения производительности функции **summation()**, вычисляющей сумму значений элементов массива. В этом примере предполагается, что только две переменных будут действительно оптимизированы по скорости доступа к ним:

```
// Демонстрация использования регистровой переменной.

#include <iostream>
using namespace std;

int summation(int nums[], int n);

int main()
{
    int vals[] = { 1, 2, 3, 4, 5 };
    int result;

    result = summation(vals, 5);

    cout << "Сумма равна " << result << "\n";

    return 0;
}

// Возвращает сумму значений элементов массива чисел int.
int summation(int nums[], int n)
{
    register int i;
    register int sum = 0;

    for(i = 0; i < n; i++)
        sum = sum + nums[i];

    return sum;
}
```

Эти переменные будут оптимизированы по скорости доступа.

В этом примере переменная **i**, управляющая циклом **for**, а также **sum**, используемая внутри цикла, объявлены с указанием класса **register**. Поскольку обе они используются в цикле, вы получите выигрыш от убыстрения доступа к ним. В этом примере предполагалось,

что оптимизировать по скорости доступа фактически можно только две переменные, поэтому `n` и `nums` не назначены регистровыми, так как обращение к ним осуществляется не так часто, как к `i` и `sum`. Однако, в среде, где можно оптимизировать более двух переменных, для них также можно было задать описатель `register` с целью дальнейшего повышения производительности.

Спросим у эксперта

Вопрос: Когда я попытался добавить описатель `register` к некоторым переменным в программе, я не ощутил выигрыша в производительности. Почему?

Ответ: Благодаря достижениям в технологии компиляторов, большинство современных компиляторов автоматически выполняют оптимизацию вашего кода. Поэтому во многих случаях добавление описателя `register` не увеличит производительность, так как переменные уже оптимизированы компилятором. Однако в некоторых случаях использование описателя `register` оказывается полезным, поскольку тем самым вы сообщаете компилятору, какие переменные с вашей точки зрения наиболее важно оптимизировать. Это может быть важным в функциях, использующих большое число переменных, когда все их оптимизировать невозможно. Таким образом, описатель `register` все еще выполняет важную роль, несмотря на успехи в разработке компиляторов.

Минутная тренировка

1. Локальная переменная класса `static` _____ свое значение при повторных вызовах функции.
2. Вы используете ключевое слово `extern` для объявления переменной без определения этой переменной. Правильно ли это?
3. Какой описатель требует от компилятора оптимизировать переменную по скорости?
 1. сохраняет
 2. Правильно.
 3. Описатель `register` требует, чтобы компилятор оптимизировал переменную по скорости.

Цель

7.5. Перечислимые типы

В C++ вы можете определить список именованных целочисленных констант. Такой список называется *перечислением*, а сами константы образуют *перечислимый тип*. Эти константы могут быть затем использованы в любом месте, где допустимо использование целого числа.

Перечислимый тип определяется с помощью ключевого слова **enum** следующим образом:

```
enum имя-типа {список-значений} список-переменных;
```

Здесь *имя-типа* определяет новый перечислимый тип, а *список-значений* является перечнем разделенных запятыми имен констант данного перечислимого типа. *список-переменных* не обязателен, потому что переменные могут быть объявлены позже с указанием для них имени данного перечислимого типа.

Следующий фрагмент определяет перечислимый тип с именем **transport**, а также две переменные **t1** и **t2** этого типа:

```
enum transport { car, truck, airplane, train, boat } t1, t2;
```

После того, как вы определили перечислимый тип, вы можете объявлять дополнительные переменные этого типа. Например, следующее предложение объявляет одну переменную типа **transport** с именем **how**:

```
transport how;
```

Это предложение можно записать и иначе:

```
enum transport how;
```

Однако использование здесь ключевого слова **enum** является избыточным. Язык C (который тоже поддерживает перечислимые типы) требует использования именно этой второй формы, поэтому вы сможете встретить ее в некоторых программах.

Считая, что перечислимый тип **transport** объявлен указанным выше образом, следующее предложение присваивает переменной **how** значение **airplane**:

```
how = airplane;
```

При использовании перечислимых типов существенно понимать, что каждая константа перечислимого типа (или просто перечислимая константа) обозначает целое число и может быть использована в любом целочисленном выражении. Если перечислимые константы не инициализированы отдельно, первое значение из списка обозначает 0, второй 1 и т. д. Таким образом, предложение

```
cout << car << ' ' << train;
```

выводит на экран **0 3**.

Хотя перечислимые константы автоматически преобразуются в целочисленные значения, последние автоматически не преобразуются в

перечислимые константы. Например, следующее предложение неправильно:

```
how = 1; // Ошибка
```

Такое предложение приведет к ошибке на этапе компиляции, так как компилятор не выполняет автоматического преобразования **int** в **transport**. Вы можете устранить ошибку в предыдущем предложении, используя приведение типа, как это показано ниже:

```
how = (transport) 1; // теперь правильно, но, возможно,  
// плохой стиль
```

Приведенное предложение присваивает переменной **how** значение **truck**, так как именно эта константа **transport** имеет значение 1. Как указано в комментарии, хотя это предложение вполне правильно, оно будет рассматриваться, как пример плохого стиля программирования, если только оно не используется в силу каких-то необычных обстоятельств.

Имеется возможность задать значения одной или нескольким константам перечислимого типа с помощью инициализатора. Для этого после имени константы следует указать знак равенства и целое число. Если используется инициализатор, каждая следующая константа приобретает значение на 1 большее, чем предыдущая. Например, приведенное ниже предложение присваивает значение 10 константе **airplane**:

```
enum transport { car, truck, airplane = 10, train, boat };
```

Теперь значения констант будут следующими:

| | |
|----------|----|
| car | 0 |
| truck | 1 |
| airplane | 10 |
| train | 11 |
| boat | 12 |

Часто можно встретиться с предположением (неверным), что перечислимые константы могут быть введены или выведены в виде строк, отображающих их имена. Это не так. Например, следующий фрагмент не будет выполняться так, как задумано:

```
// Этот фрагмент не выведет "train" на экран.  
how = train;  
cout << how;
```

Не забывайте, что обозначение **train** представляет собой просто имя целого числа; это не строка. Поэтому приведенный выше фрагмент

выведет на экран числовое значение `train`, а не строку `"train"`. Между прочим, создание кода, который будет вводить и выводить перечислимые константы в виде строк, оказывается довольно утомительным занятием. Например, для того, чтобы вывести на экран словами вид транспортного средства, которому отвечает значение переменной `how`, потребуется следующий код:

```
switch(how) {
    case car:
        cout << "Automobile";
        break;
    case truck:
        cout << "Truck";
        break;
    case airplane:
        cout << "Airplane";
        break;
    case train:
        cout << "Train";
        break;
    case boat:
        cout << "Boat";
        break;
}
```

Иногда оказывается возможным объявить массив строк и использовать значения перечислимых констант в качестве индексов для преобразования значений в соответствующие им строки. Например, приведенная ниже программа выводит на экран имена трех транспортных средств:

```
// Демонстрация работы с перечислимым типом.

#include <iostream>
using namespace std;

enum transport { car, truck, airplane, train, boat };

char name[][20] = {
    "car [автомобиль]",
    "truck [грузовик]",
    "airplane [самолет]",
    "train [поезд]",
    "boat [лодка]"
};

int main()
```



```

{
    transport how;
    how = car;
    cout << name[how] << '\n';

    how = airplane;
    cout << name[how] << '\n';

    how = train;
    cout << name[how] << '\n';

    return 0;
}

```

Значение перечислимой константы используется в качестве индекса массива.

Вот вывод этой программы:

```

car [автомобиль]
airplane [самолет]
train [поезд]

```

Метод, использованный в этой программе для преобразования значения перечислимой константы в строку, может быть применен к любому перечислимому типу, если только он не содержит инициализаторов. Для правильной индексации массива строк значения перечислимых констант должны начинаться с нуля и располагаться далее строго в восходящем порядке, каждое точно на 1 больше предыдущего.

Использование перечислимых констант в качестве индексов строковых массивов широко распространено в программировании. Например, перечисления могут использоваться в компиляторах для определения таблиц символических обозначений.

Цель

7.6. typedef

C++ позволяет вам определять новые имена типов данных с помощью ключевого слова **typedef**. Когда вы используете **typedef**, вы фактически не создаете новый тип данных, а просто даете новое имя существующему типу. Это средство помогает повысить переносимость машинно-зависимых программ; при переносе программы в другую среду достаточно изменить только предложения **typedef**. Новые имена типов также способствуют самодокументированию вашего кода, позволяя использовать в нем описательные имена для стандартных типов данных. Общая форма предложения **typedef** выглядит так:

```
typedef тип имя;
```

Здесь *тип* — это любой допустимый тип данных, а *имя* обозначает новое имя для этого типа. Новое имя не заменяет старое, существующее имя, а является дополнением к нему. Например, вы можете создать новое имя для типа **float**:

```
typedef float balance;
```

Это предложение сообщит компилятору, что он должен распознавать слово **balance** как новое имя для типа **float**. Далее вы можете создать переменную типа **float** с помощью обозначения **balance**:

```
balance over_due;
```

Здесь **over_due** является переменной с плавающей точкой типа **balance**, т. е. фактически типа **float**.

Минутная тренировка

1. Перечислимый тип представляет собой список именованных _____ констант
 2. С какого целого числа начинаются значения перечислимых констант?
 3. Покажите, как объявить **BigInt** другим именем типа **long int**.
 1. целочисленных
 2. Значения перечислимых констант начинаются с 0.
 3.

```
typedef long int BigInt;
```
-

Цель

7.7. Побитовые операторы

Поскольку C++ разрабатывался с целью предоставить полный доступ к аппаратуре компьютера, в нем предусмотрены средства для работы с отдельными битами внутри байта или слова. Эти средства реализованы в виде побитовых операторов. Под *побитовыми операциями* подразумеваются анализ, установка или сдвиг отдельных битов в таких единицах памяти, как байт или слово, которым в C++ соответствуют типы **char** и **int**. Побитовые операции не могут применяться к типам **bool**, **float**, **double**, **long double** и **void**, как и к другим более сложным типам данных. Побитовые операции имеют большое значение для широкого круга задач программирования системного уровня, в которых анализируется или устанавливается информация о состоянии устройства. В табл. 7-1 перечислены побитовые операторы. Далее каждый из них будет рассмотрен отдельно.

Таблица 7-1 Побитовые операторы

| Оператор | Действие |
|----------|---------------------------------|
| & | И (AND) |
| | ИЛИ (OR) |
| ^ | Исключающее ИЛИ (XOR) |
| ~ | Дополнение до единицы, НЕ (NOT) |
| > | Сдвиг вправо |
| < | Сдвиг влево |

Операторы И, ИЛИ, исключающее ИЛИ и НЕ

Побитовые операции И, ИЛИ, исключающее ИЛИ и дополнение до единицы (НЕ) определяются теми же таблицами истинности, что и их логические эквиваленты, за исключением того, что они действуют на битовом уровне. Исключающее ИЛИ (XOR) действует в соответствии со следующей таблицей истинности:

| p | q | $p \wedge q$ |
|---|---|--------------|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 0 | 1 | 1 |

Как показывает эта таблица, результат операции XOR истинен, только если в точности один из операндов истинен; в противном случае результат ложен.

С точки зрения наиболее обычного использования операция побитового И представляет собой способ сброса (т. е. установки в 0) отдельных битов. Другими словами, любой бит, равный 0 в любом из операндов установит в 0 соответствующий бит результата. Например:

```

1 1 0 1 0 0 1 1
1 0 1 0 1 0 1 0
&-----
1 0 0 0 0 0 1 0

```

Приведенная ниже программа демонстрирует использование оператора & для преобразования строчных букв латинского алфавита в прописные путем сброса шестого бита их кода в 0. В наборе символов ASCII строчные буквы идут в том же порядке, что и прописные, но коды строчных букв точно на 32 больше кодов прописных. Таким образом, для преобразования строчных букв в прописные можно просто установить в 0 шестой бит, что и делает эта программа:

```
// Преобразование в прописные буквы с помощью побитового И.

#include <iostream>
using namespace std;

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++) {
        ch = 'a' + i;
        cout << ch;

        // Это предложение сбрасывает 6-й бит.
        ch = ch & 223; // буква ch теперь прописная ←

        cout << ch << " ";
    }

    cout << "\n";

    return 0;
}
```

Используем побитовое И для сброса 6-го бита.

Вывод программы выглядит таким образом:

```
aA bB cC dD eE fF gG hH iI jJ
```

Значение 223, использованное в предложении с оператором `&`, представляет собой десятичное представление числа 1101 1111. Таким образом, операция `И` оставляет все биты в `ch` неизменными, кроме шестого, который устанавливается в 0.

Оператор `&` полезен также в тех случаях, когда вы хотите определять состояние того или иного бита. Например, следующее предложение проверяет, установлен ли 4-й бит в переменной **status**:

```
if(status & 8) cout << "4-й бит установлен";
```

Здесь используется число 8, потому что в двоичном коде оно составляет 0000 1000. Другими словами, число 8, транслируясь в двоичный код, устанавливает в байте только один четвертый бит. Поэтому предложение `if` дает истину, только если четвертый бит переменной **status** установлен. Любопытный пример этой операции можно увидеть в приведенной ниже функции **show_binary**, которая выводит на экран в двоичном формате содержимое своего аргумента. Позже в этом модуле вы будете использовать **show_binary** для изучения действия других побитовых операций.

```
// Отображение битов байта.

void show_binary(unsigned int u)
{
    int t;

    for(t=128; t > 0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

Функция **show_binary** с помощью операции побитового И выполняет последовательный анализ каждого бита в младшем байте переменной **u** с целью определения, установлен этот бит или сброшен. Если бит установлен, на экран выводится символ **1**, если сброшен — выводится **0**.

Побитовая операция ИЛИ, как противоположная И, может быть использована для установки отдельных битов. Любой бит, установленный в 1 в любом из операндов, установит соответствующий бит результата в 1. Например:

```

1 1 0 1 0 0 1 1
1 0 1 0 1 0 1 0
|-----|
1 1 1 1 1 0 1 1
```

С помощью операции ИЛИ можно заставить приведенную выше программу преобразования строчных букв в прописные работать наоборот, преобразуя прописные буквы в строчные:

// Преобразование в строчные буквы с помощью побитового ИЛИ.

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

int main()
{
    char ch;

    for(int i = 0 ; i < 10; i++) {
        ch = 'A' + i;
        cout << ch;
```

Используем побитовое ИЛИ для установки 6-го бита.

```
    // Это предложение устанавливает 6-й бит.
    ch = ch | 32; // буква ch теперь строчная
```

```

    cout << ch << " ";
}

cout << "\n";

return 0;
}

```

Вот вывод этой программы:

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

Установкой шестого бита каждая прописная буква преобразуется в свой строчный эквивалент.

Операция исключающего ИЛИ, обычно обозначаемая как XOR, установит бит, только если сравниваемые биты различны, как это проиллюстрировано ниже:

```

0 1 1 1 1 1 1
1 0 1 1 1 0 0 1
^ -----
1 1 0 0 0 1 1 0

```

Оператор \wedge имеет интересное свойство, которое позволяет простым способом шифровать сообщения. Если над некоторым значением X выполняется операция XOR с другим значением Y , а затем над результатом опять выполняется операция XOR с тем же Y , то в итоге получается первоначальное значение X . Другими словами, после выполнения операций

```

R1 = X  $\wedge$  Y;
R2 = R1  $\wedge$  Y;

```

$R2$ имеет то же значение, что и X . Таким образом, результат последовательности из двух операций XOR, использующих то же значение в качестве второго операнда, дает первоначальное значение. Этим обстоятельством можно воспользоваться для создания простой шифрующей программы, в которой некоторое целое число служит ключом, выступающим в качестве операнда при шифровании и расшифровке сообщения посредством операции XOR над символами сообщения. При шифровании операция XOR применяется первый раз, давая в результате зашифрованный текст. При расшифровке та же операция XOR применяется второй раз, давая в результате исходный текст. Ниже приведен простой пример, использующий этот подход для шифрования и расшифровки короткого сообщения:

// Использование XOR для шифрования и расшифровки сообщения.

```

#include <iostream>
using namespace std;

```

```

int main()
{
    char msg[] = "This is a test";
    char key = 88;

    cout << "Исходное сообщение: " << msg << "\n";
    for(int i = 0 ; i < strlen(msg); i++)
        msg[i] = msg[i] ^ key; ← Здесь создается зашифрованная строка

    cout << "Зашифрованное сообщение: " << msg << "\n";
    for(int i = 0 ; i < strlen(msg); i++)
        msg[i] = msg[i] ^ key; ← Здесь создается расшифрованная строка

    cout << "Расшифрованное сообщение: " << msg << "\n";

    return 0;
}

```

Вот вывод этой программы::

```

Исходное сообщение: This is a test
Зашифрованное сообщение: +01+x1+x9x,=+,
Расшифрованное сообщение: This is a test

```

Как и подтверждает этот вывод, в результате двух операций исключающего ИЛИ с одним и тем же ключом образуется исходное расшифрованное сообщение.

Оператор НЕ (дополнение до единицы, или обратный код) изменяет состояние всех битов операнда на обратное. Например, если некоторое целое под названием А имеет двоичный код 1001 0110, тогда ~А образует код 0110 1001. Приведенная ниже программа демонстрирует действие оператора НЕ, выводя на экран число и его обратный код в двоичной форме с помощью рассмотренной ранее функции `show_binary()`:

```

#include <iostream>
using namespace std;

void show_binary(unsigned int u);

int main()
{
    unsigned u;

    cout << "Введите число между 0 и 255: ";
    cin >> u;

    cout << "Вот двоичный код числа: ";

```

```
show_binary(u);

cout << "Вот обратный код числа: ";
show_binary(~u);

return 0;
}

// Отображение битов байта.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

Ниже приведен результат прогона программы:

```
Введите число между 0 и 255: 99
Вот двоичный код числа: 0 1 1 0 0 0 1 1
Вот обратный код числа: 1 0 0 1 1 1 0 0
```

Все рассмотренные операторы, `&`, `|`, `^` и `~`, выполняют свои операции непосредственно над каждым битом значения независимо от остальных битов. Поэтому побитовые операторы обычно не используются в условных предложениях так, как это имеет место для операторов логических и отношения. Например, если `x` равно 7, то `x && 8` дает истину, а `x & 8` дает ложь.

Цель

7.8. Операторы сдвига

Операторы сдвига, `>>` и `<<`, сдвигают все биты переменной вправо или влево на указанное число битов. Общая форма оператора сдвига вправо:

переменная >> число-битов

Общая форма оператора сдвига влево:

переменная << число-битов

Значение *число-битов* определяет, на сколько битовых мест следует сдвинуть операнд. Каждый сдвиг влево приводит к тому, что все биты указанной переменной сдвигаются влево на одну позицию, а в самый правый бит переменной записывается ноль. Каждый сдвиг вправо приводит к тому, что все биты указанной переменной сдвигаются вправо на одну позицию, а в самый левый бит переменной записывается ноль. Если, однако, переменная представляет собой целое со знаком, содержащее отрицательное значение, тогда каждый сдвиг вправо записывает с левой стороны числа 1, что сохраняет знаковый бит числа. Учтите, что сдвиги влево и вправо не являются циклическими. Другими словами, биты, выдвигаемые с одного конца числа, не переходят в его начало.

Операторы сдвигов используются только с целыми типами, такими как **int**, **char**, **long int** и **short int**. Их нельзя применять, например, к значениям с плавающей точкой.

Побитовые сдвиги могут быть весьма полезны при расшифровке ввода из внешних устройств вроде аналого-цифровых преобразователей и при чтении информации о состоянии устройства. Эти операторы можно также использовать для выполнения очень быстрого умножения и деления целых чисел. Сдвиг влево весьма эффективно умножает число на 2, а сдвиг вправо делит его на 2.

Приведенная ниже программа иллюстрирует действие операторов сдвига:

// Пример операций сдвига.

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

void show_binary(unsigned int u);

int main()
{
    int i=1, t;

    // сдвиг влево
    for(t=0; t < 8; t++) {
        show_binary(i);
        i = i << 1; ← Сдвиг влево переменной i на одну позицию
    }

    cout << "\n";

    // сдвиг вправо
    for(t=0; t < 8; t++) {
        i = i >> 1; ← Сдвиг вправо переменной i на одну позицию
        show_binary(i);
    }
```

```
    return 0;
}

// Отображение битов байта.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t=t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}
```

Программа выводит на экран следующее:

```
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
```

Минутная тренировка

1. Как записываются операторы для побитовых операций И, ИЛИ, НЕ и исключающее ИЛИ?
 2. Побитовый оператор воздействует на каждый бит в отдельности. Правильно ли это?
 3. Покажите, как сдвинуть целочисленную переменную *x* на два бита влево.
 1. Побитовые операторы имеют обозначения *&*, *|*, *~* и *^*.
 2. Правильно.
 3. *x << 2*
-

Проект 7-1**Создание функций
циклического
побитового сдвига**

В C++ имеются два оператора сдвига, но нет оператора циклического сдвига. Циклический сдвиг аналогичен рассмотренным выше операциям сдвига влево и вправо, за исключением того, что бит, выдвинутый из переменной с одного конца, переносится на другой конец. Таким образом, биты не теряются, а только перемещаются. В принципе существуют два циклических сдвига, влево и вправо. Например, число 1010 0000, циклически сдвинутое влево на одну позицию, дает 0100 0001. То же число, циклически сдвинутое вправо на одну позицию, дает 0101 0000. В каждом случае бит, выдвинутый из числа, вставляется в крайнюю позицию с другой стороны. Можно подумать, что отсутствие в C++ операторов циклических сдвигов является его недоработкой, однако в действительности это не так, поскольку, используя другие побитовые операторы, легко сконструировать как лево-, так и правосторонний циклический сдвиг.

В данном проекте создаются две функции: `rrotate()` и `lrotate()`, циклически сдвигающие байт в правом или в левом направлении. Каждая функция принимает два параметра. Первый определяет сдвигаемое значение. Второй задает число битов сдвига. Каждая функция возвращает результат. Проект использует несколько видов манипуляции с битами и демонстрирует побитовые операции в действии.

Шаг за шагом

1. Создайте файл с именем `rotate.cpp`.
2. Включите в файл приведенную ниже функцию `lrotate()`. Она выполняет циклический сдвиг влево.

```
// Циклический сдвиг байта влево на n позиций.
```

```
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;
    for(int i=0; i < n; i++) {

        t = t << 1;

        /* Если бит выдвигается из байта, это будет бит 8
           целого числа t. В этом случае поместим этот бит
           с правой стороны t. */
```

```
    if (t & 256)
        t = t | 1; // поместим 1 в самый правый бит
    }

    return t; // вернем младшие 8 bit.
}
```

Функция `lrotate()` работает следующим образом. Через параметр `val` позиций передается сдвигаемое значение, а через параметр `n` — число позиций сдвига. Функция присваивает значение `val` переменной `t`, которая объявлена как `unsigned int`. Перенос значения в `unsigned int` необходим, он позволяет получить биты, выдвигаемые с левой стороны. И вот почему. Поскольку число `unsigned int` больше, чем байт, когда бит выдвигается с левой стороны байтового числа, он просто смещается в бит 8 числа `int`. Значение этого бита может быть затем скопировано в бит 0 байтового числа, что и обеспечивает циклический сдвиг.

Фактический сдвиг осуществляется следующим образом. Запускается цикл, который выполняет заданное число сдвигов, по одному в каждом шаге. Внутри цикла значение `t` сдвигается влево на одну позицию. В результате этой операции в правый бит `t` записывается 0. Однако, если значение бита 8 результата (а это как раз бит, выдвинутый из байта) равно 1, бит 0 устанавливается в 1. В противном случае бит 0 остается со значением 0.

Бит 8 анализируется предложением

```
if (t & 256)
```

Значение 256 — это десятичное число, в котором установлен один бит 8. Таким образом, `t & 256` будет истинно, только если `t` имело 1 в бите 8.

После завершения сдвига функция возвращает `t`. Поскольку `lrotate()` объявлена с типом возврата `unsigned char`, фактически возвращаются только младшие 8 бит переменной `t`.

3. Добавьте в файл функцию `rrotate()`, приведенную ниже. Она выполняет циклический сдвиг вправо:

```
// Циклический сдвиг байта вправо на n позиций.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // Сначала сдвинем значение на 8 бит влево.
    t = t << 8;

    for(int i=0; i < n; i++) {
        t = t >> 1;
```

```

/* Если бит выдвигается из байта, это будет бит 7
   целого числа t. В этом случае поместим этот бит
   с левой стороны t. */
if (t & 128)
    t = t | 32768; // поместим 1 в самый левый бит
}

/* Наконец сдвинем результат назад
   в младшие 8 бит t. */
t = t >> 8;

return t;
}

```

Циклический сдвиг вправо несколько более сложен, чем сдвиг влево, так как значение, передаваемое в `val`, должно быть сначала перемещено во второй байт `t`, чтобы не потерять биты, выдвигаемые из этого байта с правой стороны. После завершения операции циклического сдвига полученное значение должно быть сдвинуто назад в младший байт `t`, чтобы его можно было использовать в качестве возвращаемого значения. Поскольку бит, выдвигаемый из старшего байта, перемещается в бит 7, следующее предложение проверяет, равно ли его значение 1:

```
if (t & 128)
```

Десятичное число 128 имеет установленным только бит 7. Если бит 7 установлен, над `t` выполняется операция побитового ИЛИ с числом 32768, в котором установлен один только бит 15, а биты 14...0 сброшены. В результате в `t` устанавливается бит 15, а остальные биты остаются без изменения.

4. Ниже приведен полный текст программы, которая демонстрирует использование функций `lrotate()` и `rrotate()`. Программа использует для вывода результата каждого сдвига на экран функцию `show_binary()`.

```

/*
   Проект 7-1
   Функции циклического сдвига байта влево и вправо.
*/

#include <iostream>
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()

```

```
{
    char ch = 'T';

    cout << "Исходное значение в двоичном коде:\n";
    show_binary(ch);

    cout << "Циклический сдвиг вправо 8 раз:\n";
    for(int i=0; i < 8; i++) {
        ch = rrotate(ch, 1);
        show_binary(ch);
    }

    cout << " Циклический сдвиг влево 8 раз:\n";
    for(int i=0; i < 8; i++) {
        ch = lrotate(ch, 1);
        show_binary(ch);
    }

    return 0;
}

// Циклический сдвиг байта влево на n позиций.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;
    for(int i=0; i < n; i++) {
        t = t << 1;
        /* Если бит выдвигается из байта, это будет бит 8
           целого числа t. В этом случае поместим этот бит
           с правой стороны t. */
        if(t & 256)
            t = t | 1; // поместим 1 в самый правый бит
    }

    return t; // вернем младшие 8 бит.
}

// Циклический сдвиг байта вправо на n позиций.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;
    // Сначала сдвинем значение на 8 бит влево.
```

```

t = t << 8;

for(int i=0; i < n; i++) {
    t = t >> 1;
    /* Если бит выдвигается из байта, это будет бит 7
       целого числа t. В этом случае поместим этот бит
       с левой стороны t. */
    if(t & 128)
        t = t | 32768; // поместим 1 в самый левый бит
}

/* Наконец сдвинем результат назад
   в младшие 8 бит t. */
t = t >> 8;

return t;
}

// Выведем на экран биты байта.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

Ниже приведен вывод программы:

Исходное значение в двоичном коде:

0 1 0 1 0 1 0 0

Циклический сдвиг вправо 8 раз:

0 0 1 0 1 0 1 0

0 0 0 1 0 1 0 1

1 0 0 0 1 0 1 0

0 1 0 0 0 1 0 1

1 0 1 0 0 0 1 0

0 1 0 1 0 0 0 1

1 0 1 0 1 0 0 0

0 1 0 1 0 1 0 0

Циклический сдвиг влево 8 раз:

1 0 1 0 1 0 0 0

0 1 0 1 0 0 0 1

```

1 0 1 0 0 0 1 0
0 1 0 0 0 1 0 1
1 0 0 0 1 0 1 0
0 0 0 1 0 1 0 1
0 0 1 0 1 0 1 0
0 1 0 1 0 1 0 0

```

Цель

7.9. Оператор ?

Одним из самых удивительных операторов C++ является оператор `?`. Этот оператор часто используется вместо предложений `if-else` в такой общей форме:

```

if (условие)
    var = выражение1;
else
    var = выражение2;

```

Здесь значение, присваиваемое переменной `var`, зависит от результата вычисления условия, управляющего предложением `if`.

Оператор `?` называется тернарным (троичным) оператором, потому что он требует трех операндов. Его общая форма:

```
Exp1 ? Exp2 : Exp3;
```

Здесь `Exp1`, `Exp2` и `Exp3` — выражения. Обратите внимание на использование и местоположение двоеточия.

Значение выражения `?` определяется следующим образом. Оценивается выражение `Exp1`. Если оно истинно, тогда выражение `Exp2` вычисляется и становится значением всего выражения `?`. Если же `Exp1` ложно, тогда выражение `Exp3` вычисляется и становится значением всего выражения `?`. Рассмотрим пример, в котором переменной `absvalue` присваивается абсолютное значение переменной `val`:

```
absval = val < 0 ? -val : val; // получим абсолютное значение val
```

Здесь переменной `absval` будет присвоено значение `val`, если `val` больше или равно 0. Если же `val` отрицательно, тогда `absval` будет присвоено значение `-val`, что даст, разумеется, положительное значение. Тот же код, написанный с использованием предложения `if-else`, выглядит следующим образом:

```

if(val < 0) absval = -val;
else absval = val;

```


Вот другой пример использования оператора `?`. Приведенная ниже программа делит два числа, но не допускает деления на ноль:

```
/* Эта программа использует оператор ?
   для предотвращения деления на ноль. */

#include <iostream>
using namespace std;

int div_zero();

int main()
{
    int i, j, result;

    cout << "Введите делимое и делитель: ";
    cin >> i >> j;

    // Это предложение предотвращает ошибку деления на ноль.
    result = j ? i/j : div_zero(); ← Оператор ? используется для предот-
                                   вращения ошибки деления на ноль.

    cout << "Результат: " << result;

    return 0;
}

int div_zero()
{
    cout << "Не могу делить на ноль.\n";
    return 0;
}
```

Здесь, если `j` не равно нулю, то `i` делится на `j`, а результат присваивается переменной `result`. В противном случае вызывается обработчик деления на ноль `div_zero()`, и переменной `result` присваивается значение 0.

Цель

7.10 Оператор-запятая

Другим интересным оператором C++ является запятая. Вы уже видели несколько примеров применения этого оператора в цикле `for`, где он позволял выполнить множественные инициализации или приращения. В действительности запятая может быть использована как часть любого выражения. Она сцепляет вместе несколько выражений. Значение спи-

ска выражений, разделенных запятой, совпадает со значением самого правого выражения. Значения других выражений будут отброшены. Это значит, что самое правое выражение становится значением всего выражения с разделяющими запятыми. Например, в предложении

```
var = (count=19, incr=10, count+1);
```

сначала переменная **count** получает значение 19, затем переменной **incr** присваивается значение 10, далее к **count** прибавляется 1 и, наконец, переменная **var** получает значение всего выражения с запятыми, в данном случае 20. Скобки здесь необходимы, потому что оператор-запятая имеет более низкий относительный приоритет, чем оператор присваивания.

Для того, чтобы наглядно наблюдать действие оператора-запятая, попробуйте запустить такую программу:

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

int main()
{
    int i, j;

    j = 10;

    i = (j++, j+100, 999+j); ← Запятая обозначает "делать это, и это, и это".

    cout << i;

    return 0;
}
```

Эта программа выводит на экран "1010". И вот почему. Начальное значение переменной **j** равно 10. Затем **j** увеличивается на 1. Далее **j** прибавляется к числу 100. Наконец, **j** (которая все еще равна 11) прибавляется к числу 999, что дает в результате 1010.

Существенно, что наличие запятой приводит к тому, что выполняется последовательность операций. Если запятая указывается с правой стороны в предложении присваивания, присваиваемое значение определяется, как значение самого правого выражения в списке выражений, разделяемых запятыми. Вы можете в какой-то степени считать, что оператор-запятая имеет то же значение, что и союз "и", если его использовать во фразе "сделай это, и это, и это".

Минутная тренировка

1. Каково будет значение **x** после оценки этого выражения:

```
x = 10 > 11 ? 1 : 0;
```

2. Оператор ? называют троичным оператором, потому что он имеет _____ операнда.
3. Для чего предназначена запятая?
1. Значение x будет равно 0.
 2. три
 3. Запятая заставляет выполняться последовательности предложений.
-

Множественное присваивание

C++ допускает удобный способ присваивания нескольким переменным одно и того же значения, что достигается использованием нескольких операций присваивания в одном предложении. Например, этот фрагмент присвоит переменным `count`, `incr` и `index` значение 10:

```
count = incr = index = 10;
```

В профессионально написанных программах вы часто увидите присваивание переменным общего значения таким образом.

Цель

7.11. Составное присваивание

В C++ предусмотрен специальный оператор составного присваивания, который упрощает вид предложений присваивания определенного рода. Например, предложение

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

может быть написано с использованием оператора составного присваивания следующим образом:

```
x += 10;
```

Пара операторов `+=` требует от компилятора присвоить `x` значение `x` плюс 10. Операторы составного присваивания существуют для всех бинарных операций в C++ (т. е. операций, требующих двух операндов). Общая форма операторов составного присваивания выглядит так:

переменная оператор = выражение;

Вот другой пример:

```
x = x-100;
```

записывается проще как

```
x-=100;
```

Поскольку предложение составного присваивания позволяет экономить на вводимых знаках, его иногда называют *кратким присваиванием*. В профессионально написанных программах на C++ краткое присваивание используется чрезвычайно широко, и вы должны уметь им пользоваться.

Цель

7.12. Использование оператора sizeof

Иногда бывает нужно знать размер в байтах некоторого типа данных. Поскольку размеры встроенных типов C++ различаются для разных компьютерных сред, невозможно заранее указать их для всех ситуаций. Для решения этой проблемы в C++ включен оператор времени компиляции **sizeof**, который имеет следующие общие формы:

sizeof (тип)

sizeof имя-переменной

Первый вариант возвращает размер указанного типа данных, а второй — размер указанной переменной. Таким образом, если вы хотите определить размер какого-либо типа данных, например, **int**, вы должны поместить имя типа в круглые скобки. Если же вам нужен размер переменной, скобки не нужны, хотя при желании вы можете их использовать и здесь.

Чтобы посмотреть, как работает **sizeof**, запустите эту короткую программу. Для многих 32-разрядных сред она выведет значения 1, 4, 4 и 8:

```
// Демонстрация sizeof.

#include <iostream>
using namespace std;

int main()
{
    char ch;
    int i;

    cout << sizeof ch << ' '; // размер char
    cout << sizeof i << ' '; // размер int
```

```
cout << sizeof (float) << ' '; // размер float
cout << sizeof (double) << ' '; // размер double

return 0;
}
```

Вы можете использовать оператор **sizeof** с любыми типами данных. Если, например, применить его к массиву, он вернет число байтов в этом массиве. Рассмотрим такой фрагмент:

```
int nums[4];

cout << sizeof nums;
```

Полагая тип **int** 4-байтовым, этот фрагмент выведет на экран число 16 (т. е. 4 элемента по 4 байта).

Как уже отмечалось выше, **sizeof** является оператором времени компиляции. Вся информация, необходимая для вычисления размера переменной или типа данных, становится известна во время компиляции. Оператор **sizeof** прежде всего помогает вам создавать переносимый код, который будет зависеть от размеров типов данных C++. Не забывайте, что поскольку размеры типов C++ определяются реализацией вычислительной среды, делать заранее предположения об этих размерах в своих программах — плохой стиль.

Минутная тренировка

1. Покажите, каким образом присвоить переменным **t1**, **t2** и **t3** значение 10 в одном предложении присваивания.
2. Как можно записать по-другому такое предложение:
`x = x + 10;`
3. Оператор **sizeof** возвращает размер переменной или типа в _____.

1. `t1 = t2 = t3 = 10;`
 2. `x += 100;`
 3. байтах
-

Обзор относительных приоритетов

В табл. 7-2 приведен порядок относительных приоритетов, от высшего в низшему, для всех операторов C++. Большинство операторов выполняются слева направо. Унарные операторы, операторы присваивания и оператор **?** выполняются справа налево. Заметьте, что таблица

включает несколько операторов, с которыми вы еще не сталкивались. В основном это операторы, используемые в объектно-ориентированном программировании.

Таблица 7-2. Относительный приоритет операторов C++

| Относительный приоритет | Операторы |
|-------------------------|--|
| Высший | () [] -> :: . |
| | ! ~ ++ -- * & sizeof new delete typeid <i>приведения типов</i> |
| | . * -> * |
| | * / % |
| | + - |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | |
| | && |
| | |
| | ?: |
| | = += -= *= /= %= >= <= &= ^= = |
| Низший | |

✓ Вопросы для самопроверки

1. Покажите, как следует объявить переменную `test` типа `int`, чтобы ее нельзя было изменить в программе. Дайте ей начальное значение 100.
2. Описатель `volatile` сообщает компилятору, что переменная может быть изменена какими-либо событиями вне программы. Справедливо ли это утверждение?
3. Какой описатель вы используете в многофайловом проекте, чтобы один файл узнал о глобальных переменных, объявленных в другом файле?
4. Каково наиболее важное свойство статической локальной переменной?
5. Напишите программу, содержащую функцию с именем `counter()`, которая просто отсчитывает число своих вызовов. Функция должна возвращать текущее число вызовов.

6. Рассмотрите приведенный ниже фрагмент. Какую переменную следует объявить регистровой, чтобы получить максимальный выигрыш в скорости?

```
int myfunc()
{
    int x;
    int y;
    int z;

    z = 10;
    y = 0;

    for(x=z; x < 15; x++)
        y += x;

    return y;
}
```

7. Чем **&** отличается от **&&**?
8. Что делает это предложение?

```
x *= 10;
```

9. Используя функции **rrotate()** и **rrotate()** из Проекта 7-1, возможно шифровать и расшифровывать строку. Для шифрования строки выполните циклический сдвиг каждой буквы влево на некоторое число битов, задаваемое ключом. Для расшифровки выполните циклический сдвиг каждой буквы вправо на то же число битов. Используйте ключ, состоящий из строки символов. Можно придумать много способов вычисления числа сдвигов, исходя из содержимого ключа. Проявите творческий подход. Решение, показанное в Приложении А, является лишь одним из многих возможных.
10. Самостоятельно усовершенствуйте функцию **show_binary()**, чтобы она показывала в значении **unsigned int** все биты, а не только первые восемь.

Модуль 8 Классы и объекты

Цели, достигаемые в этом модуле

- 8.1 Познакомиться с общей формой класса
- 8.2 Приобрести навыки в создании классов и объектов
- 8.3 Освоить добавление в класс членов-функций
- 8.4 Научиться использовать конструкторы и деструкторы
- 8.5 Узнать о параметрических конструкторах
- 8.6 Освоить создание встроенных функций
- 8.7 Получить представление о массивах объектов
- 8.8 Научиться инициализировать массивы объектов
- 8.9 Изучить взаимодействие указателей и объектов

Программы, которые вы составляли до сих пор, не использовали объектно-ориентированные возможности C++. Все рассмотренные программы использовали метод структурного, а не объектно-ориентированного программирования. Для того, чтобы начать писать объектно-ориентированные программы, необходимо изучить понятие класса. Класс в C++ является базовой единицей инкапсуляции. Классы используются для создания объектов. Классы и объекты являются настолько фундаментальными понятиями для C++, что почти вся оставшаяся часть книги будет посвящена именно им.

Основы классов

Давайте начнем с обзора понятий *класса* и *объекта*. *Класс* — это шаблон, который определяет форму *объекта*. Класс описывает и коды, и данные. C++ использует описание класса для конструирования объектов. Объекты являются *экземплярами* класса. Таким образом, класс по существу представляет собой комплект планов, по которым строится объект. Важно усвоить, что класс — это логическая абстракция. Лишь после того, как будет создан объект класса, в памяти появится и начнет существовать физическое представление этого класса.

Определяя класс, вы объявляете данные, которые будут в него входить, а также код, который будет работать с этими данными. Хотя очень простые классы могут содержать только код или только данные, в большинство реальных классов входят и те, и другие. Данные содержатся в *переменных экземпляра*, определенных в классе, а код содержится в *функциях*. Код и данные, составляющие класс, называются *членами* класса.

Цель

8.1. Общая форма класса

Класс создается с помощью ключевого слова **class**. Общая форма объявления простого класса выглядит следующим образом:

```
class имя-класса {  
    закрытые данные и функции  
public:  
    открытые данные и функции  
} список объектов;
```

Здесь *имя-класса* задает имя этого класса. Это имя становится именем нового типа, который может быть использован для создания объектов этого класса. Вы можете также создать объекты класса, указав их непосредственно после объявления класса в *списке объектов*, хотя это

не обязательно. После того, как класс объявлен, объекты можно создавать по мере необходимости.

Класс может содержать как закрытые, так и открытые члены. По умолчанию все члены, определенные в классе, являются закрытыми. Это означает, что к ним можно обращаться только из других членов этого класса, но не из других частей вашей программы. Закрытость членов класса является одним из способов достижения инкапсуляции — вы можете жестко контролировать доступ к определенным данным, объявив их закрытыми.

Чтобы сделать часть членов класса открытыми (т. е. разрешить доступ к ним из других частей вашей программы), вы должны объявить их после ключевого слова **public**. Все переменные и функции, определенные после описателя **public**, доступны из других частей вашей программы. Обратите внимание на то, что после описателя **public** стоит двоеточие.

Хотя нет определенных синтаксических правил, определяющих состав класса, однако разумно разработанный класс должен определять одну и только одну логическую сущность. Например, класс, содержащий имена и телефонные номера, не должен, как правило, включать в себя данные о котировках фондовой биржи, ежемесячном количестве осадков, солнечных пятнах и другую не относящуюся в делу информацию. Помещение в класс чуждой для него информации быстро запутает вашу программу!

Подытожим сказанное. В C++ класс определяет новый тип данных, который можно использовать для создания объектов. Класс представляет собой логический шаблон, определяющий взаимоотношения между его членами. Объявляя переменную класса, вы создаете объект. Объект имеет физическое воплощение и является конкретным экземпляром класса. Другими словами, объект занимает место в памяти, в то время как определение типа в памяти не существует.

Цель

8.2. Определение класса и создание объектов

Для иллюстрации техники работы с классами мы разработаем класс, который инкапсулирует информацию о транспортных средствах, конкретно, о некоторых типах автомобилей: легковых автомобилях, фургонах и грузовиках. Назовем наш класс **Vehicle**, и пусть в нем будет содержаться три единицы информации об автомобилях: допустимое число пассажиров, объем бензобака и среднее потребление топлива (в милях на галлон горючего).

Ниже показан первый вариант класса **Vehicle**. Он определяет три переменных экземпляра: **passengers**, **fuelcap** и **mpg**. Заметьте, что **Vehicle** не содержит никаких функций; пока что он является классом, включающим только данные. (В последующих подразделах мы добавив в него и функции.)

```
class Vehicle {      // класс транспортных средств
public:
    int passengers;   // число пассажиров
    int fuelcap;      // емкость бензобака в галлонах
    int mpg;          // потребление топлива в милях на галлон
};
```

Переменные экземпляра, определенные в **Vehicle**, иллюстрируют общий принцип объявления переменных. Общая форма для объявления переменной экземпляра выглядит так:

тип имя-переменной;

Здесь *тип* определяет тип переменной, а *имя-переменной* — ее имя. Таким образом, вы объявляете переменные экземпляра точно так же, как вы это делали для обычных переменных. В классе **Vehicle** перед перечнем переменных указан описатель доступа **public**. Как уже говорилось, это дает возможность обращаться к переменным из кода вне **Vehicle**.

Определение класса создает новый тип данных. В нашем случае этот новый тип имеет имя **Vehicle**. Вы будете использовать это имя для объявления объектов типа **Vehicle**. Не забывайте, что объявление класса представляет собой лишь описание типа; оно не создает реальных объектов. Таким образом, приведенный выше фрагмент программы не приводит к появлению каких-либо объектов типа **Vehicle**.

Для того, чтобы создать реальный объект **Vehicle**, просто используйте предложения объявления, например, так:

```
Vehicle minivan; // создание объекта Vehicle с именем minivan
```

После выполнения этого предложения **minivan** будет экземпляром **Vehicle**. Тем самым мы получаем “физическую” реальность.

Каждый раз, когда вы создаете экземпляр класса, вы создаете объект, содержащий свою собственную копию всех переменных экземпляра, определенных в классе. Так, каждый объект **Vehicle** будет содержать свои собственные копии переменных **passengers**, **fuelcap** и **mpg**. Для доступа к этим переменным мы будем использовать оператор-точку (.). Оператор-точка соединяет имя объекта с именем его члена. Общая форма оператора-точки такова:

объект.член

Мы видим, что объект указывается с левой стороны от точки, а член — с правой стороны. Например, чтобы присвоить переменной **fuelcap** объекта **minivan** значение 16, следует использовать такое предложение:

```
minivan.fuelcap = 16;
```

В общем случае оператор-точка используется как для обращения к переменным экземпляра, так и для вызова функций.

Вот полная программа, использующая класс **Vehicle**:

```
// Программа, использующая класс Vehicle.

#include <iostream>
using namespace std;

// Объявим класс Vehicle.
class Vehicle { ←————— Объявим класс Vehicle.
public:
    int passengers; // число пассажиров
    int fuelcap;     // емкость бензобака в галлонах
    int mpg;         // потребление топлива в милях на галлон
};

int main() {
    Vehicle minivan; // создадим объект класса Vehicle ←
    int range;

    // Присвоим значения полям объекта minivan (фургон).
    minivan.passengers = 7;
    minivan.fuelcap = 16; ←————— Обратите внимание на использование
    minivan.mpg = 21;          оператора-точки при обращении к членам.
    // Вычислим дальность пробега для полного бензобака.
    range = minivan.fuelcap * minivan.mpg;

    cout << "Фургон может возить " << minivan.passengers <<
         " на расстояние " << range << "\n";

    return 0;
}
```

Рассмотрим текст программы. Функция **main()** создает экземпляр класса **Vehicle** с именем **minivan**. Код внутри **main()** обращается к переменным экземпляра, относящимся к объекту **minivan**, присваивает им значения и затем использует эти значения. Код внутри **main()** имеет доступ к членам **Vehicle**, так как они объявлены с описателем **public**. Если бы этот описатель не был указан, обращаться к членам **Vehicle** можно было бы только из этого класса, и **main()** не смогла бы их использовать.

Запустив программу, вы получите следующий вывод:

Фургон может возить 7 на расстояние 336

Перед тем, как пойти дальше, еще раз остановимся на фундаментальном принципе: каждый объект имеет собственные копии пере-

менных экземпляра, определенных в его классе. Поэтому содержимое переменных одного объекта может отличаться от содержимого переменных другого объекта. Между двумя объектами нет никакой связи, за исключением того факта, что оба объекта принадлежат одному типу. Например, если у вас есть два объекта **Vehicle**, в каждом есть собственные копии **passengers**, **fuelcap** и **mpg**, причем содержимое этих переменных для двух объектов может различаться. Приведенная ниже программа демонстрирует это положение:

```
// Эта программа создает два объекта Vehicle.

#include <iostream>
using namespace std;

// Объявим класс Vehicle
class Vehicle {
public:
    int passengers; // число пассажиров
    int fuelcap;    // емкость бензобака в галлонах
    int mpg;        // потребление топлива в милях на галлон
};
// Каждый объект, и minivan, и sportscar, имеют собственные копии
// переменных экземпляра класса Vehicle.

int main() {
    Vehicle minivan;    // создадим объект класса Vehicle
    Vehicle sportscar;  // создадим другой объект

    int range1, range2;
    // Присвоим значения полям minivan (фургон).
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Присвоим значения полям sportscar (спортивный автомобиль).
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычислим дальность пробега для полного бензобака.
    range1 = minivan.fuelcap * minivan.mpg;
    range2 = sportscar.fuelcap * sportscar.mpg;

    cout << "Фургон может везти " << minivan.passengers <<
         " на расстояние " << range1 << "\n";

    cout << "Спортивный автомобиль может везти " <<
         sportscar.passengers <<
```

```
    " на расстояние " << range2 << "\n";  
  
    return 0;  
}
```

Легко видеть, что данные по спортивному автомобилю (объект **sportscar**) никак не связаны с данными по фургону (объект **minivan**). Рис. 8-1 отображает эту ситуацию.

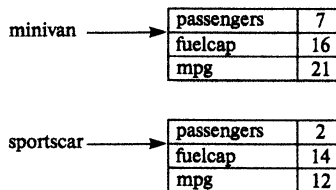


Рис. 8-1. Переменные экземпляра двух объектов никак не связаны друг с другом

Минутная тренировка

1. Какие две составляющие входят в класс?
 2. Какой оператор используется для доступа к членам класса посредством объекта?
 3. Каждый объект имеет собственные копии _____ класса.
 1. Класс может содержать коды и данные.
 2. Для доступа к членам класса посредством объекта используется оператор-точка.
 3. переменных экземпляра
-

Цель

8.3. Добавление в класс функций-членов

Пока что класс **Vehicle** содержит только данные, в нет функций. Хотя классы без функций вполне имеют право на существование, в действительности большинство классов имеют функции-члены. Как правило, функции-члены манипулируют с данными, определенными в классе, и во многих случаях предоставляют доступ к этим данным. Обычно другие части вашей программы взаимодействуют с классом посредством его функций.

Чтобы продемонстрировать пример определения и использования функций-членов, мы добавим одну такую функцию в класс **Vehicle**. Вспомним, что `main()` в последнем примере вычисляла дальность пробега транспортного средства путем умножения потребления топлива на емкость бензобака. Хотя формально такой подход правилен, это не

лучший способ работы с данными класса. Вычисление дальности пробега транспортного средства — это операция, которую лучше поручить самому классу **Vehicle**. Основание для такого подхода очевидно: дальность пробега транспортного средства зависит от емкости бензобака и потребления топлива, а обе эти величины инкапсулированы в **Vehicle**. Добавив в класс **Vehicle** функцию, вычисляющую дальность пробега, вы улучшите его объектно-ориентированную структуру.

Для добавления функции в **Vehicle**, определите ее прототип в объявлении **Vehicle**. Например, в приведенном ниже варианте **Vehicle** определена функция-член с именем **range()**, возвращающая дальность пробега транспортного средства:

```
// Объявим класс Vehicle.
class Vehicle {
public:
    int passengers; // число пассажиров
    int fuelcap;     // емкость бензобака в галлонах
    int mpg;         // потребление топлива в милях на галлон

    int range(); // вычислить и вернуть дальность пробега ←
};
```

Объявление функции-члена **range()**

Поскольку прототипы функций-членов, в частности, **range()**, входят в определение класса, их больше нигде описывать не надо.

Для реализации функции-члена вы должны сообщить компилятору, какому классу она принадлежит. Это делается путем добавления к имени функции имени ее класса. Ниже приведен один из способов реализации функции **range()**:

```
// Реализация функции-члена range.
int Vehicle::range() {
    return mpg * fuelcap;
}
```

Обратите внимание на знак двойного двоеточия **::**, разделяющий имя класса **Vehicle** и имя функции **range()**. Знак **::** называется *оператором разрешения видимости*. Этот оператор связывает имя класса с именем члена и тем самым сообщает компилятору, какому классу принадлежит этот член. В нашем случае он связывает **range()** с классом **Vehicle**. Другими словами, знак **::** устанавливает, что **range()** находится в области видимости **Vehicle**. Различные классы вполне могут иметь функции с одинаковыми именами. Компилятор определяет, какая функция принадлежит какому классу, благодаря оператору разрешения видимости и имени класса.

Тело функции **range()** состоит из единственной строки:

```
return mpg * fuelcap;
```

Это предложение возвращает дальность пробега транспортного средства, которое определяется как произведение **fuelcap** на **mpg**. Поскольку каждый объект класса **Vehicle** имеет собственную копию **fuelcap** и **mpg**, вычисление в **range()** использует копии этих переменных, принадлежащие вызывающему объекту.

Внутри **range()** обращение к переменным экземпляра **fuelcap** и **mpg** осуществляется непосредственно, без предварения их именем объекта и оператором-точкой. Когда функция-член использует переменную экземпляра, определенную в ее классе, она обращается к ней непосредственно, без явного указания имени объекта и без использования оператора-точки. Легко понять, почему это так. Функция-член всегда вызывается относительно некоторого объекта своего класса. Когда функция начинает выполняться, объект уже известен. Поэтому внутри функции-члена нет необходимости второй раз специфицировать объект. Это значит, что **fuelcap** и **mpg** внутри **range()** неявным образом ссылаются на копии переменных, входящих в тот объект, который активизировал **range()**. Разумеется, код вне **Vehicle** должен обращаться к **fuelcap** и **mpg**, только ссылаясь на конкретный объект и используя оператор-точку.

Функцию-член можно вызывать только относительно конкретного объекта. Этот вызов может осуществляться двумя способами. Во-первых, функция-член может быть вызвана кодом вне ее класса. В этом случае вы должны указать имя объекта и оператор-точку. Например, в следующем предложении функция **range()** вызывается для объекта **minivan**:

```
range = minivan.range();
```

Активизация функции-члена **minivan.range()** заставляет **range()** работать с копией переменных экземпляра, принадлежащих **minivan**. Поэтому функция возвращает дальность пробега для **minivan**.

Во втором случае функция-член может быть вызвана из другой функции-члена того же класса. Когда одна функция-член вызывает другую функцию-член того же класса, она может сделать это непосредственно, без использования оператора-точки. В этом случае компилятор уже знает, над каким объектом осуществляются действия. Только когда функция-член вызывается кодом, не принадлежащим классу, необходимо указывать имя объекта и оператор-точку.

Приведенная ниже программа собирает вместе все рассмотренные фрагменты и иллюстрирует использование функции **range()**:

```
// Программа, использующая класс Vehicle.

#include <iostream>
using namespace std;

// Объявим класс Vehicle.
class Vehicle {
```



```

public:
    int passengers; // число пассажиров
    int fuelcap;    // емкость бензобака в галлонах
    int mpg;        // потребление топлива в милях на галлон

    int range(); // вычислить и вернуть дальность пробега ←
};                                                       Объявление range( )

// Реализация функции-члена range.
int Vehicle::range() { ← Реализация range( ).
    return mpg * fuelcap;
}

int main() {
    Vehicle minivan;      // создадим объект класса Vehicle
    Vehicle sportscar;    // создадим другой объект

    int range1, range2;

    // Присвоим значения полям minivan.
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // Присвоим значения полям sportscar.
    sportscar.passengers = 2;
    sportscar.fuelcap = 14;
    sportscar.mpg = 12;

    // Вычислим дальность пробега для полного бензобака.
    range1 = minivan.range(); ← Вызов range( ) для объектов Vehicle
    range2 = sportscar.range();

    cout << " Фургон может везти " << minivan.passengers <<
        " на расстояние " << range1 << "\n";

    cout << " Спортивный автомобиль может везти " <<
        sportscar.passengers <<
        " на расстояние " << range2 << "\n";

    return 0;
}

```

Эта программа выводит следующее:

Фургон может везти 7 на расстояние 336

Спортивный автомобиль может везти 2 на расстояние 168

Минутная тренировка

1. Как называется оператор ::?
 2. Каково назначение оператора ::?
 3. Если функция-член вызывается извне своего класса, она должна вызываться с указанием объекта и с использованием оператора-точки. Это правильно или нет?
1. Оператор `·` называется оператором разрешения видимости
 2. Оператор `·` связывает класс с членом.
 3. Правильно. При вызове извне своего класса функция-член должна вызываться с указанием объекта и с использованием оператора-точки.
-

Проект 8-1

Создание класса справочника

Если бы кто-то попытался обобщить сущность класса в одном предложении, оно бы выглядело примерно так: класс инкапсулирует функциональные возможности. Конечно, часто сразу не определишь, где кончатся одни “функциональные возможности” и начнутся другие. Как правило, вы хотите, чтобы ваши классы служили строительными блоками вашего большого приложения. Чтобы удовлетворять этому требованию, каждый класс должен представлять единую функциональную единицу, выполняющую четко разграниченные функции. Получается, что вы должны разработать свои классы так, чтобы они были такими маленькими, как это только возможно — но не меньше! Действительно, классы, содержащие избыточные функциональные возможности, будут запутывать код и нарушать его структурную строгость, но классы, содержащие слишком мало функциональных возможностей, создадут фрагментарность. Где золотая середина? Именно в этом месте *наука* программирования переходит в *искусство* программирования. К счастью, большинство программистов быстро обнаруживает, что поиски золотой середины становятся все менее обременительными по мере накопления опыта работы.

Начните приобретать этот опыт с преобразования справочной системы из Проекта 3-3 в Модуле 3 в класс справочника. Посмотрим, почему такое преобразование имеет смысл. Во-первых, справочная система определяет одну логическую единицу. Она просто выводит на экран синтаксис управляющих предложений C++. Функциональные возможности этой системы компактны и четко определены. Во-вторых, помещение справочника в класс — эстетически красивый подход. Желая предложить справочную систему пользователю, вы просто создаете экземпляр объекта этой системы. Наконец, поскольку справочник инкапсулирован, он может расширяться и изменяться без нежелательных побочных эффектов в использующей его программе.

Шаг за шагом

1. Создайте новый файл с именем **HelpClass.cpp**. Чтобы не тратить время на ввод текста с клавиатуры, вы можете скопировать файл **Help3.cpp** из Проекта 3-3 в **HelpClass.cpp**.

2. Для преобразования справочной системы в класс вы должны прежде всего точно определить, что именно составляет эту систему. Например, в **Help3.cpp** имеется код, выводящий меню, вводящий запрос пользователя, проверяющий его правильность и выводящий на экран информацию о выбранном пункте. В программе также предусмотрено циклическое выполнение до нажатия клавиши q. Легко сообразить, что меню, проверка правильности запроса и вывод содержательной информации являются существом справочной системы. А вот каким образом получается запрос пользователя, как и циклическое повторение программы, прямого отношения к справочной системе не имеет. Таким образом, вам надо создать класс, который выводит справочную информацию и меню справочника, а также проверяет правильность запроса пользователя. Эти функции мы назовем соответственно **helpon()**, **showmenu()** и **isvalid()**.

3. Объявите класс **Help**, как это показано ниже:

```
// Класс, инкапсулирующий справочную систему.
class Help {
public:
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};
```

Обратите внимание на то, что это чисто функциональный класс; в нем не требуются переменные экземпляра. Как уже отмечалось ранее, класс только с данными или только с функциями вполне допустимы. (Вопрос 9 в Вопросах для самопроверки добавляет в класс **Help** переменную экземпляра.)

4. Создайте функцию **helpon()**, как это показано ниже:

```
// Вывод справочной информации.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Предложение if:\n\n";
            cout << "if(условие) предложение;\n";
            cout << "else предложение;\n";
            break;
        case '2':
            cout << "The switch:\n\n";
```

```
    cout << "switch(выражение) {\n";
    cout << " case константа:\n";
    cout << " последовательность предложений\n";
    cout << " break;\n";
    cout << " // ... \n";
    cout << " }\n";
    break;
case '3':
    cout << "Цикл for:\n\n";
    cout << "for(инициализация; условие; приращение) ";
    cout << " предложение;\n";
    break;
case '4':
    cout << "Цикл while:\n\n";
    cout << "while(условие) продолжение;\n";
    break;
case '5':
    cout << "Цикл do-while:\n\n";
    cout << "do {\n";
    cout << " предложение;\n";
    cout << "} while (условие);\n";
    break;
case '6':
    cout << "Предложение break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "Предложение continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "Предложение goto:\n\n";
    cout << "goto метка;\n";
    break;
}
cout << "\n";
}
```

5. Создайте функцию `showmenu()` :

```
// Вывод на экран меню справочной системы.
void Help::showmenu() {
    cout << "Справка по:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
}
```

```

cout << " 5. do-while\n";
cout << " 6. break\n";
cout << " 7. continue\n";
cout << " 8. goto\n";
cout << "Выберите один из пунктов (q для завершения): ";
}

```

6. Создайте функцию **isvald()**, приведенную ниже:

```

// Возвращает true, если выбор допустим.
bool Help::isvalid(char ch) {
    if(ch < '1' || ch > '8' && ch != 'q')
        return false;
    else
        return true;
}

```

7. Перепишите функцию **main()** из Проекта 3-3, чтобы она использовала новый класс **Help**. Ниже приведен полный текст **HelpClass.cpp**:

```

/*
    Проект 8-1
    Преобразование справочной системы из Проекта 3-3
    в класс Help.
*/
#include <iostream>
using namespace std;

// Класс, инкапсулирующий справочную систему.
class Help {
public:
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};

// Вывод справочной информации.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Предложение if:\n\n";
            cout << "if(условие) предложение;\n";
            cout << "else предложение;\n";
            break;
        case '2':
            cout << "Предложение switch:\n\n";
            cout << "switch(выражение) {\n";

```

```
    cout << " case константа:\n";
    cout << " последовательность предложений\n";
    cout << " break;\n";
    cout << " // ... \n";
    cout << "}\n";
    break;
case '3':
    cout << "Цикл for:\n\n";
    cout << "for(инициализация; условие; приращение) ";
    cout << " предложение;\n";
    break;
case '4':
    cout << "Цикл while:\n\n";
    cout << "while(условие) продолжение;\n";
    break;
case '5':
    cout << "Цикл do-while:\n\n";
    cout << "do {\n";
    cout << " предложение;\n";
    cout << "} while (условие);\n";
    break;
case '6':
    cout << "Предложение break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "Предложение continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "Предложение goto:\n\n";
    cout << "goto метка;\n";
    break;
}
cout << "\n";
}

// Вывод на экран меню справочной системы.
void Help::showmenu() {
    cout << "Справка по:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << " 6. break\n";
}
```

```

    cout << " 7. continue\n";
    cout << " 8. goto\n";
    cout << "Выберите один из пунктов (q для завершения): ";
}
// Возвращает true, если выбор допустим.
bool Help::isvalid(char ch) {
    if(ch < '1' || ch > '8' && ch != 'q')
        return false;
    else
        return true;
}

int main()
{
    char choice;
    Help hlpob; // создание экземпляра класса Help.

    // Используем объект Help для вывода информации.
    for(;;) {
        do {
            hlpob.showmenu();
            cin >> choice;
        } while(!hlpob.isvalid(choice));

        if(choice == 'q') break;
        cout << "\n";

        hlpob.helpon(choice);
    }

    return 0;
}

```

Когда вы испытаете эту программу, вы обнаружите, что функционально она не отличается от своей предшественницы из Модуля 3. Преимущество нового подхода в том, что теперь у вас есть компонент справочной системы, который можно повторно использовать, где бы он ни понадобился.

Цель

8.4. Конструкторы и деструкторы

В предыдущих примерах переменные экземпляра в каждом объекте `Vehicle` должны были устанавливаться вручную с помощью предложений, подобных этим:

```
minivan.passengers = 7;  
minivan.fuelcap = 16;  
minivan.mpg = 21;
```

В профессионально написанном коде на C++ такой подход никогда не применяется. Помимо возможных ошибок (вы можете позабыть установить одно из полей), просто существует более удобный способ выполнить задачу инициализации. Эта задача возлагается на конструктор.

Конструктор инициализирует объект при его создании. Конструктор имеет то же имя, что и класс, а синтаксически сходен с функцией. Однако конструкторы не возвращают никакого значения. Общая форма конструктора выглядит так:

```
имя-класса( ) {  
    //код конструктора  
}
```

Обычно конструктор используется для придания переменным экземпляра, определенным в классе, начальных значений, а также для выполнения других начальных процедур, требуемых для создания полностью сформированного объекта.

В некотором роде противоположностью конструктору является *деструктор*. Во многих случаях объекту при его удалении требуется выполнение некоторого действия или последовательности действий. Локальные объекты создаются при входе в свой блок и уничтожаются при выходе из блока. Глобальные объекты уничтожаются, когда завершается программа. Может быть много причин, приводящим к необходимости иметь деструктор. Например, объекту может понадобиться освободить ранее выделенную память, или закрыть открытый им файл. В C++ такого рода операции выполняет деструктор. Деструктор имеет то же имя, что и конструктор, но перед именем указывается знак ~. Как и конструкторы, деструкторы не имеют возвращаемого значения.

Вот простой пример использования конструктора и деструктора:

```
// Простой конструктор и деструктор.  
  
#include <iostream>  
using namespace std;  
  
class MyClass {  
public:  
    int x;  
  
    // Объявим конструктор и деструктор.  
    MyClass(); // конструктор  
    ~MyClass(); // деструктор  
};
```

Объявим конструктор и деструктор для класса **MyClass**


```
к// Реализация конструктора MyClass.
MyClass::MyClass() {
    x = 10;
}
// Реализация деструктора MyClass.
MyClass::~MyClass() {
    cout << "Уничтожаем...\n";
}

int main() {
    MyClass ob1;
    MyClass ob2;

    cout << ob1.x << " " << ob2.x << "\n";

    return 0;
}
```

Вывод этой программы выглядит таким образом:

```
10 10
Уничтожаем...
Уничтожаем...
```

В этом примере конструктор класса **MyClass** имеет такой вид:

```
// Реализация конструктора MyClass.
MyClass::MyClass() {
    x = 10;
}
```

Заметьте, что конструктор объявлен после описателя **public**. Это естественно, так как конструктор будет вызываться из кода вне его класса. Этот конструктор присваивает переменной экземпляра **x** из **MyClass** значение 10. Конструктор вызывается, когда создается объект. Например, в строке

```
MyClass ob1;
```

конструктор **MyClass ()** вызывается для объекта **ob1**, присваивая **ob1.x** значение 10. То же справедливо для объекта **ob2**. После конструирования **ob2.x** так же будет иметь значение 10.

Деструктор для **MyClass** показан ниже:

```
// Реализация деструктора MyClass.
MyClass::~MyClass() {
    cout << "Уничтожаем...\n";
}
```

Этот деструктор просто выводит сообщение, но в реальных программах деструктор используется для освобождения ресурсов (например, дескриптора файла или памяти), используемых классом.

Минутная тренировка

1. Что такое конструктор и когда он выполняется?
 2. Имеет ли конструктор возвращаемое значение?
 3. Когда вызывается деструктор?
- 1 Конструктор — это функция, которая выполняется, когда создается экземпляр объекта данного класса. Конструктор используется для инициализации создаваемого объекта
 - 2 Нет, конструктор не имеет возвращаемого значения
 - 3 Деструктор вызывается при уничтожении объекта

Цель

8.5. Параметрические конструкторы

В предыдущем примере использовался конструктор без параметров. Хотя в некоторых ситуациях именно такой конструктор и нужен, однако чаще нам требуется конструктор с одним или несколькими параметрами. Параметры добавляются в конструктор точно так же, как они добавляются в функцию: просто объявите их внутри круглых скобок после имени конструктора. Вот, например, параметрический конструктор для класса **MyClass**:

```
MyClass::MyClass(int i) {
    x = i;
}
```

Для передачи конструктору аргумента вы должны сопоставить передаваемое значение или значения с объявляемым объектом. C++ предоставляет для этого два способа. Первый способ выглядит так:

```
MyClass ob1 = MyClass(101);
```

В этом объявлении создается объект класса **MyClass** с именем **ob1**, и этому объекту передается значение 101. Однако такая форма используется редко (во всяком случае, в этом контексте), потому что второй способ короче и нагляднее. При использовании второй формы аргумент или аргументы, заключенные в круглые скобки, указываются вслед за именем объекта. Приведенное ниже предложение выполняет то же, что и предыдущее:

```
MyClass obl(101);
```

Это наиболее распространенный способ объявления параметрического объекта. Общая форма его такова:

тип-класса переменная(список-аргументов);

Здесь *список-аргументов* представляет собой перечень аргументов, передаваемых конструктору. Аргументы разделяются запятыми.



Замечание

С формальной точки зрения между двумя приведенными формами имеется небольшое различие, о котором вы узнаете позже. Это различие, впрочем, не будет влиять на программы в этом модуле, да и вообще на большинство программ, которые вам придется писать.

Вот полная программа, демонстрирующая параметрический конструктор **MyClass**:

```
// Параметрический конструктор.

#include <iostream>
using namespace std;


class MyClass {
public:
    int x;
    // Объявим конструктор и деструктор.
    MyClass(int i); // конструктор ← Добавим параметр к MyClass( ).
    ~MyClass(); // деструктор
};

// Реализация параметрического конструктора.
MyClass::MyClass(int i) {
    x = i;
}

// Реализация деструктора MyClass.
MyClass::~MyClass() {
    cout << "Уничтожение объекта, у которого значение x равно "
         << x << " \n";
}

int main() {
```

```

MyClass t1(5); 
MyClass t2(19);

cout << t1.x << " " << t2.x << "\n";

return 0;
}

```

Вывод этой программы будет таким:

```

5 19
Уничтожение объекта, у которого значение x равно 19
Уничтожение объекта, у которого значение x равно 5

```

В этом варианте программы конструктор **MyClass()** определяет один параметр с именем **i**, который используется для инициализации переменной экземпляра **x**. Таким образом, когда выполняется строка

```
MyClass ob1(5);
```

значение 5 передается в **i**, а затем присваивается переменной **x**.

В отличие от конструкторов, деструкторы не могут иметь параметров. Причину этого легко понять: не существует способа передачи параметров уничтожаемому объекту. Если вашему объекту при вызове его деструктора требуется обращение к каким-либо данным, определяемым по ходу программы, вам придется создать для этого специальную переменную. Тогда перед самым уничтожением объекта вы сможете обратиться к этой переменной. Впрочем, такая ситуация возникает редко.

Добавление конструктора в класс **Vehicle**

Мы можем улучшить класс **Vehicle**, добавив в него конструктор, который будет автоматически инициализировать поля **passenger**, **fuelcap** и **mpg** в процессе конструирования объекта. Обратите особое внимание, как создаются объекты **Vehicle**.

// Добавление конструктора в класс транспортных средств.

```

#include <iostream>
using namespace std;

// Объявим класс Vehicle.
class Vehicle {
public:
    int passengers; // число пассажиров
    int fuelcap;    // емкость бензобака в галлонах
    int mpg;        // потребление топлива в милях на галлон

```

```

// Это конструктор для Vehicle.
Vehicle(int p, int f, int m);
    int range(); // вычислить и вернуть дальность пробега
};

// Реализация конструктора Vehicle.
Vehicle::Vehicle(int p, int f, int m) {
    passengers = p;
    fuelcap = f;
    mpg = m;
}

// Реализация функции-члена range.
int Vehicle::range() {
    return mpg * fuelcap;
}

int main() {
    // Передадим значения конструктору Vehicle.
    Vehicle minivan(7, 16, 21);
    Vehicle sportscar(2, 14, 12);

    int range1, range2;

    // Вычислим дальность пробега для полного бензобака.
    range1 = minivan.range();
    range2 = sportscar.range();

    cout << "Фургон может возить " << minivan.passengers <<
        " на расстояние " << range1 << "\n";
    cout << "Спортивный автомобиль может возить " <<
        sportscar.passengers <<
        " на расстояние " << range2 << "\n";

    return 0;
}

```

Конструктор **Vehicle** инициализирует **passengers**, **fuelcap** и **mpg**

Передача информации о транспортном средстве классу **Vehicle** посредством его конструктора

Оба объекта, и **minivan**, и **sportscar**, инициализируются конструктором при их создании. Каждый объект инициализируется параметрами, указанными при вызове его конструктора. Например, в строке

```
Vehicle minivan(7, 16, 21);
```

значения 7, 16 и 21 передаются конструктору **Vehicle**, когда создается объект **minivan**. В результате копии переменных **passengers**, **fuelcap** и **mpg**, принадлежащие объекту **minivan**, будут содержать значения 7, 16 и 21 соответственно. Вывод этой программы будет, разумеется, тем же что и у предыдущего варианта.

Другой способ инициализации

Если конструктор принимает только один параметр, вы можете использовать для инициализации объекта альтернативный способ. Рассмотрим такую программу:

```
// Альтернативный способ инициализации.

#include <iostream>
using namespace std;


class MyClass {
public:
    int x;
    // Объявим конструктор и деструктор.
    MyClass(int i); // конструктор
    ~MyClass(); // деструктор
};

// Реализация параметрического конструктора.
MyClass::MyClass(int i) {
    x = i;
}

// Реализация деструктора MyClass.
MyClass::~~MyClass() {
    cout << "Уничтожение объекта, у которого значение x равно "
         << x << " \n";
}

int main() {
    MyClass ob = 5; // вызов MyClass(5)
    cout << ob.x << " \n";

    return 0;
}
```



Здесь конструктор класса **MyClass** принимает один параметр. Обратите особое внимание на способ объявления **ob** в **main()**:

```
MyClass ob = 5;
```

При такой форме инициализации 5 автоматически передается параметру **i** в конструкторе **MyClass**. Предложение инициализации воспринимается компилятором, как если бы оно было написано таким образом:

```
MyClass ob = MyClass(5);
```

Всегда, если у вас есть конструктор, требующий только один аргумент, вы можете использовать для инициализации объекта либо *ob(x)*, либо *ob=x*. Объясняется это просто: если вы создаете конструктор с одним аргументом, вы также неявно создаете преобразование типа этого аргумента в тип класса.

Не забывайте, что описанный здесь альтернативный способ годится только для конструкторов, имеющих в точности один аргумент.

Минутная тренировка

1. Покажите, как объявить конструктор для класса `Test`, который принимает один параметр типа `int` с именем `count`.
2. Приведите другую форму написания этого предложения:

```
Test ob = Test(10);
```

3. Как еще можно объявить объект из предыдущего вопроса?

1. `Test::Test(int count) {...`
 2. `Test ob(10);`
 3. `Test ob = 10;`
-

Спросим у эксперта

Вопрос: Можно ли объявить один класс в другом? Другими словами, могут ли классы быть вложенными?

Ответ: Да, вполне возможно объявить один класс внутри другого. Такая операция создает *вложенные классы*. Поскольку объявление `class` фактически создает область видимости, вложенный класс действителен только в области видимости внешнего класса. Честно говоря, благодаря богатству и гибкости других средств C++, например, наследования, о котором речь будет идти ниже, нужда во вложенных классах возникает крайне редко.

Цель

8.6. Встроенные функции

Перед тем, как мы продолжим исследование понятия класса, нам следует сделать небольшое, но важное отступление. В C++ имеется одно чрезвычайно полезное средство, называемое *встроенными функциями*, которое, хотя и не относится исключительно к объектно-ориентированному программированию, часто используется в объявлениях классов. Встроенная функция — это функция, которая фактически не вызывается, а развертывается в свой полный текст в месте своего вызова. Встроенную функцию можно создать двумя способами. Первый

заключается в использовании описателя **inline**. Например, для создания встроенной функции с именем **f**, которая возвращает **int** и не требует параметров, вы объявляете ее следующим образом:

```
inline int f()
{
    // ...
}
```

Описатель **inline** указывается перед всеми остальными элементами объявления функции.

Ценность встроенных функций заключается в их эффективности. Каждый раз при вызове обычной функции должна выполняться последовательность команд, как собственно для ее вызова, включая проталкивание аргументов функции в стек, так и для возврата из функции. В ряде случаев для выполнения этих операций требуется большое количество тактов процессора. Если же функция развертывается еще на этапе компиляции в свой полный текст, включаемый в текст основной программы, то никаких издержек не возникает, что повышает скорость выполнения вашей программы. В то же время, если текст встроенной функции велик, общий размер вашей программы также может существенно увеличиться. Поэтому лучше всего объявлять встроенными небольшие по размеру функции. Большие же функции следует, как правило, объявлять обычным образом.

Приведенная ниже программа демонстрирует использование описателя **inline**:

```
// Демонстрация описателя inline.

#include <iostream>
using namespace std;

class cl {
    int i; // закрыта 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) ←
{
    ←
}
```

Функции **get_i()** и **put_i()**
являются встроенными


```

    i = j;
}

int main()
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

Важно отдавать себе отчет в том, что, строго говоря, описатель **inline** является лишь *запросом*, а не *командой*, чтобы компилятор сгенерировал встроенный код. Имеется ряд ситуаций, в которых компилятор не сможет выполнить запрос на создание встроенной функции. Вот несколько примеров таких ситуаций:

- Некоторые компиляторы не будут компилировать встроенный код для функции, содержащей циклы, а также предложения **switch** и **goto**.
- Часто оказывается невозможным создать встроенную рекурсивную функцию.
- Могут быть запрещены встроенные функции, содержащие статические переменные.

Не забывайте, что ограничения на создание встроенных функций зависят от реализации вычислительной системы, поэтому для выяснения применимости тех или иных ограничений в вашей ситуации вы должны свериться с документацией к вашему компилятору.

Создание встроенных функций внутри класса

Другой способ создания встроенной функции заключается в определении кода функции-члена *внутри* определения класса. Любая функция, определенная внутри определения класса, автоматически делается встроенной. В этом случае нет необходимости в использовании ключевого слова **inline**. Например, предыдущую программу можно переписать следующим образом:

```

#include <iostream>
using namespace std;

class cl {
    int i; // закрыта по умолчанию
public:
    // Автоматическое создание встроенных функций.

```

```
int get_i() { return i; } ← Определение get_i() и put_i()  
void put_i(int j) { i = j; } ← внутри класса  
};  
  
int main()  
{  
    cl s;  
  
    s.put_i(10);  
    cout << s.get_i();  
  
    return 0;  
}
```

Обратите внимание на то, как записаны тексты функций. Для очень коротких функций такое расположение отражает общепринятый стиль C++. Однако вы могли написать их и таким образом:

```
class cl {  
int i; // закрыта по умолчанию  
public:  
    // встроенные функции  
    int get_i()  
    {  
        return i;  
    }  
  
    void put_i(int j)  
    {  
        i = j;  
    }  
};
```

Короткие функции вроде тех, что использованы в нашем примере, обычно определяются внутри объявления класса. Такие встроенные функции широко используются в объектно-ориентированном программировании, где часто открытая функция предоставляет доступ к закрытой переменной. Такие функции носят название *функций доступа*. Одним из условий успешного объектно-ориентированного программирования является контроль доступа к данным посредством функций-членов. Поскольку большинство программистов на C++ определяют функции доступа и другие короткие функции-члены внутри их классов, такой стиль будет использоваться в дальнейших примерах этой книги. Вам также следует привыкнуть к этому стилю.

В приведенной ниже программе класс **Vehicle** записан так, что его конструктор, деструктор и функция **range()** определены внутри класса.

Далее, поля **passenger**, **fuelcap** и **mpg** сделаны закрытыми, и для получения их значений добавлены функции доступа:

```
// Определяем конструктор, деструктор
// и функцию range(), как встроенные.

#include <iostream>
using namespace std;

// Объявляем класс Vehicle.
class Vehicle {
    // Эти члены теперь закрыты.
    int passengers; // число пассажиров
    int fuelcap;    // емкость бензобака в галлонах
    int mpg;        // потребление топлива в милях на галлон
public:
    // Это конструктор для Vehicle.
    Vehicle(int p, int f, int m) {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }

    // Вычислим и вернем дальность пробега.
    int range() { return mpg * fuelcap; }
    // Функции доступа.
    int get_passengers() { return passengers; }
    int get_fuelcap() { return fuelcap; }
    int get_mpg() { return mpg; }
};

int main() {
    // Передадим значения конструктору Vehicle.
    Vehicle minivan(7, 16, 21);
    Vehicle sportscar(2, 14, 12);

    int range1, range2;

    // Вычислим дальность пробега при полном бензобаке.
    range1 = minivan.range();
    range2 = sportscar.range();

    cout << "Фургон может возить " << minivan.get_passengers() <<
        " на расстояние " << range1 << "\n";

    cout << "Спортивный автомобиль может возить " <<
```

Сделаем эти переменные закрытыми

Определим функции встроенными и получим доступ к закрытым переменным посредством функций доступа

```
sportscar.get_passengers() <<  
"на расстояние " << range2 << "\n";  
  
return 0;  
}
```

Поскольку переменные-члены **Vehicle** теперь закрыты, для получения в функции **main()** числа пассажиров, которые может перевозить данное транспортное средство, следует использовать функцию доступа **get_passengers()**.

Минутная тренировка

1. Для чего предназначен описатель **inline**?
 2. Можно ли встроенную функцию объявить внутри объявления **class**?
 3. Что такое функция доступа?
1. Функция, объявленная с описателем **inline**, не вызывается, а разворачивается внутри программного кода.
 2. Да, встроенная функция может быть объявлена внутри объявления **class**.
 3. Функция доступа — это короткая функция, которая получает или устанавливает значение закрытой переменной экземпляра.
-

Проект 8-2

Создание класса очереди

Как вы, наверное, знаете, *структура данных* — это способ организации данных. Простейшей структурой данных является *массив*, который представляет собой линейный список, поддерживающий случайный доступ к его элементам. Массивы часто используются как основание для создания более изощренных структур данных, например, стеков и очередей. *Стек* — это список, в котором доступ к элементам осуществляется исключительно в порядке “первым вошел, последним вышел” (First-In, Last-Out, FILO). Очередь — это список, в котором доступ к элементам осуществляется исключительно в порядке “первым вошел, первым вышел” (First-In, First-Out, FIFO). Стек напоминает стопку тарелок на столе; первая (нижняя) тарелка будет использована последней. Примером очереди являются люди, ждущие обслуживания у кассира банка: первый в очереди и обслужен будет первым.

Что делает структуры данных типа стека или очереди особенно интересным, так это совмещение в них функции хранения информации с функциями доступа к этой информации. С этой точки зрения стеки и очереди представляют собой *процессоры данных*, в которых хранение и извлечение предоставляются самими структурами данных, а не вашей

программой, в которой это надо делать своими руками. Такая комбинация, очевидно, превосходно отвечает идее класса, и в этом проекте вы создадите простой класс очереди.

Очереди в принципе поддерживают две операции: “положить” и “взять”. Каждая операция “положить” помещает новый элемент в конец очереди. Каждая операция “взять” извлекает следующий элемент из головы очереди. Операции с очередью носят разрушающий характер. После того как элемент извлечен из очереди, его нельзя извлечь повторно. Очередь может также переполниться, если в ней нет свободного места для сохранения элемента; она также может оказаться пустой, если из нее извлечены все элементы.

Последнее замечание: в принципе существуют два вида очередей, кольцевая и некольцевая. Кольцевая очередь повторно использует позиции лежащего в ее основе массива, из которых были удалены элементы; некольцевая не делает этого и в конце концов все позиции в ней исчерпываются. Ради простоты в этом примере будет создана некольцевая очередь, но затратив незначительные усилия, вы сами сможете преобразовать ее в очередь кольцевого типа.

Шаг за шагом

1. Создайте файл с именем **Queue.cpp**.

2. Существуют разные способы создания очереди; метод, которым мы воспользуемся, основан на массиве. Другими словами, память для хранения элементов, помещаемых в очередь, будет представлять собой массив. Обращение к этому массиву будет выполняться с помощью двух индексов. Индекс **putloc** определяет, куда должен быть помещен следующий элемент. Индекс **getloc** указывает, из какого места массива должен извлекаться следующий элемент данных. Не забудьте, что операция извлечения данного действует разрушающим образом; один и тот же элемент нельзя извлечь дважды. Очередь, которую мы будем создавать, предназначена для хранения символов, но та же самая логика используется для объектов любых типов. Начните создавать класс **Queue** с таких строк:

```
const int maxQsize = 100;
```

```
class Queue {  
    char q[maxQsize]; // массив для хранения очереди  
    int size; // максимальное число элементов,  
              // которые могут находиться в очереди  
    int putloc, getloc; // индексы "положить" и "взять"
```

Переменная **maxQsize** с описателем **const** определяет максимальный размер создаваемой очереди. Фактический размер очереди хранится в поле с именем **size**.

3. Конструктор для класса **Queue** создает очередь заданного размера. Вот его текст:

```
public:

// Сконструируем очередь конкретной длины.
Queue(int len) {
    // Размер очереди должен быть меньше max и положителен.
    if(len > maxQsize) len = maxQsize;
    else if(len <= 0) len = 1;

    size = len;
    putloc = getloc = 0;
}
```

Если затребованный размер очереди больше, чем **maxQsize**, то создается очередь максимально возможной длины. Если затребованный размер равен нулю или отрицателен, создается очередь длиной в 1 элемент. Размер очереди сохраняется в поле **size**. Индексам “положить” и “взять” присваивается начальное нулевое значение.

4. Функция **put()**, заносщая элементы в очередь, имеет такой вид:

```
// Поместим символ в очередь.
void put(char ch) {
    if(putloc == size) {
        cout << " -- Очередь полна.\n";
        return;
    }

    putloc++;
    q[putloc] = ch;
}
```

Функция начинается с проверки, не заполнена ли очередь. Если **putloc** равен размеру очереди, следовательно, в очереди больше нет места, куда можно было бы занести новый элемент. В противном случае **putloc** увеличивается на единицу, и новый элемент заносится в эту ячейку. Таким образом, **putloc** всегда является индексом последнего сохраненного элемента.

5. Для извлечения элементов из очереди предусмотрена функция **get()**:

```
// Извлечем символ из очереди.
char get() {
    if(getloc == putloc) {
        cout << " -- Очередь пуста.\n";
        return 0;
    }
```

```

    getloc++;
    return q[getloc];
}

```

Прежде всего обратите внимание на проверку, не пуста ли очередь. Если **getlock** и **putlock** оба указывают на один и тот же элемент, очередь считается пустой. Именно поэтому в конструкторе **Queue** оба индекса, и **getlock**, и **putlock**, были инициализированы нулевыми значениями. Далее **getlock** увеличивается на единицу и из очереди извлекается очередной элемент. Таким образом, **getlock** всегда является индексом ячейки, из которой был извлечен последний элемент.

6. Ниже приведен полный текст программы **Queue.cpp**:

```

/*
    Проект 8-2
    Класс очереди символов.
*/
#include <iostream>
using namespace std;

const int maxQsize = 100;

class Queue {
    char q[maxQsize];    // массив для хранения очереди
    int size;            // максимальное число элементов,
                        // которые могут находиться в очереди
    int putloc, getloc;  // индексы "положить" и "взять"
public:

    // Сконструируем очередь конкретной длины.
    Queue(int len) {
        // Размер очереди должен быть меньше max и положителен.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Поместим символ в очередь.
    void put(char ch) {
        if(putloc == size) {
            cout << " -- Очередь полна.\n";
            return;
        }

        putloc++;

```

```
    q[putloc] = ch;
}

// Извлечем символ из очереди.
char get() {
    if(getloc == putloc) {
        cout << " -- Очередь пуста.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}
};

// Демонстрация класса очереди Queue.
int main() {
    Queue bigQ(100);
    Queue smallQ(4);
    char ch;
    int i;

    cout << "Используем bigQ для хранения латинского алфавита.\n";
    // поместим в bigQ буквы алфавита
    for(i=0; i < 26; i++)
        bigQ.put('A' + i);

    // извлечем и выведем на экран элементы из bigQ
    cout << "Содержимое bigQ: ";
    for(i=0; i < 26; i++) {
        ch = bigQ.get();
        if(ch != 0) cout << ch;
    }

    cout << "\n\n";

    cout << "Используем smallQ для демонстрации ошибок.\n";
    // Теперь используем smallQ для демонстрации ошибок
    for(i=0; i < 5; i++) {
        cout << "Пытаемся записать " <<
            (char) ('Z' - i);

        smallQ.put('Z' - i);

        cout << "\n";
    }
}
```



```

cout << "\n";

// другая ошибка в smallQ
cout << "Содержимое smallQ: ";
for(i=0; i < 5; i++) {
    ch = smallQ.get();

    if(ch != 0) cout << ch;
}

cout << "\n";
}

```

7. Вывод программы выглядит следующим образом:

Используем bigQ для хранения латинского алфавита.
 Содержимое bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Используем smallQ для генерации ошибок.
 Пытаемся записать Z
 Пытаемся записать Y
 Пытаемся записать X
 Пытаемся записать W
 Пытаемся записать V -- Очередь полна.

Содержимое smallQ: ZYXW -- Очередь пуста.

8. Попробуйте самостоятельно модифицировать **Queue** таким образом, чтобы в очередь можно было записывать объекты других типов. Например, создайте очередь для переменных **int** или **double**.

Цель

8.7. Массивы объектов

Вы можете создавать массивы объектов точно так же, как и массивы любых других типов данных. Например, приведенная ниже программа создает массив объектов **MyClass**. Обращение к объектам, составляющим элементы массива, осуществляется обычным образом с помощью индексов:

```

// Создание массива объектов.

#include <iostream>
using namespace std;

```

```
class MyClass {
    int x;
public:
    void set_x(int i) { x = i; }
    int get_x() { return x; }
};

int main()
{
    MyClass obs[4]; ← Создание массива объектов
    int i;

    for(i=0; i < 4; i++)
        obs[i].set_x(i);
    for(i=0; i < 4; i++)
        cout << "obs[" << i << "].get_x(): " <<
            obs[i].get_x() << "\n";

    return 0;
}
```

Эта программа выводит на экран следующее:

```
obs[0].get_x(): 0
obs[1].get_x(): 1
obs[2].get_x(): 2
obs[3].get_x(): 3
```

Цель

8.8. Инициализация массивов объектов

Если класс включает в себя параметрический конструктор, массив объектов может быть инициализирован. В приведенном ниже примере **MyClass** является параметрическим классом, а **obs** представляет собой инициализированный массив объектов этого класса:

```
// Инициализация массива объектов.
#include <iostream>
using namespace std;

class MyClass {
    int x;
public:
```

```

    MyClass(int i) { x = i; }
    int get_x() { return x; }
};

int main()
{
    MyClass obs[4] = { -1, -2, -3, -4 }; ← Один из способов инициа-
    int i;                                лизации массива объектов.

    for(i=0; i < 4; i++)
        cout << "obs[" << i << "].get_x(): " <<
            obs[i].get_x() << "\n";

    return 0;
}

```

В этом примере конструктору MyClass передаются значения от -1 до -4. Программа выводит на экран следующее:

```

obs[0].get_x(): -1
obs[1].get_x(): -2
obs[2].get_x(): -3
obs[3].get_x(): -4

```

Синтаксис, использованный в списке инициализаторов последнего примера представляет собой сокращение более пространной формы:

```

MyClass obs[4] = { MyClass(-1), MyClass (-2), ← Другой способ иници-
    MyClass (-3), MyClass (-4) };             ализации массива.

```

Как уже объяснялось выше, если конструктор принимает только один аргумент, то выполняется неявное преобразование типа этого аргумента в тип класса. Более длинный вариант инициализации просто вызывает конструктор явным образом.

При инициализации массива объектов, конструктор которых принимает более одного аргумента, вы должны использовать длинный вариант инициализации. Например:

```

#include <iostream>
using namespace std;

class MyClass {
    int x, y;
public:
    MyClass(int i, int j) { x = i; y = j; }
    int get_x() { return x; }
    int get_y() { return y; }
}

```

```
};

int main()
{
    MyClass obs[4][2] = {
        MyClass(1, 2), MyClass(3, 4),
        MyClass(5, 6), MyClass(7, 8),
        MyClass(9, 10), MyClass(11, 12),
        MyClass(13, 14), MyClass(15, 16)
    };

    int i;

    for(i=0; i < 4; i++) {
        cout << obs[i][0].get_x() << ' ';
        cout << obs[i][0].get_y() << "\n";
        cout << obs[i][1].get_x() << ' ';
        cout << obs[i][1].get_y() << "\n";
    }

    return 0;
}
```

Если конструктор объекта требует двух или нескольких аргументов, требуется использовать длинную форму инициализации.

В этом примере конструктор класса **MyClass** требует два аргумента. В функции **main()** массив **obs** объявляется и инициализируется с использованием прямых вызовов конструктора **MyClass**. Инициализируя массивы, вы можете всегда использовать длинную форму инициализации, даже если объект принимает только один аргумент. Просто краткая форма более удобна, если аргумент всего один. Программа выводит следующее:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

Цель

8.9. Указатели на объекты

Вы можете обратиться к объекту либо непосредственно (как это делалось во всех предыдущих примерах), либо посредством указателя на

этот объект. При обращении к конкретному элементу объекта посредством указателя на объект вы должны использовать оператор-стрелку: \rightarrow . Для включения в текст программы этого оператора надо сначала ввести знак “минус”, а затем знак “больше”.

Для объявления указателя на объект вы используете тот же синтаксис, посредством которого вы будете объявлять указатель на любой другой тип данных. Приведенная ниже программа создает простой класс **P_example** и определяет объект этого класса с именем **ob**, а также указатель **p** на объект типа **P_example**. Далее в программе иллюстрируется непосредственное обращение к **ob**, а также косвенное обращение посредством указателя:

// 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(){ cout << num << "\n"; }
};

int main()
{
    P_example ob, *p; // объявим объект и указатель на него
    ob.set_num(1); // вызов функции непосредственно через ob
    ob.show_num();

    p = &ob; // присвоим p значение адреса ob
    p->set_num(20); // вызов функции посредством указателя на ob
    p->show_num();

    return 0;
}
```

Указатели на объекты используются так же, как и указатели других типов

Обратите внимание на использование оператора-стрелки.

Обратите внимание на то, что для получения адреса **ob** используется оператор **&** (получение адреса переменной) точно так же, как это делается для получения адреса переменной любого другого типа.

Как вы уже знаете, когда указатель инкрементируется или декрементируется, он фактически увеличивается или уменьшается таким образом, чтобы указывать на следующий элемент его базового типа. То же самое происходит при операциях инкремента или декремента указателя на объект: указатель начинает указывать на следующий объект. Для иллюстрации этого правила мы модифицировали предыдущую программу так, что **ob** стал двухэлементным массивом типа **P_example**. Обратите

внимание, как для обращения к двум элементам массива над указателем выполняются операции инкремента и декремента:

// Инкремент и декремент указателя на объект.

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

class P_example {
    int num;
public:
    void set_num(int val) { num = val; }
    void show_num(){ cout << num << "\n"; }
};

int main()
{
    P_example ob[2], *p;

    ob[0].set_num(10); // непосредственное обращение к объекту
    ob[1].set_num(20);

    p = &ob[0]; // получим указатель на первый элемент
    p->show_num(); // выведем значение ob[0] через указатель

    p++; // продвинемся к следующему объекту
    p->show_num(); // выведем значение ob[1] через указатель

    p--; // вернемся к предыдущему объекту
    p->show_num(); // снова выведем значение ob[0]

    return 0;
}
```

Программа выведет на экран **10, 20, 10**

Как будет показано в дальнейших разделах этой книги, указатели на объекты играют центральную роль в одной из наиболее важных концепций C++: полиморфизме.

Минутная тренировка

1. Можно ли присвоить начальные значения массиву объектов?
 2. Если в программе имеется указатель на объект, какой оператор следует использовать для обращения к члену?
 - 1 Да, массиву объектов могут быть даны начальные значения
 - 2 При обращении к члену через указатель следует использовать оператор-стрелку
-

Ссылки на объекты

На объекты можно ссылаться точно так же, как и на данные любых других типов. Здесь нет никаких особых ограничений или правил.

✓ Вопросы для самопроверки

1. В чем разница между классом и объектом?
2. Какое ключевое слово используется для объявления класса?
3. Копия чего содержится в каждом объекте?
4. Покажите, как объявить класс с именем **Test**, содержащий две закрытые переменные типа **int** с именами **count** и **max**.
5. Какое имя имеет конструктор? А какое имя имеет деструктор?
6. Класс объявлен следующим образом:

```
class Sample {  
    int i;  
public:  
    Sample(int x) { i = x }  
    // ...  
};
```

Покажите, как объявить объект **Sample**, чтобы **i** получило начальное значение 10.

7. Если функция-член объявлена внутри объявления класса, какого рода оптимизация выполняется автоматически?
8. Создайте класс **Triangle**, в котором хранятся длина основания и высота прямоугольного треугольника в двух закрытых переменных экземпляра. Включите в класс конструктор, устанавливающий значения этих переменных. Определите две функции. Первая функция **hypot()** должна возвращать длину гипотенузы треугольника, а вторая функция **area()** — его площадь.
9. Расширьте класс **Help**, чтобы он хранил целочисленный номер **ID** каждого пользователя класса. Выводите **ID** на экран при уничтожения объекта справочника. Предусмотрите функцию **getID()**, которая при вызове будет возвращать значение **ID**.

Модуль

9

Подробнее о классах

Цели, достигаемые в этом модуле

- 9.1 Познакомиться с перегрузкой конструкторов
- 9.2 Освоить присваивание объектов
- 9.3 Научиться передавать объекты функциям
- 9.4 Узнать о возврате объектов из функций
- 9.5 Рассмотреть конструкторы копий
- 9.6 Начать использовать дружественные функции
- 9.7 Изучить структуры и объединения
- 9.8 Понять, что такое `this`
- 9.9 Освоить основы перегрузки операторов
- 9.10 Рассмотреть перегрузку операторов с помощью функций-членов
- 9.11 Познакомиться с перегрузкой операторов с помощью функций-не членов

В этом модуле мы продолжим обсуждение классов, начатое в Модуле 8. Мы рассмотрим целый ряд тем, связанных с классами, включая перегрузку конструкторов, передачу объектов функциям и возврат объектов. Мы также остановимся на специальном типе конструктора, называемом *конструктором копий*, который используется, когда требуется получить копию объекта. Далее будут описаны дружественные функции, структуры и объединения и, наконец, ключевое слово **this**. Модуль завершается обсуждением перегрузки операторов, одним из самых замечательных средств C++.

Цель

9.1. Перегрузка конструкторов

Конструкторы, хотя они и предназначены для решения вполне определенных задач, мало отличаются от других функций, и их, в частности, можно перегружать. Для перегрузки конструктора класса просто объявите разные формы этого конструктора. Например, приведенная ниже программа определяет три конструктора:

// Перегрузка конструкторов.

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

```
class Sample {
public:
    int x;
    int y;
```

```
    // Перегрузим конструктор по умолчанию.
```

```
    Sample() { x = y = 0; }
```

```
    // Конструктор с одним параметром.
```

```
    Sample(int i) { x = y = i; }
```

```
    // Конструктор с двумя параметрами.
```

```
    Sample(int i, int j) { x = i; y = j; }
```

```
};
```

```
int main() {
```

```
    Sample t; // активизация конструктора по умолчанию
```

```
    Sample t1(5); // используем Sample(int)
```

```
    Sample t2(9, 10); // используем Sample(int, int)
```

```
    cout << "t.x: " << t.x << ", t.y: " << t.y << "\n";
```

← Перегруженные конструкторы **Sample**

```
cout << "t1.x: " << t1.x << ", t1.y: " << t1.y << "\n";  
cout << "t2.x: " << t2.x << ", t2.y: " << t2.y << "\n";  
  
return 0;  
}
```

Вот вывод этой программы:

```
t.x: 0, t.y: 0  
t1.x: 5, t1.y: 5  
t2.x: 9, t2.y: 10
```

В программе создаются три конструктора. Первым объявлен конструктор без параметров; он инициализирует *x* и *y*, присваивая им значение 0. Этот конструктор становится конструктором по умолчанию, замещающей собой конструктор по умолчанию, автоматически предоставляемый C++. Второй конструктор принимает один параметр, присваивая его значение одновременно *x* и *y*. Третий конструктор принимает два параметра, инициализируя *x* и *y* индивидуально.

Перегруженные конструкторы имеют целый ряд достоинств. Во-первых, они повышают гибкость создаваемого вами класса, позволяя конструировать объекты разными способами. Во-вторых, они создают удобства для пользователя вашего класса, позволяя ему конструировать объекты способом, наиболее естественным для решения его конкретной задачи. В третьих, определяя и конструктор по умолчанию, и параметрический конструктор, вы получаете возможность создавать как инициализированные, так и неинициализированные объекты.

Цель

9.2. Присваивание объектов

Если у вас есть два объекта одного типа (другими словами, два объекта одного класса), тогда один объект можно присвоить другому. При этом недостаточно, чтобы два класса были физически одинаковы — они должны иметь еще и одно имя. По умолчанию, если один объект присваивается другому, второму объекту фактически присваивается побитовая копия данных первого объекта. В этом случае после присваивания оба объекта будут идентичны, но существовать независимо. Приведенная ниже программа демонстрирует присваивание объектов:

```
// Демонстрация присваивания объектов.
```

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

```
class Test {
    int a, b;
public:
    void setab(int i, int j) { a = i, b = j; }
    void showab() {
        cout << "a равно " << a << '\n';
        cout << "b равно " << b << '\n';
    }
};

int main()
{
    Test ob1, ob2;

    ob1.setab(10, 20);
    ob2.setab(0, 0);
    cout << "ob1 перед присваиванием: \n";
    ob1.showab();
    cout << "ob2 перед присваиванием: \n";
    ob2.showab();
    cout << '\n';

    ob2 = ob1; // присвоим объект ob1 объекту ob2

    cout << "ob1 после присваивания: \n";
    ob1.showab();
    cout << "ob2 после присваивания: \n";
    ob2.showab();
    cout << '\n';

    ob1.setab(-1, -1); // изменим ob1

    cout << "ob1 после изменения ob1: \n";
    ob1.showab();
    cout << "ob2 после изменения ob1: \n";
    ob2.showab();

    return 0;
}
```

Присваивание одного объекта другому

Вот вывод этой программы:

```
ob1 перед присваиванием:
a равно 10
b равно 20
ob2 перед присваиванием:
```

```
a равно 0
b равно 0

ob1 после присваивания:
a равно 10
b равно 20
ob2 после присваивания:
a равно 10
b равно 20

ob1 после изменения ob1:
a равно -1
b равно -1
ob2 после изменения ob1:
a равно 10
b равно 20
```

Как видно из этой программы, присваивание одного объекта другому создает два объекта, содержащих одни и те же значения. Однако в остальном эти два объекта совершенно независимы. Так, последующая модификация данных одного объекта никак не отражается на данных другого. Однако вам следует опасаться побочных эффектов, которые все же могут возникнуть. Например, если объект А содержит указатель на некоторый другой объект В, тогда и копия объекта А будет содержать поле с указателем на тот же объект В. В этом случае изменение объекта В повлияет на обе копии. В такого рода ситуациях лучше обойти побитовое копирование, осуществляемое по умолчанию, путем определения для данного класса собственного оператора присваивания, как это будет описано ниже.

Цель

9.3. Передача объектов функциям

Объект можно передать функции точно так же, как и данное любого другого типа. Объекты передаются функциям посредством обычного соглашения C++ о передаче параметров по значению. Это означает, что в функцию передается не сам объект, а его *копия*. Следовательно, изменения объекта внутри функции не отражаются на объекте, использованном в качестве аргумента функции. Приведенная ниже программа иллюстрирует сказанное:

```
// Передача функции объекта.
```

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

```
class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
    }

    int getval() { return val; }
    void setval(int i) { val = i; }
};

void display(MyClass ob) ← Функция display( ) принимает объект
                           класса MyClass в качестве параметра
{
    cout << ob.getval() << '\n';
}

void change(MyClass ob) ← Функция change( ) также принимает объ-
                           ект класса MyClass в качестве параметра
{
    ob.setval(100); // не влияет на аргумент
    cout << "Значение ob внутри change(): ";
    display(ob);
}

int main()
{
    MyClass a(10);

    cout << "Значение a перед вызовом change(): ";
    display(a); ←
    change(a); ← Передача объекта класса MyClass функциям
                  display( ) и change( )
    cout << "Значение a после вызова change(): ";
    display(a);

    return 0;
}
```

Ниже показан вывод программы:

```
Значение a перед вызовом change(): 10
Значение ob внутри change(): 100
Значение a после вызова change(): 10
```

Как видно из вывода программы, изменение значения **ob** внутри **change()** не влияет на значение **a** внутри **main()**.

Конструкторы, деструкторы и передача объектов

Хотя передача простых объектов в качестве аргументов функций является очевидной процедурой, в отношении конструкторов и деструкторов могут возникать некоторые неожиданные явления. Чтобы понять, что здесь происходит, рассмотрим следующую программу:

```
// Конструкторы, деструкторы и передача объектов.
#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора \n";
    }

    ~MyClass() { cout << " Уничтожаем\n"; }
    int getval() { return val; }
};

void display(MyClass ob)
{
    cout << ob.getval() << '\n';
}

int main()
{
    MyClass a(10);

    cout << "Перед вызовом display().\n";
    display(a);
    cout << "После возврата из display().\n";

    return 0;
}
```

Результатом работы программы будет этот неожиданный вывод:

```
Внутри конструктора
Перед вызовом display().
10
Уничтожаем
После возврата из display().
Уничтожаем ←————— Обратите внимание на второе сообщение "Уничтожаем".
```

Вы видите, что по ходу программы конструктор вызывается один раз (что происходит, когда создается объект `a`), но имеются *два* вызова деструктора. Посмотрим, почему так получается.

Когда объект передается функции, создается копия этого объекта. (И эта копия становится параметром функции.) Это значит, что начинает существовать новый объект. Когда функция завершается, копия аргумента (т. е. параметр функции) уничтожается. Здесь возникает два фундаментальных вопроса. Первый: вызывается ли конструктор объекта когда создается его копия? И второй: вызывается ли деструктор объекта, когда копия уничтожается? Ответы вполне могут, по всяком случае, поначалу, удивить вас.

Когда при вызове функции создается копия аргумента, обычный конструктор *не* вызывается. Вместо этого вызывается *конструктор копий* объекта. Конструктор копий определяет, каким образом должна создаваться копия объекта. (Позже в этом модуле будет рассказано, как создать собственный конструктор копий.) При этом, если в классе конструктор копий не описан явным образом, C++ предоставляет такой конструктор по умолчанию. Конструктор копий по умолчанию создает *побитовую* (т. е. полностью идентичную) копию объекта. Причина, по которой создается именно побитовая копия, если вдуматься, очевидна. Поскольку обычный конструктор используется для той или иной инициализации объекта, он не должен вызываться ради копирования уже существующего объекта. Такой вызов изменит содержимое объекта. Передавая объект функции, вы хотите использовать текущее, а не начальное состояние объекта.

Однако, когда функция завершается, и копия объекта, использованная в качестве аргумента, уничтожается, вызывается функция деструктора. Это необходимая операция, так как объект уходит из области видимости. Вот почему в последнем примере мы видели два вызова деструктора. Первый вызов был сделан, когда параметр функции `display()` вышел из области видимости. Второй — когда при завершении программы был уничтожен объект `a` внутри `main()`.

Подытожим: Когда создается копия объекта ради использования ее в качестве аргумента функции, обычный конструктор не вызывается. Вместо него вызывается конструктор копий по умолчанию, который создает побитовую, идентичную копию объекта. Однако при уничтожении копии (обычно это происходит при завершении функции, когда объект уходит из области видимости) вызывается обычный деструктор.

Передача объектов по ссылке

Другим способом передачи объекта в функцию является передача по ссылке. В этом случае в функцию передается ссылка на объект, и функция выполняет операции непосредственно над объектом, используемым в качестве аргумента. Соответственно, изменения, вносимые функцией в параметр, *будут* изменять аргумент, из чего следует, что передача по

ссылке применима не во всех случаях. Однако, в тех случаях, когда она допустима, возникают два преимущества. Во-первых, поскольку в функции передается только адрес объекта, а не целый объект, передача по ссылке должна осуществляться значительно быстрее и эффективнее передачи объекта по значению. Во-вторых, когда объект передается по ссылке, не появляется никаких новых объектов и, соответственно, не тратится время на конструирование и уничтожение временного объекта.

Ниже приведен пример, иллюстрирующий передачу объекта по ссылке:

```
// Конструкторы, деструкторы и передача объектов.
#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора\n";
    }

    ~MyClass() { cout << "Уничтожаем\n"; }
    int getval() { return val; }
    void setval(int i) { val = i; }
};

void display(MyClass &ob) ← Здесь объект класса MyClass
{                               передается по ссылке.
    cout << ob.getval() << '\n';
}

void change(MyClass &ob) ←
{
    ob.setval(100);
}

int main()
{
    MyClass a(10);

    cout << "Перед вызовом display().\n";
    display(a);
    cout << "После возврата из display().\n";

    change(a);
    cout << "После вызова change().\n";
}
```



```
display(a);  
  
return 0;  
}
```

Ниже приведен вывод этой программы:

```
Внутри конструктора  
Перед вызовом display().  
10  
После возврата из display().  
После вызова change().  
100  
Уничтожаем
```

В этой программе обе функции, и `display()`, и `change()`, используют передачу параметра по ссылке. Следовательно, в функцию передается не копия аргумента, а его адрес, и функция выполняет операции непосредственно над аргументом. Например, когда вызывается функция `change()`, ей по ссылке передается `a`. Изменения параметра `ob`, выполненные в `change()`, влияют на `a` в `main()`. Заметьте также, что выполняются только по одному вызову конструктора и деструктора. Так происходит потому, что создается и уничтожается только один объект, а Во временных объектах эта программа не нуждается.

Потенциальные проблемы при передаче объектов

Даже когда объекты передаются в функции обычным способом передачи по значению, что теоретически защищает и изолирует передаваемый аргумент, могут возникать побочные эффекты, которые могут влиять на объект, используемый в качестве аргумента, вплоть до его уничтожения. Например, если объект при своем создании выделяет некоторые системные ресурсы (скажем, память), а при уничтожении эти ресурсы освобождает, тогда локальная копия внутри функции, когда будет вызван деструктор, освободит эти ресурсы. Это вызовет проблемы, так как исходный объект все еще использует эти ресурсы. Обычно такая ситуация завершается разрушением исходного объекта.

Одно из решений этой проблемы состоит в передаче объекта по ссылке, как это было показано в предыдущем подразделе. В этом случае не создается копии объекта, и, соответственно, объект не уничтожается при выходе из функции. Как уже отмечалось ранее, передача объектов по ссылке может также ускорить вызов функции, поскольку фактически в функцию передается только адрес объекта. Однако передача объекта по ссылке применима не во всех случаях. К счастью, существует

более общее решение: вы можете создать собственный конструктор копий. В этом случае вы можете точно определить, как должна создаваться копия объекта и, тем самым, избежать описанных выше проблем. Однако перед тем, приступить к изучению конструктора копий, рассмотрим другой связанный с копированием вопрос, где также могут проявиться преимущества собственного конструктора копий.

Цель

9.4. Возврат объектов

Объекты могут не только передаваться в функции, но и возвращаться из них. Для возврата объекта прежде всего необходимо при объявлении функции указать, что она возвращает тип класса. В самой функции следует вернуть объект, используя обычное предложение **return**. Приведенная ниже программа имеет функцию-член с именем **mkBigger()**. Эта функция возвращает объект, который придает переменной **val** значение вдвое большее, чем вызывающий объект:

```
// Возврат объектов.

#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    // Обычный конструктор.
    MyClass(int i) {
        val = i;
        cout << "Внутри конструктора\n";
    }
    ~MyClass() {
        cout << "Уничтожаем\n";
    }

    int getval() { return val; }
    // Возврат объекта.
    MyClass mkBigger() { ←———— mkBigger( ) возвращает объект класса MyClass
        MyClass o(val * 2);
        return o;
    }
};

void display(MyClass ob)
{
```

```

    cout << ob.getval() << '\n';
}

int main()
{
    cout << "Перед конструированием а.\n";
    MyClass a(10);
    cout << "После конструирования а.\n\n";

    cout << "Перед вызовом display().\n";
    display(a);
    cout << "После возврата из display().\n\n";

    cout << "Перед вызовом mkBigger().\n";
    a = a.mkBigger();
    cout << "После возврата из mkBigger().\n\n";

    cout << "Перед вторым вызовом display().\n";
    display(a);
    cout << "После возврата из display().\n\n";

    return 0;
}

```

Программа выводит следующее:

Перед конструированием а.

Внутри конструктора

После конструирования а.

Перед вызовом display().

10

Уничтожаем

После возврата из display().

Перед вызовом mkBigger().

Внутри конструктора

Уничтожаем

Уничтожаем ←————— Обратите внимание на второе сообщение "Уничтожаем"

После возврата из mkBigger().

Перед вторым вызовом display().

20

Уничтожаем

После возврата из display().

Уничтожаем

В этом примере функция `mkBigger()` создает локальный объект с именем `a`, у которого значение `val` вдвое больше, чем у вызывающего объекта. Этот объект затем возвращается функцией и присваивается объекту `a` внутри `main()`. Далее `a` уничтожается, в результате чего появляется первое сообщение “Уничтожаем”. Но откуда берется второй вызов деструктора?

Когда функция возвращает объект, то автоматически создается временный объект, содержащий возвращаемое значение. Именно этот объект фактически возвращается функцией. После того, как значение возвращено, этот объект уничтожается. Вот почему на экран было выведено второе сообщение “Уничтожаем” перед сообщением “После возврата из `mkBigger()`”. Это индикация уничтожения временного объекта.

Как было и в случае передачи функции объекта, при возврате объекта из функции возникают потенциальные проблемы. Уничтожение этого временного объекта может в некоторых ситуациях привести к неожиданным побочным эффектам. Например, если объект, возвращаемый функцией, имеет деструктор, освобождающий ресурс (память или дескриптор файла), этот ресурс будет освобожден, несмотря на то, что объект, которому присвоено возвращаемое значение, все еще использует его. Решение этой проблемы требует создания конструктора копий, который будет описан в следующем подразделе.

Последнее замечание: функция может вернуть объект по ссылке, но нужно проявлять осторожность, чтобы объект, на который указывает ссылка, не ушел из области видимости при завершении функции.

Минутная тренировка

1. Конструкторы не могут перегружаться. Правильно ли это?
 2. Когда объект передается в функцию по значению, создается его копия. Уничтожается ли эта копия, когда происходит возврат из функции?
 3. Когда функция возвращает объект, создается временный объект, содержащий возвращаемое значение. Правильно ли это?
1. Неправильно, конструкторы могут перегружаться
 2. Да, копия аргумента уничтожается, когда происходит возврат из функции
 3. Правильно, возвращаемое из функции значение представляет собой временный объект
-

Цель

9.5.

Создание и использование конструктора копий

Как показали предыдущие примеры, когда объект передается функции или возвращается из функции, создается копия этого объекта. По умолчанию эта копия есть побитовый дубликат исходного объекта. Такое копирование по умолчанию часто оказывается приемлемым, одна-

ко в тех случаях, когда оно вас не устраивает, вы можете указать точный способ создания копии, явным образом определив конструктор копий для класса. Конструктор копий (его еще называют копирующим конструктором или конструктором копирования) представляет собой специальный тип перегруженного конструктора, который активизируется автоматически, как только потребуется копия объекта.

Для начала вспомним, зачем нам понадобилось явно определять конструктор копий. Когда объект передается в функцию, создается побитовая (т. е. точная) копия этого объекта, которая становится параметром, принимающим объект. Однако имеются случаи, когда такая идентичная копия нежелательна. Например, если объект использует ресурс, скажем, открытый файл, тогда копия будет использовать *тот же самый* ресурс, что и исходный объект. В результате если копия как-то изменяет ресурс, он будет изменен также и для исходного объекта! Более того, когда функция завершается, копия объекта будет уничтожена с вызовом деструктора объекта. Это также может привести к нежелательному воздействию на исходный объект.

Схожая ситуация возникает, если объект возвращается функцией. Компилятор в этом случае создаст временный объект, содержащий копию значения, возвращаемого функцией. (Это делается автоматически, и мы никак на эту процедуру повлиять не можем.) Этот временный объект уходит из области видимости, как только значение вернулось в вызывающую программу, что приводит к вызову деструктора временного объекта. Если, однако, деструктор уничтожает что-то, требуемое вызывающей программой, возникнут неприятности.

В основе этих проблем лежит создание побитовой копии объекта. Для того, чтобы от проблем избавиться, необходимо точно определить, что должно происходить, когда создается копия объекта, чтобы избежать нежелательных побочных эффектов. Для этого надо создать собственный конструктор копий.

Перед тем, как мы приступим к исследованию возможностей конструктора копий, важно подчеркнуть, что C++ определяет два вида ситуаций, в которых значение одного объекта передается другому. Первая ситуация — присваивание. Вторая — инициализация, причем она может происходить тремя способами:

- когда один объект явным образом инициализирует другой, как это имеет место при объявлении;
- когда создается копия объекта с целью передачи ее в функцию;
- когда создается временный объект (обычно в качестве возвращаемого значения).

Конструктор копий активизируется только при инициализации. Операция присваивания не вызывает конструктор копий.

Наиболее общая форма конструктора копий выглядит так:

```
имя-класса (const имя-класса& объект) {
    //тело конструктора
}
```

Здесь *объект* представляет собой ссылку на объект, используемый для инициализации другого объекта. Пусть, например, у нас есть класс **MyClass**, и *y* есть объект типа **MyClass**. Тогда следующие предложения будут вызывать конструктор копий класса **MyClass**:

```
MyClass x = y; // y явным образом инициализирует x
func1(y);      // y передается как параметр
y = func2();    // y принимает возвращаемый объект
```

В первых двух случаях конструктору копий будет передана ссылка на *y*. В третьем случае конструктору копий будет передана ссылка на объект, возвращаемый функцией `func2()`. Таким образом, в тех случаях, когда объект передается в качестве параметра, возвращается функцией или используется в операции инициализации, вызывается конструктор копий с целью дублирования объекта.

Не забывайте, что конструктор копий не вызывается, когда один объект присваивается другому. Например, в приведенном ниже фрагменте конструктор копий вызван не будет:

```
MyClass x;
MyClass y;

x = y; // здесь конструктор копий не вызывается.
```

Последнее предложение обслуживается оператором присваивания, а не конструктором копий.

Приведенная ниже программа демонстрирует создание конструктора копий:

```
/* Конструктор копий активизируется, когда объект
   передается функции. */

#include <iostream>
using namespace std;

class MyClass {
    int val;
    int copynumber;
public:
    // Обычный конструктор.
    MyClass(int i) {
        val = i;
        copynumber = 0;
        cout << "Внутри обычного конструктора\n";
    }
}
```

```

// Конструктор копий
MyClass(const MyClass &o) { ←—————Это конструктор копий класса MyClass
    val = o.val;
    copynumber = o.copynumber + 1;
    cout << "Внутри конструктора копий.\n";
}

~MyClass() {
    if(copynumber == 0)
        cout << "Уничтожение оригинала.\n";
    else
        cout << "Уничтожение копии " <<
            copynumber << "\n";
}

int getval() { return val; }
};

void display(MyClass ob)
{
    cout << ob.getval() << '\n';
}

int main()
{
    MyClass a(10);

    display(a); ←—————Конструктор копий вызывается, когда функции
                        display( ) передается аргумент a

    return 0;
}

```

Программа выводит на экран следующее:

```

Внутри обычного конструктора
Внутри конструктора копий.
10
Уничтожение копии 1
Уничтожение оригинала.

```

Вот что происходит при запуске программы. Когда внутри **main()** создается объект **a**, значение его переменной **copynumber** устанавливается в 0 обычным конструктором. Далее **a** передается параметру **ob** функции **display()**. В процессе передачи параметра вызывается конструктор копий, и создается копия **a**. Конструктор копирования, выполняя свой код, инкрементирует значение **copynumber**. Когда **display()** завершается,

ob уходит из области видимости. Это приводит к вызову его деструктора. Наконец, когда завершается **main()**, **a** уходит из области видимости.

Вы можете попробовать поэкспериментировать с этой программой. Например, создайте функцию, которая возвращает объект класса **MyClass**, и посмотрите, когда будет вызван конструктор копий.

Минутная тренировка

1. Каким образом создается копия объекта, когда вызывается конструктор копий по умолчанию?
 2. Конструктор копий вызывается, когда одному объекту присваивается значение другого. Правильно ли это?
 3. Почему может возникнуть необходимость в явном определении конструктора копий для класса?
 - 1 Конструктор копий по умолчанию создает побитовую (т.е. идентичную) копию
 - 2 Неправильно, конструктор копий не вызывается, когда один объект присваивается другому
 - 3 Вам может понадобиться явное определение конструктора копий, когда копия объекта не должна быть идентичной оригиналу, например, для предотвращения нанесения повреждений объекту-оригиналу
-

Цель

9.2. Дружественные функции

В принципе к закрытым членам класса имеют доступ только другие члены этого класса. Однако имеется возможность организовать доступ к закрытым членам класса из функции, не входящей в данный класс, посредством объявления ее *другом* класса. Чтобы сделать функцию другом класса (создать *дружественную* классу функцию), вы включаете ее прототип в открытую (**public**) секцию объявления класса и предваряете ее ключевым словом **friend**. Например, в приведенном ниже фрагменте функция **frnd()** объявлена другом класса **MyClass**:

```
class MyClass {  
    // ...  
public:  
    friend void frnd(MyClass ob);  
    // ...  
};
```

Вы видите, что ключевое слово **friend** предваряет остальные элементы прототипа. Функция может быть другом более чем одного класса.

Ниже приведен короткий пример использования дружественной функции для выяснения того, имеют ли закрытые поля класса **MyClass** общий знаменатель:


```
// Демонстрация дружественной функции.

#include <iostream>
using namespace std;

class MyClass {
    int a, b;
public:
    MyClass(int i, int j) { a=i; b=j; }
    friend int comDenom(MyClass x); // дружественная функция
};

/* Обратите внимание на то, что comDenom()
   не является функцией-членом какого-либо класса. */
int comDenom(MyClass x)
{
    /* Поскольку comDenom() дружественна классу MyClass,
       она может непосредственно обращаться к a и b. */
    int max = x.a < x.b ? x.a : x.b;

    for(int i=2; i <= max; i++)
        if((x.a%i)==0 && (x.b%i)==0) return i;

    return 0;
}

int main()
{
    MyClass n(18, 111);

    if(comDenom(n)) ← comDenom вызывается обычным образом без
        cout << "Общий знаменатель равен " <<      указания объекта и оператора-точки
        comDenom(n) << "\n";
    else
        cout << "Общего знаменателя нет.\n";

    return 0;
}
```

В этом примере функция **comDenom()** не является членом **MyClass**. Однако она имеет полный доступ к закрытым членам **MyClass**. Конкретно, она может обратиться к **x.a** и **x.b**. Заметьте также, что **comDenom()** вызывается обычным образом, т. е. без связи с каким-либо объектом и без оператора-точки. Поскольку **comDenom()** не является функцией-членом, ее не надо уточнять именем объекта (собственно говоря, ее *нельзя* уточнять именем объекта). Обычно дружественной

функции передаются один или несколько объектов класса, которому она является другом, как это и имеет место в случае `comDenom()`.

Хотя мы не получили никаких преимуществ, объявив `comDenom()` другом, а не членом `MyClass`, существуют обстоятельства, при которых дружественная функция оказывается весьма ценным средством. Во-первых, друзья могут быть полезны при перегрузке некоторых типов операторов, как это будет показано ниже в этом же модуле. Во-вторых, дружественные функции упрощают создание некоторых типов функций ввода-вывода, как это будет показано в Модуле 11.

Третья причина, оправдывающая введение в язык дружественных функций, заключается в том, что в некоторых случаях два или несколько классов могут содержать члены, взаимосвязанные с другими частями вашей программы. Представьте себе, например, два различных класса, `Cube` и `Cylinder`, которые определяют характеристики куба и цилиндра, причем одной из характеристик является цвет объекта. Чтобы получить возможность легко сравнивать цвета куба и цилиндра, вы можете определить дружественную функцию, которая сравнивает цветовые характеристики двух объектов, возвращая истину, если цвета совпадают и ложь, если они различаются. Приведенная ниже программа иллюстрирует эту идею:

```
// Дружественные функции могут быть друзьями двух или более
// классов.

#include <iostream>
using namespace std;

class Cylinder; // упреждающее объявление

enum colors { red, green, yellow };

class Cube {
    colors color;
public:
    Cube(colors c) { color = c; }
    friend bool sameColor(Cube x, Cylinder y); ← sameColor( ) является другом Cube
    // ...
};

class Cylinder {
    colors color;
public:
    Cylinder(colors c) { color = c; }
    friend bool sameColor(Cube x, Cylinder y); ← sameColor( ) является также другом Cylinder.
    // ...
};
```

```
bool sameColor(Cube x, Cylinder y)
{
    if(x.color == y.color) return true;
    else return false;
}

int main()
{
    Cube cubel(red);
    Cube cube2(green);
    Cylinder cyl(green);

    if(sameColor(cubel, cyl))
        cout << "куб1 and цилиндр одного цвета.\n";
    else
        cout << "куб1 and цилиндр разных цветов.\n";

    if(sameColor(cube2, cyl))
        cout << "куб2 and цилиндр одного цвета.\n";
    else
        cout << "куб2 and цилиндр разных цветов.\n";

    return 0;
}
```

Вот вывод этой программы:

```
куб1 and цилиндр разных цветов.
куб2 and цилиндр одного цвета.
```

Обратите внимание на то, что эта программа использует *упреждающее объявление* (называемое также *упреждающей ссылкой*, или *ссылкой вперед*) в отношении класса **MyClass**. Такая ссылка необходима из-за того, что объявление функции **sameColor()** внутри **Cube** ссылается на класс **Cylinder** еще перед его объявлением. Для того, чтобы создать упреждающее объявление класса, просто используйте форму, показанную в этой программе.

Друг одного класса может быть членом другого. Например, предыдущую программу можно переписать так, чтобы функция **sameColor()** была членом **Cube**. Обратите внимание на оператор разрешения видимости при объявлении **sameColor()** другом **Cylinder**:

```
/* Функция может быть членом одного класса
   и другом второго. */

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

```

class Cylinder; // упреждающее объявление

enum colors { red, green, yellow };

class Cube {
    colors color;
public:
    Cube(colors c) { color= c; }
    bool sameColor(Cylinder y); ← sameColor( ) теперь член класса Cube
    // ...
};

class Cylinder {
    colors color;
public:
    Cylinder(colors c) { color = c; }
    friend bool Cube::sameColor(Cylinder y); ← Cube::sameColor( ) является членом класса Cylinder
    // ...
};

bool Cube::sameColor(Cylinder y) {
    if(color == y.color) return true;
    else return false;
}

int main()
{
    Cube cub1(red);
    Cube cube2(green);
    Cylinder cyl(green);

    if(cub1.sameColor(cyl))
        cout << "куб1 and цилиндр одного цвета.\n";
    else
        cout << "куб1 and цилиндр разных цветов.\n";

    if(cube2.sameColor(cyl))
        cout << "куб2 and цилиндр одного цвета.\n";
    else
        cout << "куб2 and цилиндр разных цветов.\n";

    return 0;
}

```

Поскольку функция **sameColor()** является членом **Cube**, она может непосредственно обращаться к переменной **color** объектов типа **Cube**. Только объекты типа **Cylinder** должны передаваться этой функции.

Минутная тренировка

1. Что такое дружественная функция? С помощью какого ключевого слова она объявляется?
 2. Если дружественная функция вызывается для объекта, используется ли оператор-точка?
 3. Может ли дружественная функция одного класса быть членом другого?
1. Дружественная функция — это функция, не являющаяся членом класса, но имеющая доступ к закрытым членам класса, для которого она является другом. Дружественная функция объявляется с помощью ключевого слова **friend**.
 2. Нет, дружественная функция вызывается как обычная функция, не являющаяся членом класса.
 3. Да, друг одного класса может быть членом другого.

Цель

9.7. Структуры и объединения

В дополнение к ключевому слову **class**, C++ предоставляет еще две возможности создать структуру данных классового типа. Во-первых, можно создать *структуру*. Во-вторых, можно создать *объединение*. Ниже описываются обе эти возможности.

Структуры

Структуры унаследованы из языка C; они объявляются с помощью ключевого слова **struct**. Описатель **struct** синтаксически схож с **class**, оба создают классový тип. В языке C **struct** может содержать только данные-члены, но в C++ это ограничение снято. В C++ **struct** — это, в сущности, просто другой способ задать класс. Фактически в C++ единственное различие между **class** и **struct** заключается в том, что по умолчанию все члены в **struct** открыты, а в **class** закрыты. Во всех остальных отношениях структуры и классы эквивалентны.

Вот пример структуры:

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

```
struct Test {
    int get_i() { return i; } // эти члены public
    void put_i(int j) { i = j; } // по умолчанию
private:
    int i;
};
int main()
{
```

Члены структуры по умолчанию открыты.

```
Test s;  
  
s.put_i(10);  
cout << s.get_i();  
  
return 0;  
}
```

Эта простая программа определяет структурный тип с именем **Test**, в котором **get_i()** и **put_i()** открыты, а **i** закрыта. Обратите внимание на использование ключевого слова **private** для определения закрытых элементов структуры.

Ниже приведена программа, эквивалентная предыдущей, но в которой вместо **struct** использовано ключевое слово **class**:

```
#include <iostream>  
using namespace std;  
  
class Test {  
    int i; // private по умолчанию  
public:  
    int get_i() { return i; }  
    void put_i(int j) { i = j; }  
};  
  
int main()  
{  
    Test s;  
  
    s.put_i(10);  
    cout << s.get_i();  
  
    return 0;  
}
```

В большинстве своем программисты на C++ используют **class** для определения формы объекта, содержащего функции-члены, а **struct** используют традиционным образом, для описания объектов, содержащих только данные-члены. Иногда для описания структуры, которая не содержит членов-функций, используется аббревиатура “POD”, означающая “plain old data”, что приблизительно можно перевести как “чистые данные, как в старину”.

Спросим у эксперта

Вопрос: Если **struct** и **class** так похожи, почему в C++ имеются оба ключевые слова?

Ответ: На первый взгляд наличие в языке структур и классов, имеющих фактически одинаковые возможности, является избыточным. Многие начинающие работать на C++ удивляются, зачем существует эта очевидное дублирование. Нередко можно даже услышать предложение устранить одно из этих ключевых слов, либо **class**, либо **struct**.

Ответ на такие рассуждения заключается в том, что разработчики C++ старались обеспечить совместимость с C. В C++, как он определен на сегодня, стандартная структура C является вполне допустимой. В C, где отсутствует концепция открытых или закрытых членов структуры, все члены структуры по умолчанию открыты. Поэтому и в C++ члены структуры по умолчанию объявляются **public**, а не **private**. Ключевое слово **class** было введено с отчетливой идеей поддержки инкапсуляции, и представляется разумным, что члены класса по умолчанию объявляются закрытыми. Однако, чтобы не потерять совместимость с C в этом отношении, умолчание для структур надо было оставить, поэтому и было добавлено новое ключевое слово. Однако имеется и более серьезное основание для разделения структур и классов. Поскольку **class** является сущностью, синтаксически отделенной от **struct**, определение класса имеет возможность развиваться в направлениях, которые уже не будут синтаксически совместимы с C-подобными структурами. Раз эти два понятия разделены, будущее развитие C++ не будет связано заботами о совместимости с C-подобными структурами.

Объединения

Объединение представляет собой область памяти, совместно используемую двумя или большим числом переменных. Объединение создается посредством ключевого слова **union**, и его объявление схоже с объявлением структуры, как это показано в следующем примере:

```
union utype {  
    short int i;  
    char ch;  
};
```

Здесь определено объединение, в котором значения **short int** и **char** занимают одну и ту же ячейку памяти. Однако следует внести ясность: *невозможно*, чтобы объединение содержало *обе* переменные, целочисленную и символьную, *в одно и то же время*, потому что **i** и **ch** перекрывают друг друга. Программа может в любой момент трактовать данные в объединении либо как целочисленную переменную, либо как символ. Таким образом, объединение дает вам возможность рассматривать двумя или большим числом способов одну и ту же порцию данных.

Вы можете объявить переменную типа объединения, поместив ее имя в конец объявления **union**, или использовав отдельное предложе-

ние объявления. Например, для объявления объединенной переменной `u_var` типа `utype`, можно написать:

```
utype u_var;
```

В `u_var` обе переменные, и целочисленная `i`, и символьная `ch`, используют одну и ту же память. (Разумеется, `i` занимает два байта, а `ch` использует только один.) На рис. 9-1 показано, как располагаются `i` и `ch` по одному и тому же адресу.

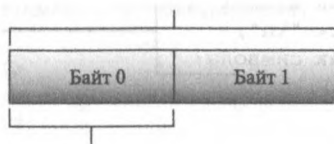


Рис. 9-1. Как переменные `i` и `ch` размещаются по одному и тому же адресу

Согласно идеологии C++, объединение представляет собой класс, в котором все элементы занимают одну и ту же память. Фактически `union` определяет классовой тип. Объединение может содержать функции конструктора и деструктора, так же, как и функции-члены. Поскольку объединение унаследовано от C, его члены по умолчанию открыты.

Ниже приведена программа, использующая объединение для вывода на экран символов, составляющих младший и старший байты короткого целого (в предположении, что короткое целое занимает два байта):

```
// Демонстрация объединения.
#include <iostream>
using namespace std;

union u_type {
    u_type(short int a) { i = a; };
    u_type(char x, char y) { ch[0] = x; ch[1] = y; }

    void showchars(){
        cout << ch[0] << " ";
        cout << ch[1] << "\n";
    }

    short int i;
    char ch[2];
};
```

Данные-члены объединения располагаются в одной и той же памяти


```

int main()
{
    u_type u(1000);
    u_type u2('X', 'Y');

    cout << "u как целое: ";
    cout << u.i << "\n";
    cout << "u как символы: ";
    u.showchars();

    cout << "u2 как целое: ";
    cout << u2.i << "\n";
    cout << "u2 как символы: ";
    u2.showchars();

    return 0;
}

```

Данное в объекте **u_type** могут рассматриваться как **short int** или как два **char**

Вывод программы выглядит так.

```

u как целое: 1000
u как символы: ⅂ ♡
u2 как целое: 22872
u2 как символы: X Y

```

Как это видно из вывода программы, используя объединение `u_type`, можно рассматривать одни и те же данные разными способами.

Как и структуры, объединения C++ произошли от своего предшественника в С. Однако в С объединения могут включать только данные-члены; функции и конструкторы недопустимы. В C++ объединения имеют расширенные возможности класса. Однако из того, что C++ придает объединениям большие возможности и гибкость, не следует, что вы должны их использовать. Часто объединения содержат только данные. Однако в случаях, когда вы сможете инкапсулировать объединение вместе с манипулирующими им программами, вы получите возможность добавить в свою программу весьма сложную структуру.

В C++ существует ряд ограничений на использование объединений. Большая их часть связана со средствами C++, которые будут обсуждаться в последующих разделах этой книги, поэтому мы упомянем их здесь только для полноты картины. Прежде всего, **union** не может наследовать класс. Далее, **union** не может выполнять роль базового класса. **union** не может включать виртуальные функции-члены. Переменные **static** не могут быть членами **union**. Нельзя использовать член-ссылку. **union** не может иметь в качестве члена никакой объект, перегружающий оператор `=`. Наконец, никакой объект не может быть членом **union**, если этот объект имеет явный конструктор или деструктор.

Безымянные объединения

В C++ имеется специальный тип объединения, называемый *безымянным объединением*. Безымянное объединение не включает имя типа, и для такого объединения нельзя объявить переменные. Такое объединение просто указывает компилятору, что его члены должны размещаться в общей памяти. Однако обращение к самим переменным осуществляется непосредственно, без использования обычного синтаксиса с оператором-точкой. Рассмотрим в качестве примера такую программу:

```
// Демонстрация безымянного объединения.
```

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

```
int main()
{
```

```
    // объявим безымянное объединение
```

```
    union { ←────────────────────────────────── Это безымянное объединение
        long l;
        double d;
        char s[4];
    } ;
```

```
    // теперь обратимся к элементам объединения непосредственно
```

```
    l = 100000; ←────────────────────────────────┐
    cout << l << " ";                             │
    d = 123.2342; ←────────────────────────────────┐ Обращение к элементам безымянного объе-
    cout << d << " ";                             │ динения осуществляется непосредственно
    strcpy(s, "hi"); ←────────────────────────────────┐
    cout << s;
```

```
return 0;
}
```

Вы видите, что обращение к элементам объединения осуществляется, как если бы они были объявлены обычными локальными переменными. Собственно, в своей программе именно так вы и будете их использовать. Далее, несмотря на то, что они определены в объявлении **union**, они находятся в той же области видимости, что и остальные локальные переменные в том же блоке. Отсюда следует, что имена членов безымянного объединения не должны конфликтовать с другими идентификаторами, известными в той же области видимости.

Все ограничения обычных объединений относятся так же и к безымянным, со следующими добавлениями. Во-первых, элементами бе-

зымянного объединения могут быть только данные; функции-члены не допускаются. Безымянные объединения не могут содержать элементы **private** и **protected**. (Описатель **protected** будет обсуждаться в Модуле 10.) Наконец, глобальные безымянные объединения должны быть объявлены, как **static**.

Цель

9.8. Ключевое слово **this**

Перед тем как перейти к перегрузке, необходимо описать еще одно ключевое слово C++: **this**. Каждый раз, когда активизируется функция-член, она автоматически получает указатель с именем **this** на объект, для которого она вызвана. Указатель **this** является *неявным* параметром всех функций-членов. Таким образом, внутри функции-члена **this** может быть использован для обращения к данному объекту.

Как вы знаете, функция-член может непосредственно обращаться к закрытым данным своего класса. Например, если описан такой класс:

```
class Test {
    int i;
    void f() { ... };
    // ...
};
```

то внутри функции **f()** для присваивания переменной **i** значения 10 может быть использовано такое предложение:

```
i = 10;
```

Фактически, однако, это предложение является сокращенным вариантом следующего:

```
this->i = 10;
```

Чтобы освоиться с указателем **this**, рассмотрите следующую короткую программу:

```
// Используем указатель "this".

#include <iostream>
using namespace std;

class Test {
    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()
{
    Test o;

    o.load_i(100);
    cout << o.get_i();

    return 0;
}
```

Эта программа выводит число 100. Пример, конечно, тривиален, и никому не придет в голову использовать указатель **this** таким образом. Вскоре, однако, вы увидите, почему указатель **this** так важен в программировании на C++.

Еще одно замечание. Дружественные функции не получают указателя **this**, потому что они не являются членами класса. Указатель **this** доступен только функциям-членам.

Минутная тренировка

1. Может ли **struct** содержать функции-члены?
 2. Что является определяющей характеристикой структуры **union**?
 3. На что ссылается **this**?
1. Да, **struct** может содержать функции-члены
 2. Данные-члены структуры **union** занимают одну и ту же память
 3. **this** является указателем на объект, для которого была вызвана функция-член
-

Цель

9.9. Перегрузка операторов

Оставшаяся часть этого модуля посвящена исследованию одного из самых замечательных и мощных средств C++: *перегрузке операторов*. В C++ операторы могут быть перегружены относительно создаваемого вами классового типа. Принципиальная ценность перегруженных опе-

раторов заключается в том, что они позволяют вам эффективно интегрировать новые типы данных в вашу программную среду.

Когда вы перегружаете оператор, вы определяете смысл этого оператора для конкретного класса. Например, класс, который определяет связный список, мог бы использовать оператор `+` для добавления в список новых объектов. Класс, реализующий стек, мог бы использовать оператор `+` для проталкивания объекта в стек. Другой класс мог бы использовать тот же оператор `+` для совершенно других целей. Когда оператор перегружается, его исходный смысл не теряется. Просто относительно конкретного класса возникает новая операция, обозначаемая этим оператором. Поэтому перегрузка оператора `+`, скажем, для работы со связным списком, никак не затронет его назначения от-носительно целых чисел (где он выполняет операцию сложения).

Перегрузка операторов тесно связана с перегрузкой функций. Для перегрузки оператора вы должны определить, что означает эта операция относительно того класса, где осуществляется перегрузка. Для этого вы создаете функцию **operator** (операторную функцию). Общая форма функции **operator** выглядит так:

```
тип имя-класса::operator#(список-аргументов)
{
    //операции
}
```

Здесь перегружаемый оператор должен заместить знак `#`, а *тип* — это тип значения, возвращаемого определяемой операцией. Хотя перегруженный оператор может возвращать любой тип по вашему желанию, однако часто возвращаемое оператором значение имеет тот же тип, что и класс, для которого оператор перегружается. В этом случае обеспечивается использование перегруженного оператора в составных выражениях. Конкретная природа списка аргументов определяется несколькими факторами, описанными в последующих подразделах.

Операторные функции могут быть как членами, так и не членами класса. Однако операторные функции-не члены класса часто объявляются друзьями класса. Перегрузка операторных функций, являющихся и не являющихся членами класса, выполняется схоже, однако с некоторыми различиями, которые будут описаны ниже.



Замечание

Большое количество операторов в C++ делает тему перегрузки операторов весьма объемной, и в этой книге нет возможности изложить все ее аспекты. Желаящих познакомиться с исчерпывающим описанием перегрузки операторов адресую к моей книге: *C++: The Complete Reference*, Osborne/McGraw-Hill.

Цель

9.10. Перегрузка операторов с использованием функций-членов

Начнем изучение операторных функций-членов с простого примера. Приведенная ниже программа создает класс **ThreeD**, который обслуживает координаты объекта в трехмерном пространстве. Программа перегружает операторы `+` и `=` для этого класса. Давайте рассмотрим ее в деталях.

```
// Определение + и = для класса ThreeD.

#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // Трехмерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    ThreeD operator+(ThreeD op2); // op1 подразумевается
    ThreeD operator=(ThreeD op2); // op1 подразумевается

void show() ;
};

// Перегрузим +.
ThreeD ThreeD::operator+(ThreeD op2) ← Перегрузка + для ThreeD
{
    ThreeD temp;

    temp.x = x + op2.x; // Это сложение целых чисел
    temp.y = y + op2.y; // и по отношению к ним знак + сохраняет
    temp.z = z + op2.z; // свое первоначальное назначение.
    return temp; ← Возврат нового объекта. Аргументы остались без изменений
}

// Перегрузим присваивание.
ThreeD ThreeD::operator=(ThreeD op2) ← Перегрузка = для ThreeD
{
    x = op2.x; // Это присваивание целых чисел
    y = op2.y; // и по отношению к ним знак = сохраняет
    z = op2.z; // свое первоначальное назначение.
```

```
return *this; ← Возврат модифицированного объекта.
}

// Вывести координаты X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3), b(10, 10, 10), c;
    cout << "Исходное значение a: ";
    a.show();
    cout << " Исходное значение b: ";
    b.show();

    cout << "\n";

    c = a + b; // сложим a и b
    cout << "Значение c после c = a + b: ";
    c.show();

    cout << "\n";

    c = a + b + c; // сложим a, b и c
    cout << "Значение c после c = a + b + c: ";
    c.show();

    cout << "\n";

    c = b = a; // демонстрация множественного присваивания
    cout << "Значение c после c = b = a: ";
    c.show();
    cout << "Значение b после c = b = a: ";
    b.show();

    return 0;
}
```

Программа выводит на экран следующее:

```
Исходное значение a: 1, 2, 3
Исходное значение b: 10, 10, 10
```

Значение `c` после `c = a + b`: 11, 12, 13

Значение `c` после `c = a + b + c`: 22, 24, 26

Значение `c` после `c = b = a`: 1, 2, 3

Значение `b` после `c = b = a`: 1, 2, 3

Изучая текст программы, вы могли удивиться тому, что обе операторные функции имеют только по одному параметру, хотя они перегружают бинарные операции. Причина этого явного противоречия заключается в том, что когда бинарный оператор перегружается с помощью функции-члена, явным образом в нее передается только один аргумент. Другой аргумент передается неявным образом посредством указателя `this`. Таким образом, в строке

```
temp.x = x + op2.x;
```

`x` обозначает `this->x`, т. е. переменную `x` того объекта, для которого вызвана операторная функция. Во всех случаях операторная функция вызывается для объекта, указанного с левой стороны знака операции. Объект с правой стороны передается функции в качестве аргумента.

В общем случае операторная функция не нуждается в параметрах при перегрузке унарного оператора и требует указания только одного параметра при перегрузке бинарного оператора. (Вы не можете перегрузить тернарный оператор ?.) В любом случае объект, для которого вызывается операторная функция, неявным образом передается в функцию посредством указателя `this`.

Чтобы уяснить себе, как работает перегрузка операторов, рассмотрим последнюю программу детально, начиная с перегруженного оператора `+`. Когда над двумя объектами типа `ThreeD` выполняется операция `+`, значения их соответствующих координат складываются, как это видно из текста функции `operator+()`. Заметьте, однако, что эта функция не изменяет значений ни одного из операндов. Вместо этого она возвращает объект типа `ThreeD`, содержащий результат операции. Чтобы понять, почему операция `+` не изменяет содержимого обоих объектов, подумайте, как выполняется стандартная арифметическая операция `+` в выражении `10 + 12`. Результат этой операции 22, однако ни 10, ни 12 не изменились. Хотя и нет определенного правила, запрещающего перегруженному оператору `+` изменять значение одного из его операндов, лучше, чтобы действия перегруженного оператора соответствовали его первоначальному назначению.

Обратите внимание на то, что `operator+()` возвращает объект типа `ThreeD`. Хотя функция в принципе могла бы возвращать любой допустимый в C++ тип, возврат объекта типа `ThreeD` позволяет использовать оператор `+` в составных выражениях вроде `a+b+c`. В этом выражении сложение `a+b` дает результат типа `ThreeD`. Это значение затем прибавляется к `c`. Если бы сложение `a+b` давало результат любого другого типа, составное выражение не могло бы выполняться должным образом.

В противоположность оператору `+`, оператор присваивания модифицирует один из своих аргументов. (В этом, впрочем, и состоит сущность присваивания.) Поскольку функция `operator=()` вызывается для объекта, который указывается с левой стороны знака `=`, то именно этот объект модифицируется операцией присваивания. Чаще всего возвращаемое значение перегруженного оператора присваивания после того, как присваивание выполнено, становится объектом с левой стороны знака `=`. (Это соответствует традиционному действию оператора `=`.) Например, для того, чтобы можно было написать предложение вроде

```
a = b = c = d;
```

необходимо, чтобы `operator=()` возвращал объект, на который указывает `this`, а это как раз объект, указанный с левой стороны предложения присваивания. В результате возникает возможность выполнять присваивание по цепочке. Использование указателя `this` в операции присваивания является одним из его важнейших применений.

В рассматриваемой программе в сущности не было необходимости в перегрузке операции `=`, поскольку оператор присваивания по умолчанию, предоставляемый C++, вполне годится для класса **ThreeD**. (Как уже отмечалось ранее, операция присваивания по умолчанию представляет собой побитовое копирование.) В нашем примере оператор `=` был перегружен просто чтобы продемонстрировать процедуру перегрузки. В принципе вам надо будет перегружать `=` только в тех случаях, когда действующее по умолчанию побитовое копирование даст неправильный результат. Поскольку оператор `=` по умолчанию работает удовлетворительно для класса **ThreeD**, в последующих примерах этого модуля он перегружаться не будет.

Другие вопросы

Перегружая бинарные операторы, имейте в виду, что во многих случаях имеет значение порядок операндов. Так, хотя `A + B` представляет собой коммутативную операцию, `A - B` таковой не является. (`A - B` не то же самое, что `B - A`!) Поэтому, реализуя перегруженные варианты некоммутативных операторов, помните, какой операнд стоит с левой стороны, и какой с правой. Например, вот как надо перегрузить минус для класса **ThreeD**:

```
// Перегрузка вычитания.
```

```
ThreeD ThreeD::operator-(ThreeD op2)
{
    ThreeD temp;
```

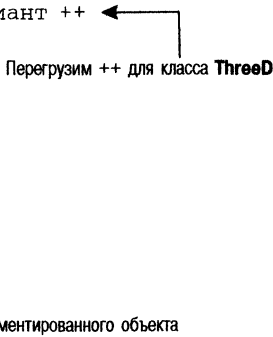
```
temp.x = x - op2.x;  
temp.y = y - op2.y;  
temp.z = z - op2.z;  
  
return temp;  
}
```

Не забывайте, что вызывает операторную функцию операнд, стоящий с левой стороны знака операции. Правый операнд передается явным образом.

Использование функций-членов для перегрузки унарных операторов

Унарные операторы, такие как ++, -- или унарные - и +, также могут быть перегружены. Как уже отмечалось выше, если унарный оператор перегружается посредством функции-члена, операторной функции явным образом не передаются никакие объекты. Просто операция выполняется над объектом, который вызывает функцию посредством неявно передаваемого указателя **this**. Ниже приведена для примера программа, которая определяет операцию инкремента для объектов типа **ThreeD**:

```
// Перегрузка унарного оператора ++.  
  
#include <iostream>  
using namespace std;  
  
class ThreeD {  
    int x, y, z; // Трехмерные координаты  
public:  
    ThreeD() { x = y = z = 0; }  
    ThreeD(int i, int j, int k) {x = i; y = j; z = k; }  
  
    ThreeD operator++(); // префиксный вариант ++  
  
    void show() ;  
};  
  
// Перегрузим префиксный вариант ++.  
ThreeD ThreeD::operator++()  
{  
    x++; // инкремент x, y и z  
    y++;  
    z++;  
    return *this; }  
  
// Возврат инкрементированного объекта
```



```
// Выведем координаты X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3);

    cout << "Исходное значение a: ";
    a.show();

    ++a; // инкремент a
    cout << "Значение после ++a: ";
    a.show();

    return 0;
}
```

Вывод этой программы выглядит таким образом:

```
Исходное значение a: 1, 2, 3
Значение после ++a: 2, 3, 4
```

Как это удостоверяет вывод программы, **operator++()** выполняет инкремент каждой координаты в объекте и возвращает модифицированный объект. Как и раньше, эта процедура соответствует традиционному пониманию смысла оператора ++.

Как вы знаете, операторы ++ и -- имеют префиксные и постфиксные формы. Например, предложения

```
++x;

и

x++;
```

являются допустимыми применениями оператора инкремента. Как было указано в комментариях в последней программе, функция **operator++()** определяет префиксную форму ++ для класса **ThreeD**. Однако, постфиксная форма этого оператора также может быть перегружена. Прототип постфиксной формы оператора ++ для класса **ThreeD** выглядит так:

```
ThreeD operator++(int notused);
```

Параметр **notused** не используется функцией и на него не следует обращать внимания. Этот параметр просто предоставляет способ для компилятора различить префиксную и постфиксную формы оператора инкремента. (Постфиксный декремент использует тот же подход.)

Вот один из способов реализации постфиксного варианта операции ++ для класса **ThreeD**:

```
// Перегрузим постфиксный вариант ++.
ThreeD ThreeD::operator++(int notused)  ← Обратите внимание на
{                                         параметр notused
    ThreeD temp = *this; // сохраним исходное значение

    x++; // инкремент x, y и z
    y++;
    z++;
    return temp; // вернем исходное значение
}
```

Обратите внимание на сохранение этой функцией исходного состояния операнда с помощью предложения

```
ThreeD temp = *this;
```

и возврата, перед завершением, объекта **temp**. Обычный смысл постфиксного инкремента заключается в том, что сначала определяется значение операнда, и лишь затем выполняется его инкремент. Следовательно, перед тем, как исходное значение операнда будет инкрементировано, необходимо сохранить это исходное значение и использовать в качестве возвращаемого значения именно его, а не модифицированное значение.

Приведенная ниже программа реализует обе формы оператора ++:

```
// Демонстрация префиксного и постфиксного ++.

#include <iostream>
using namespace std;

class ThreeD {
    int x, y, z; // Трехмерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) {x = i; y = j; z = k; }

    ThreeD operator++(); // префиксный вариант ++
    ThreeD operator++(int notused); // постфиксный вариант ++

    void show() ;
};
```

```
// Перегрузим префиксный вариант ++.
ThreeD ThreeD::operator++()
{
    x++; // инкремент x, y и z
    y++;
    z++;
    return *this; // вернем измененное значение
}

// Перегрузим постфиксный вариант ++.
ThreeD ThreeD::operator++(int notused)
{
    ThreeD temp = *this; // сохраним исходное значение

    x++; // инкремент x, y и z
    y++;
    z++;
    return temp; // вернем исходное значение
}

// Выведем координаты X, Y, Z.
void ThreeD::show( )
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3);
    ThreeD b;

    cout << "Исходное значение a: ";
    a.show();

    cout << "\n";

    ++a; // префиксный инкремент ←———— Вызов функции префиксного инкремента
    cout << "Значение после ++a: ";
    a.show();

    a++; // постфиксный инкремент ←———— Вызов функции постфиксного инкремента
    cout << "Значение после a++: ";
    a.show();

    cout << "\n";
}
```

```
b = ++a; // b получает значение a после инкремента
cout << "Значение a после b = ++a: ";
a.show();
cout << "Значение b после b = ++a: ";
b.show();

cout << "\n";

b = a++; // b получает значение a перед инкрементом
cout << "Значение a после b = a++: ";
a.show();
cout << "Значение b после b = a++: ";
b.show();

return 0;
}
```

Вывод этой программы выглядит следующим образом:

Исходное значение a: 1, 2, 3

Значение после ++a: 2, 3, 4

Значение после a++: 3, 4, 5

Значение a после b = ++a: 4, 5, 6

Значение b после b = ++a: 4, 5, 6

Значение a после b = a++: 5, 6, 7

Значение b после b = a++: 4, 5, 6

Не забывайте, что если знак ++ указан перед своим операндом, вызывается **operator++()**. Если он указан после своего операнда, вызывается **operator++(int notused)**. Тот же самый подход используется для перегрузки префиксного или постфиксного оператора декремента для любого класса. Вы можете в качестве упражнения попробовать определить оператор декремента для **ThreeD**.

Любопытно отметить, что ранние версии C++ не делали различия между префиксной и постфиксной формами операторов инкремента и декремента. В этих старых версиях для обоих вариантов оператора вызывалась префиксная форма операторной функции. Работая со старыми C++-программами, имейте в виду это обстоятельство.

Минутная тренировка

1. Операторы должны перегружаться для каждого класса отдельно. Правильно ли это?

2. Сколько параметров имеет операторная функция-член для бинарных операторов?
3. При использовании бинарной операторной функции-члена левый операнд передается в функцию посредством _____.
1. Правильно, оператор должен перегружаться для конкретного класса
2. Бинарная операторная функция-член имеет один параметр, который принимает правый операнд
3. **this**

Цель

9.11. Операторные функции-не члены

Вы можете перегрузить оператор для конкретного класса, используя функцию, не являющуюся членом этого класса. Часто эта функция определяется, как дружественная этому классу. Как вы уже знаете, дружественные функции не имеют указателя **this**. Следовательно, если дружественная функция используется для перегрузки оператора, то для бинарного оператора в нее должны передаваться явным образом оба операнда, а при перегрузке унарного оператора — один операнд. С помощью дружественных функций могут быть перегружены все операторы, за исключением **=**, **()**, **[]** и **->**.

Приведенная ниже программа использует для перегрузки операции **+** в классе **ThreeD** вместо функции-члена дружественную функцию:

// Использование дружественной операторной функции.

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

```
class ThreeD {
    int x, y, z; // Трехмерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
```

```
    friend ThreeD operator+(ThreeD op1, ThreeD op2);
    void show();
};
```

Здесь функция **operator+()** является другом **ThreeD**. Заметьте, что ей требуются два параметра

// **+** теперь является дружественной функцией.

```
ThreeD operator+(ThreeD op1, ThreeD op2)
{
```

```
    ThreeD temp;
    temp.x = op1.x + op2.x;
    temp.y = op1.y + op2.y;
    temp.z = op1.z + op2.z;
    return temp;
}

// Выведем координаты X, Y, Z.
void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3), b(10, 10, 10), c;

    cout << "Исходное значение a: ";
    a.show();
    cout << " Исходное значение b: ";
    b.show();

    cout << "\n";

    c = a + b; // сложим a и b
    cout << "Значение c после c = a + b: ";
    c.show();

    cout << "\n";

    c = a + b + c; // сложим a, b и c
    cout << "Значение c после c = a + b + c: ";
    c.show();

    cout << "\n";

    c = b = a; // демонстрация множественного присваивания
    cout << "Значение c после c = b = a: ";
    c.show();
    cout << "Значение b после c = b = a: ";
    b.show();

    return 0;
}
```


Вывод этой программы выглядит таким образом:

Исходное значение a: 1, 2, 3

Исходное значение b: 10, 10, 10

Значение c после $c = a + b$: 11, 12, 13

Значение c после $c = a + b + c$: 22, 24, 26

Значение c после $c = b = a$: 1, 2, 3

Значение b после $c = b = a$: 1, 2, 3

Вы видите, что теперь в функцию `operator+()` передаются оба операнда. Левый операнд передается в `op1`, правый — в `op2`.

Во многих случаях использование для перегрузки оператора дружественной функции вместо функции-члена не дает никаких преимуществ. Однако, имеется одна ситуация, в которой дружественная функция оказывается весьма полезной: если вы хотите, чтобы с левой стороны бинарной операции был указан объект встроенного типа. Для того, чтобы понять, почему это так, поразмыслите над следующим. Как вы знаете, указатель на объект, активизирующий операторную функцию-член, передается через `this`. В случае бинарного оператора функция активизируется объектом, стоящим слева. Если для объекта, стоящего слева, интересующая нас операция определена, то все хорошо. Пусть, например, для некоторого объекта `T` определены операции присваивания и целочисленного сложения. Тогда приведенное ниже предложение

```
T = T + 10; // будет работать
```

вполне допустимо. Поскольку объект `T` указан слева от оператора `+`, он активизирует перегруженную операторную функцию, которая (очевидно) умеет прибавлять целочисленное значение к некоторому элементу `T`. Однако, следующее предложение работать не будет:

```
T = 10 + T; // не будет работать
```

Проблема здесь в том, что объект, указанный слева от оператора `+`, является переменной типа `int`, встроенным типом, для которого не определены операции над целым числом и объектами типа, которому принадлежит объект `T`.

Решение поставленной проблемы заключается в перегрузке `+` с использованием двух дружественных функций. В этом случае операторной функции явным образом передаются оба аргумента, и она активизируется, как и любая другая перегруженная функция, исходя из типов ее аргументов. Один вариант операторной функции `+` выполняет операции *объект + целое*, другой — *целое + объект*. Перегрузка `+` (или любого другого бинарного оператора) с помощью дружественных функций

позволяет помещать встроенный тип как с левой, так и с правой стороны оператора. Приведенная ниже программа иллюстрирует эту технику. Она определяет два варианта функции **operator+()** для объектов типа **ThreeD**. Оба прибавляют целочисленное значение к каждой из переменных экземпляра класса **ThreeD**. Целочисленная переменная может быть указана как с левой, так и с правой стороны оператора.

```
// Перегрузка int + объект и объект + int.
```

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

```
class ThreeD {
    int x, y, z; // Трехмерные координаты
public:
    ThreeD() { x = y = z = 0; }
    ThreeD(int i, int j, int k) { x = i; y = j; z = k; }

    friend ThreeD operator+(ThreeD op1, int op2);
    friend ThreeD operator+(int op1, ThreeD op2);

    void show() ;
};
```

Эти функции обеспечивают операции объект + int и int + объект

```
// Эта функция обеспечивает ThreeD + int
ThreeD operator+(ThreeD op1, int op2)
{
```

```
    ThreeD temp;

    temp.x = op1.x + op2;
    temp.y = op1.y + op2;
    temp.z = op1.z + op2;
    return temp;
}
```

```
// Эта функция обеспечивает int + ThreeD
ThreeD operator+(int op1, ThreeD op2)
```

```
{
    ThreeD temp;

    temp.x = op2.x + op1;
    temp.y = op2.y + op1;
    temp.z = op2.z + op1;
    return temp;
}
```

```
// Выведем координаты X, Y, Z.
```

```

void ThreeD::show()
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

int main()
{
    ThreeD a(1, 2, 3), b;

    cout << "Исходное значение a: ";
    a.show();

    cout << "\n";

    b = a + 10; // объект + int
    cout << "Значение b после b = a + 10: ";
    b.show();

    cout << "\n";

    b = 10 + a; // int + объект ← Здесь встроенный тип указан с
    cout << "Значение b после b = 10 + a: ";      левой стороны оператора +
    b.show();

    return 0;
}

```

Вывод программы выглядит так:

Исходное значение of a: 1, 2, 3

Значение b после b = a + 10: 11, 12, 13

Значение b после b = 10 + a: 11, 12, 13

Поскольку функция **operator+()** перегружена дважды, она обеспечивает оба варианта размещения целочисленной переменной и объекта типа **ThreeD** в операциях сложения.

Использование дружественной функции для перегрузки унарного оператора

Унарный оператор тоже можно перегрузить с помощью дружественной функции. Однако, если вы перегружаете ++ или --, вы должны передавать операнды в функцию в качестве параметров-ссылок.

Поскольку параметр-ссылка представляет собой неявный указатель на аргумент, изменения параметра приведут к изменениям аргумента. Использование параметра-ссылки позволит функции выполнять инкремент или декремент объекта, указанного в качестве операнда.

Если для перегрузки операторов инкремента или декремента используется дружественная функция, префиксная форма принимает один параметр (который является операндом). Постфиксная форма принимает два параметра. Второй параметр имеет тип `int` и не используется.

Вот как можно перегрузить обе формы дружественной функции `operator++()` для класса `ThreeD`:

```
/* Перегрузка префиксной операции ++ с помощью дружественной
   функции.
   Такая перегрузка требует использования параметра-ссылки.
*/

ThreeD operator++(ThreeD &op1)
{
    op1.x++;
    op1.y++;
    op1.z++;
    return op1;
}

/* Перегрузка постфиксной операции ++ с помощью дружественной
   функции.
   Такая перегрузка требует использования параметра-ссылки.
*/

ThreeD operator++(ThreeD &op1, int notused)
{
    ThreeD temp = op1;

    op1.x++;
    op1.y++;
    op1.z++;
    return temp;
}
```

Спросим у эксперта

Вопрос: Возникают ли какие-либо специфические проблемы при перегрузке операторов отношения?

Ответ: Перегрузка оператора отношения, например, `==` или `<`, выполняется обычным образом. Однако, имеется одна сложность. Как вы знаете, пере-

груженная операторная функция часто возвращает объект класса, для которого она перегружена. Однако перегруженные операторы отношения должны возвращать **true** или **false**. В этом случае они будут работать так же, как и обычные операторы отношения, и их можно будет использовать в условных выражениях. Те же рассуждения справедливы и для логических операторов.

Чтобы показать вам, как можно реализовать перегруженный оператор отношения, приведем функцию, перегружающую оператор `==` для класса **ThreeD**:

```
// Перегрузка ==.
bool ThreeD::operator==(ThreeD op2)
{
    if((x == op2.x) && (y == op2.y) && (z == op2.z))
        return true;
    else
        return false;
}
```

Если в программе реализована операторная функция `operator==()`, то следующий фрагмент вполне допустим:

```
ThreeD a(1, 1, 1), b(2, 2, 2);
// ...
if(a == b) cout << "a равно b\n";
else cout << "a не равно b\n";
```

Минутная тренировка

1. Сколько параметров имеет бинарная операторная функция-не член?
 2. Как следует передавать параметр при использовании операторной функции не-члена для перегрузки оператора `++`?
 3. Одно из преимуществ использования дружественных операторных функций заключается в том, что они позволяют использовать с левой стороны операнда встроенный тип (например, `int`). Справедливо ли это утверждение?
-
1. Бинарная операторная функция-не член имеет два параметра
 2. При использовании операторной функции не-члена для перегрузки оператора `++` операнд должен передаваться по ссылке
 3. Справедливо, использование дружественных операторных функций позволяет использовать встроенный тип (например, `int`) с левой стороны операнда.
-

Советы и ограничения при перегрузке операторов

Действие перегруженного оператора применительно к классу, для которого он определен, может не иметь никакого отношения к его

использованию по умолчанию, когда он применяется к встроенным типам C++. Например, операторы << и >>, когда они используются с объектами `cout` и `cin`, имеют мало общего с теми же операторами, используемыми с целочисленными переменными. Однако, ради ясности и удобства чтения вашей программы перегруженный оператор должен, если это возможно, отражать дух своего традиционного использования. Например, оператор `+`, перегруженный для класса `ThreeD`, концептуально схож с тем же оператором, используемым с целочисленными переменными. Было бы мало смысла определить оператор `+` для некоторого класса таким образом, чтобы он выполнял действия, которые вы ожидаете от, например, оператора `||`. Наша основная мысль заключается в том, что хотя вы можете придать перегруженному оператору любой смысл, какой вы только пожелаете, ради ясности программы лучше всего, если новое значение оператора схоже с его исходным назначением.

При перегрузке операторов возникают некоторые ограничения. Во-первых, вы не можете изменить относительный приоритет оператора. Во-вторых, вы не можете изменить число операндов, требуемых оператором, хотя, с другой стороны, ваша операторная функция может игнорировать операнд. Наконец, за исключением оператора вызова функции, операторные функции не могут иметь аргументов по умолчанию.

Почти все операторы C++ могут быть перегружены. Это относится и к таким специальным операторам, как оператор индексирования массива `[]`, оператор вызова функции `()`, оператор `->`. Всего четыре оператора не могут быть перегружены. Это:

`:: * ?`

Оператор `*` имеет особое назначение, и его использование выходит за рамки данной книги.

Проект 9-1

Создание класса, определяющего множество

Перегрузка операторов помогает вам создавать классы, которые допускают полную интеграцию в программную среду C++. Подумайте о следующем: определив необходимые операторы, вы можете использовать любой классовый тип в программе в точности так же, как используете встроенные типы. Вы можете воздействовать на объекты этого класса посредством операторов и использовать объекты этого класса в выражениях. Для иллюстрации создания и интеграции нового класса в

среди C++ в этом проекте создается класс **Set**, который определяет новый тип множества.

Перед тем, как мы приступим к разработке класса, важно точно определить, что мы будем понимать под множеством. В плане данного проекта мы будем считать множеством коллекцию уникальных элементов. Это значит, что никакие два элемента в данном множестве не могут быть одинаковыми. Порядок членов множества значения не имеет. Так, множество

{ A, B, C }

совпадает с множеством

{ A, C, B }

Множество может быть также пустым.

Множество поддерживает ряд операций. Мы реализуем следующие:

- Прибавление элемента к множеству
- Удаление элемента из множества
- Объединение множеств
- Получение разности множеств

Добавление элемента к множеству и удаление элемента из множества не требуют пояснений. Оставшиеся операции не столь очевидны.

Объединением двух множеств является множество, содержащее все элементы из обоих множеств. (Разумеется, дублирование элементов недопустимо.) Для выполнения операции объединения множеств мы воспользуемся оператором +.

Разностью двух множеств является множество, содержащее те элементы первого множества, которые не являются частью второго множества. Для выполнения операции разности над множествами мы будем использовать оператор —. Например, если даны два множества S1 и S2, тогда следующее предложение удаляет из S1 все элементы S2, помещая результат в S3:

$S3 = S1 - S2;$

Если S1 и S2 совпадают, тогда S3 будет пустым множеством.

Класс **Set** будет также включать функцию **IsMember()**, которая определяет, является ли указанный элемент членом данного множества.

Разумеется, над множествами можно выполнять и другие операции. Некоторые из них вы сможете реализовать, отвечая на Вопросы для самопроверки. Другие вы можете попытаться разработать самостоятельно.

Ради простоты наш класс **Set** будет содержать множество символов, однако те же самые общие принципы можно использовать для создания класса **Set**, предназначенного для хранения элементов других типов.

Шаг за шагом

1. Создайте файл с именем **Set.cpp**.
2. Начните создавать **Set** с разработки объявления этого класса, как показано ниже:

```
const int MaxSize = 100;

class Set {
    int len; // число членов
    char members[MaxSize]; // этот массив будет содержать
                           // множество

    /* Функция find() закрыта, потому что она не используется
       вне класса Set. */
    int Set::find(char ch); // найти элемент

public:
    // Конструируем пустое множество.
    Set() { len = 0; }

    // Возвращает число элементов множества.
    int getLength() { return len; }

    void showset(); // выводит множество на экран
    bool isMember(char ch); // проверяет на членство

    Set operator +(char ch); // добавляет элемент
    Set operator -(char ch); // удаляет элемент

    Set operator +(Set ob2); // образует объединение
    Set operator -(Set ob2); // образует разность
};
```

Каждое множество хранится в массиве **char** с именем **members**. Число членов, фактически входящих в множество, хранится в переменной **len**. Максимальный размер множества ограничен величиной **MaxSize**, которая имеет значение 100. (Вы можете увеличить это значение, если хотите работать с большими множествами.)

Конструктор **Set** создает *пустое множество*, т. е. множество, в котором нет членов. Создавать какие-либо еще конструкторы нет необходимости, как и определять явный конструктор копий для класса **Set**, поскольку побитовое копирование по умолчанию в данном случае вполне приемлемо. Функция **getLength()** возвращает значение **len**, которое представляет собой число элементов, находящихся в множестве в данный момент.

3. Начнем определять функции-члены, начав с закрытой функции **find()**, показанной ниже:

```
/* Возвращает индекс элемента, задаваемого
   параметром ch, или -1, если элемент не найден. */
int Set::find(char ch) {
    int i;

    for(i=0; i < len; i++)
        if(members[i] == ch) return i;

    return -1;
}
```

Эта функция определяет, является ли элемент, переданный в качестве параметра **ch**, членом множества. Она возвращает индекс найденного элемента или **-1**, если заданный элемент не входит в множество. Эта функция закрыта, так как она не используется вне класса **Set**. Как было объяснено ранее, функции-члены могут быть закрытыми в своем классе. Закрытая функция-член может быть вызвана только другими функциями-членами данного класса.

4. Добавьте функцию **showset()**, текст которой приведен ниже:

```
// Вывести множество на экран.
void Set::showset() {
    cout << "{ ";
    for(int i=0; i<len; i++)
        cout << members[i] << " ";

    cout << "}\n";
}
```

Эта функция выводит на экран содержимое множества.

5. Добавьте функцию **IsMember()**, которая определяет, является ли указанный элемент членом множества. Текст функции приведен ниже:

```
/* Возвращает true, если ch есть член множества.
   В противном случае возвращает false. */
bool Set::isMember(char ch) {
    if(find(ch) != -1) return true;
    return false;
}
```

Эта функция вызывает **find()** для определения того, является ли **ch** членом множества. Если является, **IsMember()** возвращает **true**; в противном случае она возвращает **false**.

6. Теперь начните включать в программу операторы для работы с множеством, начав с оператора `+`. Для этого следует перегрузить `+` для объектов типа **Set**, как это показано ниже. Этот вариант добавляет элементы в множество.

```
// Добавление уникального элемента в множество.
Set Set::operator +(char ch) {
    Set newset;

    if(len == MaxSize) {
        cout << "Множество полно.\n";
        return *this; // вернуть существующее множество
    }

    newset = *this; // дублируем существующее множество

    // смотрим, не существует ли уже этот элемент
    if(find(ch) == -1) { // если не найден, можно добавить
        // добавим новый элемент в множество
        newset.members[newset.len] = ch;
        newset.len++;
    }
    return newset; // возвращаем обновленное множество
}
```

Эта функция достойна более внимательного рассмотрения. Прежде всего создается новое множество, которое содержит элементы существующего плюс символ, полученный в **ch**. Перед тем, как символ в **ch** будет добавлен, выполняется проверка, есть ли в множестве свободное место для помещения в него нового элемента. Если для нового элемента место есть, исходное множество присваивается множеству **newset**. Далее вызывается функция **find()** для определения, нет ли уже в множестве символа **ch**. В любом случае возвращается множество **newset**. Таким образом, исходное множество в результате этой операции остается без изменений.

7. Перегрузите `-`, чтобы можно было удалять элемент из множества:

```
// Удаляет элемент из множества.
Set Set::operator -(char ch) {
    Set newset;
    int i = find(ch); // i будет равно -1, если элемент не найден

    // копирование и уплотнение оставшихся элементов
    for(int j=0; j < len; j++)
        if(j != i) newset = newset + members[j];
}
```

```
    return newset;
}
```

Эта функция начинает с создания нового пустого множества. Затем вызывается `find()`, чтобы определить индекс элемента `ch` в исходном множестве. Вспомним, что `find()` возвращает `-1`, если `ch` не является членом. Далее элементы исходного множества добавляются к новому множеству за исключением элемента, индекс которого равен индексу, возвращенному функцией `find()`. В результате образуемое множество содержит все элементы исходного за исключением `ch`. Если `ch` не входил в исходное множество, тогда оба множества будут эквивалентны.

8. Еще раз перегрузите `+` и `-`, как это показано ниже. Эти варианты реализуют объединение и разность множеств:

```
// Объединение множеств.
Set Set::operator +(Set ob2) {
    Set newset = *this; // копируем первое множество

    // Добавляем уникальные элементы из второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset + ob2.members[i];

    return newset; // возвращаем обновленное множество
}

// Разность множеств.
Set Set::operator -(Set ob2) {
    Set newset = *this; // копируем первое множество

    // Вычитаем элементы второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset - ob2.members[i];

    return newset; // возвращаем обновленное множество
}
```

Легко заметить, что эти функции при выполнении своих операций используют предварительно определенные варианты операторов `+` и `-`. В случае объединения создается новое множество, которое содержит элементы первого множества. Затем к нему добавляются элементы второго множества. Поскольку оператор `+` добавляет элемент только в том случае, если его еще нет в множестве, результирующее множество является объединением (без дублирования) двух исходных. Оператор разности множеств вычитает совпадающие элементы.

9. Ниже приведен полный текст класса `Set` вместе с функцией `main()`, демонстрирующей работу с ним:

```
/*
    Проект 9-1

    Класс множества для символов.
*/
#include <iostream>
using namespace std;

const int MaxSize = 100;

class Set {
    int len; // число членов
    char members[MaxSize]; // этот массив будет содержать
                           // множество

    /* Функция find() закрыта, потому что она не используется
       вне класса Set. */
    int Set::find(char ch); // найти элемент

public:
    // Конструируем пустое множество.
    Set() { len = 0; }

    // Возвращает число элементов множества.
    int getLength() { return len; }

    void showset(); // выводит множество на экран
    bool isMember(char ch); // проверяет на членство

    Set operator +(char ch); // добавляет элемент
    Set operator -(char ch); // удаляет элемент

    Set operator +(Set ob2); // образует объединение
    Set operator -(Set ob2); // образует разность
};

/* Возвращает индекс элемента, задаваемого
    параметром ch, или -1, если элемент не найден. */
int Set::find(char ch) {
    int i;

    for(i=0; i < len; i++)
        if(members[i] == ch) return i;

    return -1;
}
```

```
// Вывести множество на экран.
void Set::showset() {
    cout << "{ ";
    for(int i=0; i<len; i++)
        cout << members[i] << " ";

    cout << "}\n";
}

/* Возвращает true, если ch есть член множества.
   В противном случае возвращает false. */
bool Set::isMember(char ch) {
    if(find(ch) != -1) return true;
    return false;
}

// Добавление уникального элемента в множество.
Set Set::operator +(char ch) {
    Set newset;

    if(len == MaxSize) {
        cout << "Множество полно.\n";
        return *this; // вернуть существующее множество
    }

    newset = *this; // дублируем существующее множество

    // смотрим, не существует ли уже этот элемент
    if(find(ch) == -1) { // если не найден, можно добавить
        // добавим новый элемент в множество
        newset.members[newset.len] = ch;
        newset.len++;
    }
    return newset; // возвращаем обновленное множество
}

// Удаляет элемент из множества.
Set Set::operator -(char ch) {
    Set newset;
    int i = find(ch); // i будет равно -1, если элемент не найден

    // копирование и уплотнение оставшихся элементов
    for(int j=0; j < len; j++)
        if(j != i) newset = newset + members[j];

    return newset;
}
```

```
// Объединение множеств.
Set Set::operator +(Set ob2) {
    Set newset = *this; // копируем первое множество

    // Добавляем уникальные элементы из второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset + ob2.members[i];

    return newset; // возвращаем обновленное множество
}

// Разность множеств.
Set Set::operator -(Set ob2) {
    Set newset = *this; // копируем первое множество

    // Вычитаем элементы второго множества.
    for(int i=0; i < ob2.len; i++)
        newset = newset - ob2.members[i];

    return newset; // возвращаем обновленное множество
}

// Демонстрация класса Set.
int main() {
    // сконструируем 10-элементное пустое множество
    Set s1;
    Set s2;
    Set s3;

    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';

    cout << "s1 после добавления A B C: ";
    s1.showset();

    cout << "\n";

    cout << "Проверяем на членство, используя isMember().\n";
    if(s1.isMember('B'))
        cout << "B является членом s1.\n";
    else
        cout << "B не является членом s1.\n";

    if(s1.isMember('T'))
        cout << "T является членом s1.\n";
```

```
else
    cout << "Т не является членом s1.\n";
cout << "\n";

s1 = s1 - 'B';
cout << "s1 после s1 = s1 - 'B': ";
s1.showset();

s1 = s1 - 'A';
cout << "s1 после s1 = s1 - 'A': ";
s1.showset();

s1 = s1 - 'C';
cout << "s1 после a1 = s1 - 'C': ";
s1.showset();

cout << "\n";

s1 = s1 + 'A';
s1 = s1 + 'B';
s1 = s1 + 'C';
cout << "s1 после добавления A B C: ";
s1.showset();

cout << "\n";

s2 = s2 + 'A';
s2 = s2 + 'X';
s2 = s2 + 'W';
cout << "s2 после добавления A X W: ";
s2.showset();

cout << "\n";

s3 = s1 + s2;
cout << "s3 после s3 = s1 + s2: ";
s3.showset();

s3 = s3 - s1;
cout << "s3 после s3 - s1: ";
s3.showset();

cout << "\n";

cout << "s2 после s2 = s2 - s2: ";
s2 = s2 - s2; // очистка s2
```

```
s2.showset();

cout << "\n";

s2 = s2 + 'C'; // добавим ABC in обратном порядке
s2 = s2 + 'B';
s2 = s2 + 'A';

cout << "s2 после добавления C B A: ";
s2.showset();

return 0;
}
```

Вывод программы выглядит таким образом:

s1 после добавления A B C: { A B C }

Проверяем на членство, используя isMember().

B является членом s1.

T не является членом s1.

s1 после s1 = s1 - 'B': { A C }

s1 после s1 = s1 - 'A': { C }

s1 после s1 = s1 - 'C': { }

s1 после добавления A B C: { A B C }

s2 после добавления A X W: { A X W }

s3 после s3 = s1 + s2: { A B C X W }

s3 после s3 - s1: { X W }

s2 после s2 = s2 - s2: { }

s2 после добавления C B A: { C B A }

✓ Вопросы для самопроверки

1. Что такое конструктор копий и когда он вызывается? Приведите общую форму конструктора копий.
2. Объясните, что происходит, когда объект возвращается функцией. Конкретно, когда вызывается деструктор?

3. Дан такой класс:

```
class T {  
    int i, j;  
public:  
    int sum() {  
        return i + j;  
    }  
};
```

Покажите, как можно иначе написать функцию `sum()`, чтобы она использовала указатель `this`.

4. Что такое структура? Что такое объединение?
5. К чему относится `*this` внутри функции?
6. Что такое дружественная функция?
7. Приведите общую форму перегрузки бинарной операторной функции-члена.
8. Что вы должны сделать, чтобы могли выполняться операции, в которых участвует как классový тип, так и встроенный?
9. Можно ли перегрузить оператор-знак вопроса (`?`)? Можно ли изменить относительный приоритет оператора?
10. Для класса `Set`, разработанного в Проекте 9-1, определите операторы `<` и `>` так, чтобы они определяли, является ли одно множество подмножеством или надмножеством другого. Пусть оператор `<` возвращает `true`, если левое множество является подмножеством множества с правой стороны, и `false` в обратном случае. Пусть `>` возвращает `true`, если левое множество является надмножеством множества с правой стороны, и `false` в обратном случае.
11. Для класса `Set` определите оператор `&` так, чтобы он возвращал пересечение двух множеств.
12. Попытайтесь самостоятельно дополнить `Set` другими операторами. Например, попробуйте определить оператор `|` так, чтобы он возвращал *строгую дизъюнкцию* двух множеств. Строгая дизъюнкция состоит из всех элементов обоих множеств за исключением совпадающих.

Модуль 10 Наследование, виртуальные функции и полиморфизм

Цели, достигаемые в этом модуле

- 10 1 Познакомиться с основами наследования
- 10 2 Понять, как производные классы наследуют от базовых
- 10.3 Узнать, зачем используется защищенный доступ
- 10 4 Освоить вызов конструкторов базовых классов
- 10 5 Научиться создавать многоуровневую иерархическую структуру классов
- 10 6 Познакомиться со случаями наследования от нескольких базовых классов
- 10 7 Узнать, когда вызываются конструкторы и деструкторы, входящие в иерархию классов
- 10.8 Рассмотреть применение указателей на базовый класс в качестве указателей на производные классы
- 10 9 Научиться создавать виртуальные функции
- 10 10 Освоить использование чистых виртуальных функций и абстрактных классов

В этом модуле обсуждаются три средства C++, имеющих непосредственное отношение к объектно-ориентированному программированию: наследование, виртуальные функции и полиморфизм. Наследование — это средство, позволяющее одному классу наследовать характеристики другого. Опираясь на понятие наследования, вы можете создать базовый класс, в котором будут определены характерные черты, общие для определенного круга объектов. Эти общие черты могут наследоваться другими, более специфическими классами, каждый из которых добавит к ним более частные средства, характерные для данного производного класса. Возможности наследования находят свое применение в понятии виртуальной функции. Виртуальные функции поддерживают полиморфизм, философию “одного интерфейса, многих методов”, являющуюся одним из краеугольных камней объектно-ориентированного программирования.

Цель

10.1. Основы наследования

Согласно языку C++, наследуемый класс называется *базовым классом*. Класс, который наследует от базового, называется *производным классом*. Таким образом, производный класс является специализированным вариантом базового. Производный класс наследует все члены, определенные в базовом классе, и добавляет к ним свои собственные специфические элементы.

В C++ наследование реализуется путем включения одного класса (базового) в объявление другого (производного). Другими словами, в объявлении производного класса указывается, какой класс является для него базовым. Давайте начнем с простого примера, иллюстрирующего некоторые ключевые черты наследования. Приведенная ниже программа создает базовый класс **TwoDShape** (двумерная фигура), который содержит ширину и высоту двумерного объекта, а также производный класс **Triangle** (треугольник). Обратите особое внимание на способ объявления класса **Triangle**:

```
// Простая иерархия классов.

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

// Класс для двумерных объектов.
class TwoDShape {
public:
    double width;
    double height;
```

```
void showDim() {
    cout << "Ширина and высота составляют " <<
        width << " и " << height << "\n";
}
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

    double area() {
        return width * height / 2;
    }

    void showStyle() {
        cout << "Треугольник " << style << "\n";
    }
};

int main() {
    Triangle t1;
    Triangle t2;

    t1.width = 4.0;
    t1.height = 4.0;
    strcpy(t1.style, "равнобедренный");

    t2.width = 8.0;
    t2.height = 12.0;
    strcpy(t2.style, "прямоугольный");

    cout << "Данные о t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";
    cout << "Данные о t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Площадь равна " << t2.area() << "\n";

    return 0;
}
```

Triangle наследует от **TwoDShape**
Обратите внимание на синтаксис

Triangle может работать с членами **TwoDShape**, как если бы они были его собственными.

Все члены **Triangle** доступны объектам **Triangle**, даже те, который наследованы от **TwoDShape**

Вывод этой программы выглядит следующим образом:

```
Данные о t1:
Треугольник равнобедренный
Ширина и высота составляют 4 and 4
Площадь равна 8

Данные о t2:
Треугольник прямоугольный
Ширина и высота составляют 8 and 12
Площадь равна 48
```

В этой программе **TwoDShape** определяет атрибуты “обобщенной” двумерной фигуры, например, квадрата, прямоугольника, треугольника и т. д. Класс **Triangle** создает специфический тип класса **TwoDShape**, в данном случае треугольник. Класс **Triangle** включает все члены **TwoDShape** и добавляет поле **style**, функцию **area()** и функцию **showStyle()**. Описание вида треугольника хранится в поле **style**, **area()** вычисляет и возвращает площадь треугольника, а **showStyle()** выводит на экран вид треугольника.

Следующая строка показывает, как **Triangle** наследует от **TwoDShape**:

```
class Triangle : public TwoDShape {
```

Здесь **TwoDShape** — это базовый класс; от него наследует класс **Triangle**, являющийся производным классом. Как показывает приведенный пример, синтаксис для описания наследования весьма прост.

Поскольку **Triangle** наследует все члены своего базового класса, **TwoDShape**, он может внутри функции **area()** обращаться к переменным **width** и **height**. Кроме того, внутри **main()** объекты **t1** и **t2** могут обращаться к **width** и **height** непосредственно, как если бы они были частью **Triangle**. На рис. 10-1 схематически показано, как класс **TwoDShape** входит в **Triangle**.

Еще одно замечание: хотя **TwoDShape** является базовым для **Triangle**, он также является полностью независимым, самостоятельным классом. То, что некоторый класс является базовым для производного класса, никак не значит, что он не может быть использован сам по себе.

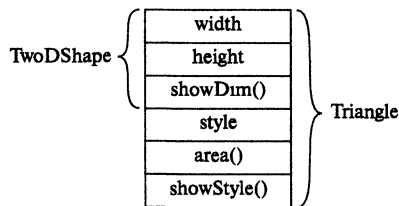


Рис. 10-1. Схематическое изображение класса **Triangle**

Общая форма наследования приведена ниже:

```
class производный-класс : доступ базовый-класс {  
    // тело производного класса  
}
```

Здесь описатель *доступ* не является обязательным. Если он присутствует, он должен быть **public**, **private** или **protected**. Ниже в этом же модуле вы прочитаете больше об этих описателях. Поначалу все наши производные классы будут использовать доступ **public**. Указание этого вида доступа означает, что все открытые члены базового класса будут также открытыми членами производного.

Основное преимущество наследования заключается в возможности, после того, как вы создали базовый класс, определяющий общие характеристики группы объектов, использовать его для создания любого числа более специфических производных классов. Каждый производный класс может детально определить собственные средства. Вот пример другого класса, наследуемого от того же **TwoDShape**, который инкапсулирует прямоугольники:

```
// Производный от TwoDShape класс для прямоугольников.  
class Rectangle : public TwoDShape {  
public:  
    bool isSquare() {  
        if(width == height) return true;  
        return false;  
    }  
  
    double area() {  
        return width * height;  
    }  
};
```

Класс **Rectangle** включает все содержимое **TwoDShape** и добавляет функцию **isSquare()**, которая определяет, не является ли прямоугольник квадратом, а также функцию **area()** для вычисления площади прямоугольника.

Доступ к членам и наследование

Как вы узнали из Модуля 8, члены класса часто объявляются закрытыми, чтобы предотвратить неавторизованный доступ или неразумное использование. Наследование от класса *не* нарушает ограничение доступа к закрытым членам. Таким образом, хотя производный класс включает все члены базового, он не может обратиться к закры-

тым членам базового класса. Например, если переменные **width** и **height** объявить в классе **TwoDShape** закрытыми, как это показано ниже, тогда **Triangle** не будет иметь к ним доступа.

```
/* Производным классам закрыт доступ
   к закрытым членам базового класса. */

class TwoDShape {
    // Эти члены теперь закрыты
    double width; ←———— width и height теперь объявлены закрытыми
    double height;
public:
    void showDim() {
        cout << "Ширина и высота составляют " <<
            width << " и " << height << "\n";
    }
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];
    double area() {
        return width * height / 2; // Ошибка! Доступ закрыт.
    }

    void showStyle() {
        cout << "Треугольник " << style << "\n";
    }
};
```

К закрытым членам базового
класса обращаться нельзя

Класс **Triangle** не будет компилироваться, потому что обращение к **width** и **height** внутри функции **area()** приведет к нарушению доступа. Поскольку **width** и **height** теперь закрыты, к ним можно обратиться только из членов их собственного класса. Производные классы доступа к ним не имеют.

Поначалу вам может показаться, что запрещение доступа к закрытым членам базового класса из производных классов является серьезным ограничением, потому что это во многих случаях лишит нас возможности использовать закрытые члены. К счастью, это не так, поскольку C++ предоставляет ряд средств для решения этой проблемы. Одним из них является атрибут **protected** (защищенный), который будет описан в следующем подразделе. Второе средство заключается в использовании открытых функций для обеспечения доступа к закрытым данным. Как вы уже знаете из материала предыдущих модулей, C++-программисты обычно пре-

доставляют доступ к закрытым членам класса посредством функций. Функции, предоставляющие доступ к закрытым членам, носят название *функций доступа*. Ниже приведен новый вариант класса **TwoDShape**, в который добавлены функции доступа для переменных **width** и **height**:

```
// Обращение к закрытым данным посредством функций доступа.
#include <iostream>
#include <cstring>
using namespace std;

// Класс для двумерных объектов.
class TwoDShape {
// эти члены закрыты
    double width;
    double height;
public:

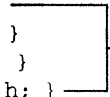
    void showDim() {
        cout << "Ширина and высота составляют " <<
            width << " и " << height << "\n";
    }
    // функции доступа
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
public:
    char style[20];

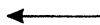
    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Треугольник " << style << "\n";
    }
};

int main() {
    Triangle t1;
    Triangle t2;
```



Функция доступа для width и height



Использование функций доступа для получения ширины и высоты.


```
t1.setWidth(4.0);
t1.setHeight(4.0);
strcpy(t1.style, "равнобедренный");

t2.setWidth(8.0);
t2.setHeight(12.0);
strcpy(t2.style, "прямоугольный");

cout << "Данные о t1:\n";
t1.showStyle();
t1.showDim();
cout << "Площадь равна " << t1.area() << "\n";

cout << "\n";
cout << "Данные о t2:\n";
t2.showStyle();
t2.showDim();
cout << "Площадь равна " << t2.area() << "\n";

return 0;
}
```

Спросим у эксперта

Вопрос: Я услышал в дискуссии о программировании на языке Java термины *суперкласс* и *подкласс*. Имеют ли эти термины смысл в C++?

Ответ: То, что в Java называется суперклассом, в C++ именуется базовым классом. Подкласс в Java — это производный класс в C++. Вы можете услышать обе пары терминов применительно к обоим языкам, однако в этой книге мы будем продолжать использовать стандартную терминологию C++. Кстати, в языке C# также приняты термины базовый класс и производный класс.

Минутная тренировка

1. Каким образом указывается, что данный класс является производным от базового?
 2. Включает ли в себя производный класс члены базового класса?
 3. Имеет ли производный класс доступ к закрытым членам своего базового класса?
-
- 1 Базовый класс указывается через двоеточие после имени производного класса
 - 2 Да, производный класс включает ли в себя все члены своего базового класса
 - 3 Нет, производный класс не имеет доступа к закрытым членам своего базового класса

Цель

10.2. Управление доступом к базовому классу

Как уже было сказано, если один класс наследует от другого, все члены базового класса становятся членами производного. Однако доступность членов базового класса внутри производного определяется описателем доступа, используемым при наследовании членов базового класса. Описатель доступа в базовом классе может принимать формы **public** (открытый), **private** (закрытый) или **protected** (защищенный). Если описатель доступа не используется, то по умолчанию для структуры **class** принимается описатель **private**, а для структуры **struct** — описатель **public**. Рассмотрим следствия использования описателей доступа **public** и **private** (описатель **protected** будет описан в следующем подразделе).

Если базовый класс наследуется как **public**, все открытые члены базового класса становятся открытыми членами производного класса. Во всех случаях закрытые элементы базового класса остаются закрытыми в этом классе и недоступны для членов производного класса. Например, в приведенной ниже программе открытые члены **B** становятся открытыми членами **D**. Таким образом, все они доступны из других частей программы:

// Демонстрация наследования **public**.

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

```
class B {
    int i, j;
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};
```

class D : public B { ←————— Класс **B** наследуется как **public**.

```
    int k;
public:
    D(int x) { k = x; }
    void showk() { cout << k << "\n"; }
    // i = 10; // Ошибка! i объявлена private в B и недоступна в
D.
};
```

↑
Доступ к члену **i** запрещен, так как **i** закрыта в классе **B**

```
int main()
```

```

{
    D ob(3);
    ob.set(1, 2);    // функция доступа базового класса
    ob.show();       // функция доступа базового класса

    ob.showk();      // использует член производного класса

    return 0;
}

```

Поскольку функции `set()` и `show()` объявлены как **public** в классе **B**, их можно вызвать для объекта типа **D** внутри `main()`. Поскольку `i` и `j` объявлены как **private**, они остаются закрытыми в классе **B**. Поэтому строка

```
// i = 10; // Ошибка! i объявлена private в B и недоступна в D.
```

закомментирована. **D** не имеет доступа к закрытым членам **B**.

Противоположностью открытому является закрытое наследование. Если базовый класс наследуется как **private**, все открытые члены базового класса становятся закрытыми членами в производном классе. Например, приведенная ниже программа не будет компилироваться, так как функции `set()` и `show()` теперь являются закрытыми членами **D**, и, следовательно, не могут быть вызваны из `main()`:

```
/* Демонстрация наследования private.
```

```
Эта программа компилироваться не будет. */
```

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

```
class B {
    int i, j;
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};
```

```
// Открытые элементы в B становятся закрытыми в D.
```

```
class D : private B {
```

```
    int k;
```

```
public:
```

```
    D(int x) { k = x; }
```

```
    void showk() { cout << k << "\n"; }
```

```
};
```

← Теперь **B** наследуется как **private**.

```
int main()
{
    D ob(3);

    ob.set(1, 2); // Ошибка, нет доступа к set()
    ob.show();    // Ошибка, нет доступа к show()

    return 0;
}
```

Теперь **set()** and **show()** недоступны через объекты **D**.

Подытожим: если базовый класс наследуется как **private**, открытые члены базового класса становятся закрытыми членами производного класса. Это значит, что они все еще доступны из членов производного класса, но из других частей вашей программы к ним обратиться нельзя.

Цель

10.3. Использование защищенных членов

Как вы знаете, закрытый член базового класса недоступен из производного класса. Отсюда можно сделать заключение, что если вы хотите, чтобы производный класс имел доступ к какому-то члену базового, его следует объявить открытым. Однако объявление члена с дескриптором **public** сделает его открытым и для всего остального кода, что часто нежелательно. К счастью, наше заключение неправильно, так как C++ позволяет создавать *защищенные члены*. Защищенный член является открытым внутри иерархии классов, но закрытым вне этой иерархии.

Защищенный член создается указанием дескриптора доступа **protected**. Если член класса объявлен как **protected**, этот член, за одним важным исключением, является закрытым. Исключение проявляется, когда этот защищенный член наследуется. Такой защищенный член базового класса оказывается доступным из производного класса. Таким образом, используя дескриптор **protected**, вы можете создавать члены класса, которые закрыты в своем классе, но тем не менее наследуются и доступны из производного класса. Дескриптор **protected** может также использоваться со структурами.

Рассмотрим такую программу:

```
// Демонстрация защищенных членов.

#include <iostream>
```

```

using namespace std;
class B {
protected:
    int i, j; // закрыты в B, но доступны из D
public:
    void set(int a, int b) { i = a; j = b; }
    void show() { cout << i << " " << j << "\n"; }
};

class D : public B {
    int k;
public:
    // D может обращаться к i и j, принадлежащим B.
    void setk() { k = i*j; }

    void showk() { cout << k << "\n"; }
};

int main()
{
    D ob;

    ob.set(2, 3); // OK, set() открыта в B
    ob.show();   // OK, show() открыта в B

    ob.setk();
    ob.showk();

    return 0;
}

```

← Переменные *i* и *j* объявлены **protected**

D имеет доступ к *i* и *j*, так как они защищенные, а не закрытые

В этой программе, поскольку **B** наследуется классом **D** как открытый класс, а *i* и *j* объявлены защищенными, функция **setk()** из класса **D** может к ним обращаться. Если бы *i* и *j* были объявлены в **B** закрытыми, тогда класс **D** не имел бы к ним доступа, и программа не стала бы компилироваться.

Когда базовый класс наследуется с атрибутом **public**, защищенные члены базового класса становятся защищенными членами производного класса. Когда базовый класс наследуется с атрибутом **private**, защищенные члены базового класса становятся закрытыми членами производного класса.

Описатель доступа **protected** может быть указан в любом месте в объявлении класса, хотя обычно его используют после того, как объявлены закрытые (по умолчанию) члены класса, но до объявления открытых членов. Поэтому наиболее распространенная полная форма объявления класса выглядит таким образом:

```
class имя-класса {  
    // закрытые члены класса  
protected:  
    // защищенные члены  
public:  
    // открытые члены  
};
```

Разумеется, защищенная секция не является обязательной.

В дополнение к определению атрибута защищенности для членов класса, ключевое слово **protected** может также служить описателем доступа при наследовании базового класса. Если базовый класс наследуется как защищенный, все открытые и защищенные члены базового класса становятся защищенными членами производного класса. Например, в последнем примере, если D наследует от B таким образом:

```
class D : protected B {
```

тогда все члены B станут защищенными членами D.

Минутная тренировка

1. Если базовый класс наследуется как закрытый, открытые члены базового класса становятся закрытыми членами производного класса. Правильно ли это?
 2. Можно ли сделать закрытый член базового класса открытым посредством наследования?
 3. Чтобы сделать член доступным внутри иерархии, но закрытым в любом другом месте, какой описатель доступа следует использовать?
1. Правильно, если базовый класс наследуется как закрытый, открытые члены базового класса становятся закрытыми членами производного класса
 2. Нет, закрытый член всегда остается закрытым в своем классе
 3. Для того, чтобы член был доступен внутри иерархии, но закрыт в остальных местах программы, следует использовать описатель доступа **protected**.
-

Спросим у эксперта

Вопрос: Не могли бы вы дать краткий обзор описателям **public**, **protected** и **private**?

Ответ: Если член класса объявлен как **public**, к нему можно обратиться из любого места в программе. Если член объявлен как **private**, он доступен только членам своего класса. При этом производные классы не имеют доступа к членам **private** базового класса. Если член объявлен как

protected, он доступен только членам своего класса и производных классов. Таким образом, атрибут **protected** позволяет члену наследоваться, но оставаться закрытым внутри иерархии классов

Если базовый класс наследуется с атрибутом **public**, его открытые члены становятся открытыми членами производного класса, а его защищенные члены становятся защищенными членами производного класса. Если базовый класс наследуется с атрибутом **protected**, его открытые и защищенные члены становятся защищенными членами производного класса. Если базовый класс наследуется с атрибутом **private**, его открытые и защищенные члены становятся закрытыми членами производного класса. Во всех случаях закрытые члены базового класса остаются закрытыми в этом классе.

Конструкторы и наследование

В иерархии классов вполне возможно, чтобы и базовые, и наследуемые классы имели собственные конструкторы. В этом случае возникает важный вопрос: какой конструктор отвечает за построение объекта производного класса, тот, который находится в базовом классе, или тот, который находится в производном, или оба вместе? Ответ заключается в следующем: конструктор базового класса конструирует часть объекта, относящуюся к базовому классу, а конструктор производного класса конструирует часть, относящуюся к производному классу. Это представляется разумным, поскольку базовый класс ничего не знает про производный класс и не имеет доступа к его элементам. Поэтому конструирование двух частей объекта производного класса должно осуществляться отдельно. Предыдущие примеры опирались на конструкторы по умолчанию, создаваемые автоматически, поэтому с обсуждаемым здесь вопросом мы не сталкивались. Однако на практике большинство классов имеют конструкторы. Здесь мы рассмотрим вопросы их взаимодействия.

Если конструктор определен только в производном классе, процесс построения объекта очевиден: надо просто сконструировать объект производного класса. Часть объекта, относящаяся к базовому классу, конструируется автоматически конструктором базового класса по умолчанию.

Ниже приведен переработанный вариант класса **Triangle**, в котором определен конструктор. Кроме того, член **style** объявлен закрытым, поскольку теперь его будет устанавливать конструктор:

```
// Добавление в класс Triangle конструктора.
```

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

```

using namespace std;

// Класс для двумерных объектов.
class TwoDShape {
    // эти члены закрыты
    double width;
    double height;
public:
    void showDim() {
        cout << "Ширина and высота составляют " <<
            width << " и " << height << "\n";
    }

    // функции доступа
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:

    // Конструктор для класса Triangle.
    Triangle(char *str, double w, double h) {
        // Инициализация части, относящейся к базовому классу.
        setWidth(w);
        setHeight(h);
        // Инициализация части, относящейся к производному классу.
        strcpy(style, str);
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        cout << "Треугольник " << style << "\n";
    }
};

int main() {
    Triangle t1("равнобедренный", 4.0, 4.0);

```

Инициализация части **Triangle**, относящейся к **TwoDShape**

Инициализация **style**, переменной, специфической для **Triangle**


```

Triangle t2("прямоугольный", 8.0, 12.0);

cout << "Данные о t1:\n";
t1.showStyle();
t1.showDim();
cout << "Площадь равна " << t1.area() << "\n";

cout << "\n";
cout << "Данные о t2:\n";
t2.showStyle();
t2.showDim();
cout << "Площадь равна " << t2.area() << "\n";

return 0;
}

```

В этой программе конструктор класса **Triangle** инициализирует члены **TwoDShape**, которые **Triangle** наследует от **TwoDShape**, а также собственное поле **style**.

Когда оба класса, и базовый, и производный, определяют свои конструкторы, процесс становится несколько более сложным, потому что при создании объекта должны выполняться оба конструктора.

Цель

10.4. Вызов конструктора базового класса

Если в базовом классе есть свой конструктор, производный класс должен вызывать его явным образом, чтобы инициализировать часть объекта, относящуюся к базовому классу. Производный класс может вызывать конструктор, определенный в базовом классе, посредством расширенной формы объявления конструктора производного класса. Общая форма этого расширенного объявления выглядит так:

```

производный-конструктор (список-аргументов) : базовый-
конструктор (список-аргументов)
{
    тело конструктора производного класса
}

```

Здесь *базовый конструктор* представляет собой имя базового класса, от которого наследует производный класс. Обратите внимание на двоеточие, отделяющее объявление конструктора производного класса

от конструктора базового класса. (Если класс наследует от более чем одного базового класса, тогда конструкторы базовых классов отделяются друг от друга запятыми.)

Приведенная ниже программа показывает способ передачи аргументов конструктору базового класса. В ней определен конструктор класса **TwoDShape**, который инициализирует переменные **width** и **height**:

```
// Добавление конструктора в класс TwoDShape.

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

// Класс для двумерных объектов.
class TwoDShape {
    // эти члены закрыты
    double width;
    double height;
public:

    // Конструктор для TwoDShape.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }

    void showDim() {
        cout << "Ширина и высота составляют " <<
            width << " и " << height << "\n";
    }

    // функции доступа
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:

    // Конструктор для класса Triangle.
    Triangle(char *str, double w,
```

```

double h) : TwoDShape(w, h) { ←—————Вызов конструктора TwoDShape
strcpy(style, str);
}

double area() {
return getWidth() * getHeight() / 2;
}

void showStyle() {
cout << "Треугольник " << style << "\n";
}
};

int main() {
Triangle t1("равнобедренный", 4.0, 4.0);
Triangle t2("прямоугольный", 8.0, 12.0);

cout << "Данные о t1:\n";
t1.showStyle();
t1.showDim();
cout << "Площадь равна " << t1.area() << "\n";

cout << "\n";
cout << "Данные о t2:\n";
t2.showStyle();
t2.showDim();
cout << "Площадь равна " << t2.area() << "\n";

return 0;
}

```

В этой программе **Triangle()** вызывает **TwoDShape** с параметрами **w** и **h**, которые инициализируют переменные **width** и **height** этими значениями. **Triangle** уже не инициализирует их сам. Ему требуется инициализировать только переменную, специфическую для этого класса, именно, **style**. Такая процедура оставляет за **TwoDShape** возможность конструировать свои подобъекты любым способом, какой ему будет удобен. Более того, **TwoDShape** может увеличить свои функциональные возможности, о чем существующие производные классы нечего не будут знать, и это не приведет к разрушению существующих программ.

Конструктором производного класса может быть вызвана любая форма конструктора из определенных в базовом классе. Выполняться будет конструктор с соответствующим списком аргументов. Ниже приведены расширенные варианты обоих классов, **TwoDShape** и **Triangle**, которые включают дополнительные конструкторы:

```
// Добавление конструктора в TwoDShape.
```

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

```
// Класс для двумерных объектов.
class TwoDShape {
```

```
// Эти члены закрыты
    double width;
    double height;
public:
```

```
    // Конструктор по умолчанию.
    TwoDShape() {
        width = height = 0.0;
    }
```

```
    // Конструктор для TwoDShape.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
```

```
    // Конструируем объект с равными шириной и высотой.
    TwoDShape(double x) {
        width = height = x;
    }
```

```
    void showDim() {
        cout << "Ширина and высота составляют " <<
            width << " и " << height << "\n";
    }
```

```
    // функции доступа
    double getWidth() { return width; }
    double getHeight() { return height; }
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};
```

```
// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:
```

Различные конструкторы **TwoDShape**



```

/* Конструктор по умолчанию. Он автоматически вызывает
   конструктор по умолчанию TwoDShape. */
Triangle() {
    strcpy(style, "неизвестен");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
          double h) : TwoDShape(w, h) {
    strcpy(style, str);
}

// Сконструируем равнобедренный треугольник.
Triangle(double x) : TwoDShape(x) {
    strcpy(style, "равнобедренный");
}

double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Треугольник " << style << "\n";
}
};

int main() {
    Triangle t1;
    Triangle t2("прямоугольный", 8.0, 12.0);
    Triangle t3(4.0);

    t1 = t2;

    cout << "Данные о t1: \n";
    t1.showStyle();
    t1.showDim();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";

    cout << "Данные о t2: \n";
    t2.showStyle();
    t2.showDim();
    cout << "Площадь равна " << t2.area() << "\n";

    cout << "\n";

```

Различные кон-
структоры **Triangle**

```
cout << "Данные о t3: \n";
t3.showStyle();
t3.showDim();
cout << "Площадь равна " << t3.area() << "\n";

cout << "\n";

return 0;
}
```

Вот вывод этой программы:

Данные о t1:
Треугольник прямоугольный
Ширина и высота составляют 8 and 12
Площадь равна 48

Данные о t2:
Треугольник прямоугольный
Ширина и высота составляют 8 and 12
Площадь равна 48

Данные о t3:
Треугольник равнобедренный
Ширина и высота составляют 4 and 4
Площадь равна 8

Минутная тренировка

1. Каким образом производный класс активизирует выполнение конструктора своего базового класса?
 2. Можно ли передать параметры конструктору базового класса?
 3. Какой конструктор отвечает за инициализацию той части объекта производного класса, которая относится к базовому классу? Конструктор базового класса или производного класса?
- 1 Производный класс указывает конструктор базового класса в своем конструкторе
 - 2 Да, конструктору базового класса можно передать параметры
 - 3 За инициализацию той части объекта производного класса, которая относится к базовому классу, отвечает конструктор, определенный в базовом классе
-

Проект 10-1 **Расширение класса Vehicle**

В этом проекте создается подкласс класса **Vehicle**, разработанного в Модуле 8. Как вы помните, класс **Vehicle** инкапсулировал ин-

формацию о транспортных средствах, включая число перевозимых пассажиров, емкость бензобака и скорость потребления горючего. Мы можем использовать класс **Vehicle** в качестве отправной точки для разработки более специализированных классов. К примеру, одним из транспортных средств является грузовик (truck). К важным характеристикам грузовика относится его грузоподъемность. Таким образом, создавая класс **Truck**, вы можете сделать его производным от **Vehicle**, добавив переменную экземпляра, содержащую величину грузоподъемности. В этом проекте мы создадим класс **Truck**. Переменные экземпляра **Vehicle** мы сделаем закрытыми, и включим в класс функции доступа для получения значений этих переменных.

Шаг за шагом

1. Создайте файл с именем **TruckDemo.cpp** и скопируйте в него последнюю реализацию **Vehicle** из Модуля 8.
2. Создайте класс **Truck**, как это показано ниже:

```
// Используем Vehicle для создания специализированного
// класса Truck.
class Truck : public Vehicle {
    int cargocap; // грузоподъемность в фунтах
public:

    // Это конструктор для Truck.
    Truck(int p, int f,
          int m, int c) : Vehicle(p, f, m)
    {
        cargocap = c;
    }

    // Функция доступа для cargocap.
    int get_cargocap() { return cargocap; }
};
```

Мы видим, что **Truck** наследует от **Vehicle**; кроме того, в него добавляется член **cargocap**. Таким образом, **Truck** включает все общие атрибуты транспортных средств, определенные в **Vehicle**. Добавить в него надо только элементы, специфические именно для этого класса.

3. Ниже приведен полный текст программы, демонстрирующей создание и использование класса **Truck**:

```
// Создание подкласса Truck, производного от Vehicle.
#include <iostream>
```

```
using namespace std;

// Объявим класс Vehicle.
class Vehicle {
    // Эти члены закрыты.
    int passengers; // число пассажиров
    int fuelcap;     // емкость бензобака в галлонах
    int mpg;         // потребление топлива в милях на галлон
public:
    // Это конструктор для Vehicle.
    Vehicle(int p, int f, int m) {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }

    // Вычислить и вернуть дальность пробега.
    int range() { return mpg * fuelcap; }

    // Функции доступа.
    int get_passengers() { return passengers; }
    int get_fuelcap() { return fuelcap; }
    int get_mpg() { return mpg; }
};

// Используем Vehicle для создания специализированного
// класса Truck.
class Truck : public Vehicle {
    int cargocap; // грузоподъемность в фунтах
public:

    // Это конструктор для Truck.
    Truck(int p, int f,
          int m, int c) : Vehicle(p, f, m)
    {
        cargocap = c;
    }

    // Функция доступа для cargocap.
    int get_cargocap() { return cargocap; }
};

int main() {

    // Сконструируем несколько грузовиков
    Truck semi(2, 200, 7, 44000); // Фургон
```



```

Truck pickup(3, 28, 15, 2000); // Пикап
int dist = 252; //Расстояние

cout << "Фургон может возить " << semi.get_cargocap() <<
      " фунтов груза.\n";
cout << "Он имеет дальность пробега " <<
      semi.range() << " миль.\n";
cout << "Пробег в " << dist << " миль требует " <<
      dist / semi.get_mpg() <<
      " галлонов топлива.\n\n";

cout << "Пикап может возить " << pickup.get_cargocap() <<
      " фунтов груза.\n";
cout << "Он имеет дальность пробега " <<
      pickup.range() << " миль.\n";
cout << "Пробег в " << dist << " миль требует " <<
      dist / pickup.get_mpg() <<
      " галлонов топлива.\n";

return 0;
}

```

4. Вывод программы будет таким:

Фургон может возить 44000 фунтов груза.
 Он имеет дальность пробега 1400 миль.
 Пробег в 252 миль требует 36 галлонов топлива.

Пикап может возить 2000 фунтов груза.
 Он имеет дальность пробега 420 миль.
 Пробег в 252 миль требует 16 галлонов топлива.

5. От класса **Vehicle** можно образовать много других типов классов. Например, следующая заготовка создает класс внедорожника, который содержит величину дорожного просвета для машины:

```

// Создадим класс внедорожника
class OffRoad : public Vehicle {
    int groundClearance; // дорожный просвет в дюймах
public:
    // ...
};

```

Ключевой момент здесь заключается в том, что как только вы создали базовый класс, определяющий общие характеристики объекта,

этот базовый класс может наследоваться для образования специализированных классов. Каждый производный класс просто добавляет свои специфические характеристики. В этом и заключается суть наследования.

Цель

10.5. Создание многоуровневой иерархии классов

До сих пор мы имели дело с простыми иерархическими структурами, состоящими только из двух классов: базового и производного. В действительности можно создавать иерархические структуры, содержащие любое число уровней наследования. Как уже отмечалось, производный класс вполне допустимо использовать в качестве базового для следующего производного класса. Например, если у нас имеются три класса **A**, **B** и **C**, то **C** может быть производным от **B**, а **B**, в свою очередь, производным от **A**. В случае такой ситуации каждый производный класс наследует все содержимое всех своих базовых классов. В нашем примере **C** наследует все содержимое **B** и **A**.

Для демонстрации смысла многоуровневой иерархии рассмотрим следующую программу. В ней производный класс **Triangle** (треугольник) используется в качестве базового для создания производного класса **ColorTriangle** (цветной треугольник). **ColorTriangle** наследует все содержимое классов **Triangle** и **TwoDShape**, и добавляет поле **color**, в котором хранится цвет треугольника.

```
// Многоуровневая иерархия.
#include <iostream>
#include <cstring>
using namespace std;

// Класс для двумерных объектов.
class TwoDShape {
    // эти члены закрыты
    double width;
    double height;
public:

    // Конструктор по умолчанию.
    TwoDShape() {
        width = height = 0.0;
    }
}
```

```
// Конструктор для TwoDShape.
TwoDShape(double w, double h) {
    width = w;
    height = h;
}

// Конструируем объект с равными шириной и высотой.
TwoDShape(double x) {
    width = height = x;
}

void showDim() {
    cout << "Ширина и высота составляют " <<
        width << " и " << height << "\n";
}

// функции доступа
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:
    /* Конструктор по умолчанию. Он автоматически вызывает
       конструктор по умолчанию TwoDShape. */
    Triangle() {
        strcpy(style, "неизвестно");
    }

    // Конструктор с тремя параметрами.
    Triangle(char *str, double w,
        double h) : TwoDShape(w, h) {
        strcpy(style, str);
    }

    // Конструируем равнобедренный треугольник.
    Triangle(double x) : TwoDShape(x) {
        strcpy(style, "равнобедренный");
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }
}
```

```

void showStyle() {
    cout << "Треугольник " << style << "\n";
}
};

// Расширим Triangle.
class ColorTriangle : public Triangle {
    char color[20];
public:
    ColorTriangle(char *clr, char *style, double w,
        double h) : Triangle(style, w, h) {
        strcpy(color, clr);
    }

    // Выведем цвет.
    void showColor() {
        cout << "Цвет " << color << "\n";
    }
};

int main() {
    ColorTriangle t1("Синий", "прямоугольный", 8.0, 12.0);
    ColorTriangle t2("Красный", "равнобедренный", 2.0, 2.0);

    cout << "Данные о t1:\n";
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    cout << "Площадь равна " << t1.area() << "\n";

    cout << "\n";

    cout << "Данные о t2:\n";
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    cout << "Площадь равна " << t2.area() << "\n";

    return 0;
}

```

ColorTriangle наследует от Triangle, который наследует от TwoDShape.

Объект ColorTriangle может вызывать функции, определенные им самим и его базовыми классами.

Вывод программы выглядит следующим образом:

```

Данные о t1:
Треугольник прямоугольный
Высота и ширина составляют 8 и 12

```

Цвет Синий
Площадь равна 48

Данные о t2:
Треугольник равнобедренный
Ширина и высота составляют 2 и 2
Цвет Красный
Площадь равна 2

Благодаря наследованию класс **ColorTriangle** может использовать предварительно определенные классы **Triangle** и **TwoDShape**, добавив только специфическую информацию, которая нужна ему для его собственного, специфического использования. В этом заключается часть ценности наследования; наследование является базой для повторного использования кодов.

Рассмотренный пример иллюстрирует еще один важный принцип. Если конструктор базового класса в иерархии классов требует параметры, тогда все производные классы должны передавать эти параметры вверх по цепочке классов. Это правило остается справедливым независимо от того, требует ли сам производный класс параметры или нет.

Цель

10.6. Наследование от нескольких базовых классов

В C++ имеется возможность создать производный класс, который будет наследовать одновременно от двух или нескольких базовых классов (так называемое множественное наследование). Например, в этой короткой программе класс **D** наследует и от **B1**, и от **B2**:

```
// Пример нескольких базовых классов.
#include <iostream>
using namespace std;

class B1 {
protected:
    int x;
public:
    void showx() { cout << x << "\n"; }
};

class B2 {
protected:
```

```
int y;
public:
void showy() { cout << y << "\n"; }
};

// Множественное наследование.
class D: public B1, public B2 { ← Здесь D одновременно на-
public:                               следует и от B1, и от B2
    /* x и y доступны, потому что они объявлены
       в B1 и B2 защищенными, а не закрытыми. */
    void set(int i, int j) { x = i; y = j; }
};

int main()
{
    D ob;

    ob.set(10, 20); // берется из D
    ob.showx();     // из B1
    ob.showy();     // из B2

    return 0;
}
```

Как показывает этот пример, для того, чтобы наследовать от более чем одного базового класса, вы должны использовать список базовых классов, разделяемых запятыми. Не забудьте также для каждого базового наследуемого класса указать описатель доступа.

Цель

10.7. Когда выполняются функции конструктора и деструктора

Поскольку и базовый класс, и производный класс могут содержать конструкторы и деструкторы, важно понимать, в каком порядке они выполняются. Конкретно, в каком порядке вызываются конструкторы, когда начинает существовать объект производного класса? А когда объект перестает существовать, в каком порядке вызываются деструкторы? Для того, чтобы ответить на эти вопросы, начнем с такой простой программы:

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

```
class B {
public:
    B() { cout << "Конструируется базовая часть\n"; }
    ~B() { cout << "Уничтожается базовая часть\n"; }
};

class D: public B {
public:
    D() { cout << "Конструируется производная часть\n"; }
    ~D() { cout << "Уничтожается производная часть\n"; }
};

int main()
{
    D ob;

    // не делаем ничего, только конструируем и уничтожаем ob

    return 0;
}
```

Как указано в комментарии к функции `main()`, эта программа просто конструирует и уничтожает объект с именем `ob`, принадлежащий классу `D`. Если программу запустить, она выведет на экран следующее:

```
Конструируется базовая часть
Конструируется производная часть
Уничтожается производная часть
Уничтожается базовая часть
```

Как показывает этот вывод, сначала выполняется конструктор `B`, затем конструктор `D`. Далее (поскольку `ob` в этой программе немедленно уничтожается) вызывается деструктор `D`, а вслед за ним — деструктор `B`.

Результаты проведенного эксперимента можно обобщить следующим образом. Когда создается объект производного класса, сначала вызывается конструктор базового класса, затем конструктор производного класса. Когда объект производного класса уничтожается, сначала вызывается его деструктор, затем деструктор базового класса. Другими словами, конструкторы выполняются в порядке наследования их классов. Деструкторы выполняются в порядке, обратном порядку наследования их классов.

В случае многоуровневой иерархии классов (т. е. когда производный класс выступает в качестве базового класса для другого производного класса), используется то же общее правило: конструкторы

вызываются в порядке наследования их классов; деструкторы вызываются в обратном порядке. Если конструктор наследует более чем от одного класса одновременно, конструкторы вызываются в порядке их перечисления (слева направо) в списке базовых конструкторов производного класса. Деструкторы вызываются в обратном порядке, справа налево.

Спросим у эксперта

Вопрос: Почему конструкторы вызываются в порядке наследования их классов, а деструкторы в обратном порядке?

Ответ: Если вдуматься, то установленный порядок вызова конструкторов разумен. Поскольку базовый класс ничего не знает о производном классе, то любая требуемая ему инициализация должна выполняться независимо и, возможно, как необходимое условие инициализации, выполняемой производным классом. Поэтому конструктор базового класса должен выполняться первым.

Аналогично, вполне разумно, что деструкторы выполняются в порядке, обратном порядку образования их классов. Поскольку базовый класс служит основой для производного, уничтожение базового класса предполагает уничтожение и производного класса. Поэтому деструктор производного класса должен быть вызван перед тем, как объект будет полностью уничтожен.

Минутная тренировка

1. Можно ли использовать производный класс в качестве базового для другого производного класса?
 2. В каком порядке вызываются конструкторы в иерархии классов?
 3. В каком порядке вызываются деструкторы в иерархии классов?
- 1 Да, производный класс может быть использован в качестве базового для другого производного класса
 - 2 Конструкторы вызываются в порядке наследования их классов
 - 3 Деструкторы вызываются в порядке, обратном порядку наследования их классов

Цель

10.8. Указатели на производные классы

Перед тем, как приступить к изучению виртуальных функций и полиморфизма, нам необходимо обсудить некоторые важные аспек-

ты использования указателей. Указатели на базовый и производный классы связаны друг с другом, чего мы не наблюдаем для указателей других типов. Как правило, указатель одного типа не может указывать на объект другого типа. Однако указатели базовых и производных классов выпадают из этого правила. В C++ указатель базового класса может быть использован в качестве указателя на объект любого другого класса, производного от этого базового. Предположим, например, что у нас есть базовый класс **B** и производный от него класс **D**. Любой указатель, объявленный как указатель на **B**, можно использовать для указания на объект типа **D**. Таким образом, если имеются объявления:

```
B *p;      // указатель на объект типа B
B B_ob;    // объект типа B
D D_ob;    // объект типа D
```

то следующие предложения вполне правильны:

```
p = &B_ob; // p указывает на объект типа B
p = &D_ob; /* p указывает на объект типа D,
            который является производным от B. */
```

Базовый указатель можно использовать только для обращения к тем частям производного объекта, которые наследованы от базового класса. Так, в этом примере **p** можно использовать для обращения ко всем элементам **D_ob**, унаследованным от **B_ob**. Однако к элементам, специфическим для **D_ob**, посредством **p** обратиться нельзя (если только не применено приведение типа).

Еще необходимо усвоить, что хотя посредством базового указателя можно обратиться к производному объекту, обратное несправедливо. Вы не можете обратиться к объекту базового типа, используя указатель на производный класс.

Как вы уже знаете, инкремент и декремент указателя осуществляется относительно его базового типа. Поэтому в случае, когда указатель базового класса указывает на производный объект, операции инкремента или декремента *не приведут* к смещению указателя на следующий объект производного класса. Вместо этого указатель будет указывать на то, что он считает следующим объектом базового класса. Таким образом, если указатель базового класса указывает на производный объект, вы не должны выполнять над ним операции инкремента или декремента.

Тот факт, что указатель на базовый тип может быть использован для указания на любой объект, производный от этого базового типа, является исключительно важным и фундаментальным для C++. Как вы вскоре узнаете, эта возможность позволяет C++ реализовать полиморфизм времени выполнения.

Ссылки на производные типы

Аналогично описанному только что использованию указателей, ссылка на базовый класс может быть использована для обращения к объекту производного класса. Чаще всего эта возможность используется в параметрах функций. Параметр-ссылка базового класса может принимать объекты как базового класса, так и любых классов, производных от этого базового.

Цель

10.9

Виртуальные функции и полиморфизм

Фундамент, на котором C++ строит свою поддержку полиморфизма, состоит из наследования и указателей на базовые классы. Конкретным средством, которое фактически реализует полиморфизм, является виртуальная функция. Оставшаяся часть этого модуля будет посвящена этому важному средству.

Основы виртуальных функций

Виртуальной называется функция, объявленная в базовом классе с описателем **virtual** и переопределенная в одном или нескольких производных классах. Таким образом, каждый производный класс может иметь собственный вариант виртуальной функции. Свойства виртуальности проявляются у виртуальных функций, когда для их вызова используется указатель базового класса. Если виртуальная функция вызывается посредством указателя базового класса, C++ определяет, какой из вариантов этой виртуальной функции вызвать, основываясь на *типе* объекта, на который указывает указатель. Следовательно, если базовый класс содержит виртуальную функцию, и два или больше различных классов наследуют от этого базового класса, то в зависимости от того, на объект какого производного класса указывает указатель базового класса, будут вызываться различные варианты виртуальной функции.

Вы объявляете функцию виртуальной внутри базового класса, предваряя ее объявление ключевым словом **virtual**. Когда виртуальная функция переопределяется в производном классе, нет необходимости повторять ключевое слово **virtual** (хотя не будет ошибкой и его повторение).

Класс, который содержит виртуальные функции, называется *полиморфным* классом. Этот термин также относится и к классу, который наследует от базового класса, содержащего виртуальную функцию.

В следующей программе продемонстрирована работа с виртуальной функцией:

// Короткий пример использования виртуальной функции.

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

```
class B {
public:
```

```
    virtual void who() { // Определим виртуальную функцию
        cout << "Базовый класс\n";
    }
```

```
};
```

```
class D1 : public B {
public:
```

```
    void who() { // переопределим who() для D1
        cout << "Первый производный класс\n";
    }
```

```
};
```

```
class D2 : public B {
public:
```

```
    void who() { // переопределим who() для D2
        cout << "Второй производный класс\n";
    }
```

```
};
```

```
int main()
{
```

```
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;
```

```
    p = &base_obj;
    p->who(); // обращение к who из B
```

```
    p = &D1_obj;
    p->who(); // обращение к who из D1
```

```
    p = &D2_obj;
    p->who(); // обращение к who из D2
```

```
    return 0;
```

```
}
```

Объявим виртуальную функцию

Переопределим виртуальную функцию для D1

Переопределим виртуальную функцию второй раз для D2.

Вызов виртуальной функции посредством указателя базового класса.

Эта программа выводит на экран следующее:

Базовый класс
Первый производный класс
Второй производный класс

Рассмотрим эту программу детально, чтобы понять, как она работает.

Как вы видите, функция `who()` объявлена в классе **B** как виртуальная. Это означает, что функцию можно переопределить в производном классе. Внутри каждого из производных классов **D1** и **D2** функция переопределяется относительно своего класса. Внутри `main()` объявлены четыре переменные: `base_obj`, представляющая собой объект типа **B**; `p`, являющаяся указателем на объекты класса **B**; наконец, `D1_obj` и `D2_obj` – объекты двух производных классов. Далее переменной `p` присваивается адрес объекта `base_obj`, после чего вызывается функция `who()`. Поскольку `who()` объявлена как виртуальная, C++ определяет во время выполнения программы, какой вариант `who()` вызывать, основываясь на типе объекта, на который указывает `p`. В данном случае `p` указывает на объект типа **B**, поэтому вызывается вариант `who()` из класса **B**. Далее по ходу программы `p` присваивается адрес объекта `D1_obj`. Вспомним, что указатель базового класса может ссылаться на объект любого производного класса. Теперь при вызове `who()` C++ снова анализирует, на объект какого типа указывает `p` и, в зависимости от этого типа, определяет, какой вариант `who()` следует вызвать. Поскольку `p` указывает на объект типа **D1**, выполняется вариант `who()` из этого класса. Аналогично, когда `p` присваивается адрес `D2_obj`, выполняется вариант `who()`, объявленный внутри **D2**.

Подытожим: если виртуальная функция вызывается посредством указателя базового класса, выбор фактически выполняемого варианта виртуальной функции осуществляется во время выполнения, исходя из типа объекта, на который указывает этот указатель.

Хотя виртуальные функции обычно вызываются посредством указателей базового класса, их можно также вызывать обычным образом, используя синтаксис с оператором-точкой. Отсюда следует, что в предыдущем примере было бы синтаксически правильно вызвать `who()` с помощью такого предложения:

```
D1_obj.who();
```

Однако вызов виртуальной функции таким образом игнорирует ее полиморфные атрибуты. Только когда виртуальная функция вызывается посредством указателя базового класса (или ссылки на базовый класс), достигается полиморфизм времени выполнения.

На первый взгляд описанное выше переопределение виртуальной функции в производном классе представляет собой особую форму пе-

перегрузки функции. Однако это не так. Фактически эти два средства фундаментально различаются. Прежде всего, перегруженная функция должна отличаться числом или типом своих параметров, в то время как переопределенная виртуальная функция должна иметь в точности то же число и те же типы параметров. Прототипы виртуальной функции и ее переопределений должны в точности совпадать. Если прототипы различаются, тогда функция рассматривается просто как перегруженная, и ее виртуальная сущность теряется. Другое ограничение заключается в том, что виртуальная функция обязана быть членом, а не другом класса, для которого она определяется. Однако виртуальная функция может быть другом другого класса. Наконец, виртуальными могут быть деструкторы, но не конструкторы.

Виртуальные функции наследуются

После того, как функция объявлена виртуальной, она остается таковой независимо от того, через сколько производных классов она прошла. Например, если класс **D2** объявлен производным не от **B**, а от **D1**, как это показано в следующем фрагменте, тогда функция **who()** все равно остается виртуальной:

```
// D2 производный от D1, а не от B.
class D2 : public D1 {
public:
    void who() { // Определение who() в классе D2
        cout << "Второй производный класс\n";
    }
};
```

Если производный класс не переопределяет виртуальную функцию, тогда используется функция в том виде, в каком она определена в базовом классе. В качестве примера испытайте приведенный ниже вариант предыдущей программы. В нем класс **D2** не переопределяет функцию **who()**:

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

class B {
public:
    virtual void who() {
        cout << "Базовый класс\n";
    }
};
```

```
class D1 : public B {
public:
    void who() {
        cout << "Первый производный класс\n";
    }
};

class D2 : public B { ← D2 не переопределяет who().
    // who() не определена
};

int main()
{
    B base_obj;
    B *p;
    D1 D1_obj;
    D2 D2_obj;

    p = &base_obj;
    p->who(); // обращение к who() из B

    p = &D1_obj;
    p->who(); // обращение к who() из D1

    p = &D2_obj;
    p->who(); /* обращение к who() из B, потому что
               D2 не переопределил who() */

    return 0;
}
```

Здесь вызывается **who()**,
которая определена в **B**.

Вот вывод этой программы:

```
Базовый класс
Первый производный класс
Базовый класс
```

Из-за того, что класс **D2** не переопределил **who()**, используется вариант **who()**, определенный в базовом классе **B**.

Не забывайте, что наследование виртуальных функций имеет иерархический характер. Если последний пример изменить так, чтобы **D2** был производным не от **B**, а от **D1**, тогда при вызове функции **who()** для объекта типа **D2** будет вызываться не **who()** из класса **B**, а вариант **who()**, объявленный внутри **D1**, потому что класс **D1** является ближайшим классом к **D2** в иерархии классов.

Зачем нужны виртуальные функции?

Как уже отмечалось ранее, виртуальные функции в сочетании с производными классами позволяют C++ поддерживать полиморфизм времени выполнения. Полиморфизм весьма важен для объектно-ориентированного программирования, потому что он дает возможность обобщенному классу задать те функции, которые будут общими для всех классов, производных от этого класса, и, в то же время, позволяет каждому производному классу определить специфическую реализацию всех или некоторых из этих функций. Иногда эту идею выражают таким образом: базовый класс диктует общий *интерфейс* для любого объекта, производного от базового класса, но позволяет производным классам определять фактический *метод*, используемый для реализации этого интерфейса. Именно поэтому для описания полиморфизма часто используют фразу “один интерфейс, множество методов”.

Успешное практическое приложение полиморфизма требует понимания того, что базовые и производные классы образуют иерархию, в которой можно проследить движение от большей к меньшей обобщенности (от базовых классов к производным). Правильно разработанный базовый класс предоставляет все элементы, которые производный класс может использовать непосредственно. Он также определяет те функции, которые производный класс должен реализовать самостоятельно. Такой подход придает производному классу гибкость в определении собственных методов, и, тем не менее, навязывает согласованный, единообразный интерфейс. Единообразие означает, что поскольку форма интерфейса определена в базовом классе, все производные классы будут использовать один и тот же общий интерфейс. Таким образом, использование виртуальных функций предоставляет возможность базовому классу определить обобщенный интерфейс, который будет использоваться всеми производными классами.

Вы, конечно, можете задать вопрос, почему единообразный интерфейс с множеством реализаций так важен. Ответ опять возвращает нас к главной движущей силе объектно-ориентированного программирования: оно помогает программистам справляться с программами неуклонно возрастающей сложности. Если, например, вы правильно разработали свою программу, то вы знаете, что ко всем объектам, наследуемым от базового класса, обращение будет происходить единообразно, несмотря на то, что их конкретные действия будут варьироваться от одного производного класса к другому. В результате вам надо будет иметь дело только с одним интерфейсом, а не с несколькими. Кроме того, ваш производный класс может использовать все или любые функциональные возможности, предоставляемые базовым классом. Вам не надо заново изобретать эти возможности.

Отделение интерфейса от реализации является также основой для создания *библиотек классов*, которые могут разрабатываться сторонними фирмами. Если эти библиотеки реализованы правильно, они предоставляют общий интерфейс, который вы можете использовать при разработке собственных производных классов, отвечающих вашим конкретным требованиям. Например, библиотеки классов Microsoft Foundation Classes (MFC) и более новая разработка NET Framework Windows Forms поддерживают программирование в системе Windows. Используя классы этих библиотек, вы наследуете значительную долю функциональных возможностей, требуемых от Windows-программ. Вам только остается добавить элементы, необходимые для вашего конкретного приложения. При программировании сложных систем такой подход имеет огромные преимущества.

Приложение виртуальных функций

Для того, чтобы продемонстрировать мощь виртуальных функций, мы используем их в классе **TwoDShape**. В предыдущих примерах каждый класс, производный от **TwoDShape**, определял функцию с именем **area()**. Это наводит на мысль, что было бы лучше сделать **area()** виртуальной функцией класса **TwoDShape**, позволив каждому производному классу переопределить эту функцию, поместив в нее формулу для вычисления площади того типа фигур, которые инкапсулирует данный класс. Приводимая ниже программа иллюстрирует этот подход. Кроме того, в ней для удобства в класс **TwoDShape** включено поле с названием фигуры. (Это облегчает демонстрацию классов.)

```
// Использование виртуальных функций и полиморфизма.
```

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

// Класс для двумерных объектов.
class TwoDShape {
    // эти члены закрыты
    double width;
    double height;
    // добавим поле для названия фигуры
    char name[20];
public:
```



```

// Конструктор по умолчанию.
TwoDShape() {
    width = height = 0.0;
    strcpy(name, "неизвестно");
}

// Конструктор для TwoDShape.
TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
}

// Конструируем объект с равными шириной и высотой.
TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
}

void showDim() {
    cout << "Ширина и высота составляют " <<
        width << " и " << height << "\n";
}

// функции доступа
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }


// Добавим в TwoDShape функцию area() и сделаем
// ее виртуальной.
virtual double area() {
    cout << "Ошибка: area() должна быть переопределена.\n";
    return 0.0;
}

};

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:
    /* Конструктор по умолчанию. Он автоматически вызывает
       конструктор по умолчанию TwoDShape. */

```

Функция **area()**
теперь виртуальная.



```
Triangle() {
    strcpy(style, "неизвестно");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
          double h) : TwoDShape(w, h, "треугольник") {
    strcpy(style, str);
}

// Конструируем равнобедренный треугольник.
Triangle(double x) : TwoDShape(x, "треугольник") {
    strcpy(style, "равнобедренный");
}

// Переопределяем функцию area(), объявленную в TwoDShape.
double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Треугольник " << style << "\n";
}
};

// Производный от TwoDShape класс для прямоугольников.
class Rectangle : public TwoDShape {
public:

    // Конструируем прямоугольник.
    Rectangle(double w, double h) :
        TwoDShape(w, h, "прямоугольник") { }

    // Конструируем квадрат.
    Rectangle(double x) :
        TwoDShape(x, "прямоугольник") { }
    bool isSquare() {
        if(getWidth() == getHeight()) return true;
        return false;
    }

    // Это другое переопределение функции area().
    double area() {
        return getWidth() * getHeight();
    }
};
```

← Переопределение area() в Triangle

← Переопределение area() в Rectangle

```


int main() {
    // объявим массив указателей на объекты TwoDShape.
    TwoDShape *shapes[5];

    shapes[0] = &Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);
    shapes[4] = &TwoDShape(10, 20, "обобщенную фигуру");

    for(int i=0; i < 5; i++) {
        cout << "объект представляет собой " <<
            shapes[i]->getName() << "\n";
        cout << "Площадь равна " <<
            shapes[i]->area() << "\n";
        cout << "\n";
    }

    return 0;
}

```


 Теперь для каждого объекта вызывается
 правильный вариант функции **area()**

Ниже приведен вывод этой программы:

объект представляет собой треугольник
 Площадь равна 48

объект представляет собой прямоугольник
 Площадь равна 100

объект представляет собой прямоугольник
 Площадь равна 40

объект представляет собой треугольник
 Площадь равна 24.5

объект представляет собой обобщенную фигуру
 Ошибка: area() должна быть переопределена.
 Площадь равна 0

Рассмотрим программу в деталях. Прежде всего, **area()** объявлена как виртуальная в классе **TwoDShape** и переопределена в классах **Triangle** и **Rectangle**. Внутри **TwoDShape** для функции **area()** предусмотрена реализация-заготовка, которая просто информирует пользователя, что функция должна быть переопределена в производном классе. Каждое переопределение **area()** представляет реализацию, отвечающую типу объекта, инкапсулированного в данном производном клас-

се. Таким образом, если бы вы, например, реализовывали класс эллипсов, тогда `area()` должна была бы вычислять площадь эллипса.

В рассмотренной программе имеется одно важное место. Обратите внимание на то, что переменная `shapes` в `main()` объявлена как массив указателей на объекты `TwoDShape`. Однако элементам этого массива присвоены указатели на объекты `Triangle`, `Rectangle` и `TwoDShape`. Это допустимо, потому что указатель базового класса может указывать на объект производного класса. Программа затем просматривает этот массив в цикле, выводя информацию о каждом объекте. Будучи весьма простым, этот фрагмент тем не менее иллюстрирует мощь как наследования, так и виртуальных функций. Тип объекта, на который указывает указатель базового класса, определяется во время выполнения, и дальнейшие действия зависят от этого типа. Если объект является производным от `TwoShape`, его площадь можно получить, вызвав функцию `area()`, причем интерфейс этой операции одинаков, для какого бы типа фигуры он не использовался.

Минутная тренировка

1. Что такое виртуальная функция?
 2. В чем важность виртуальных функций?
 3. Если переопределенная виртуальная функция вызывается посредством указателя базового класса, какой вариант функции будет выполняться?
 1. Виртуальной называется функция, объявленная с описателем `virtual` в базовом классе, переопределенная в производном классе
 2. Виртуальные функции предоставляют один из способов поддержки полиморфизма в C++
 3. Вариант виртуальной функции, который будет выполняться, определяется типом объекта, на который указывает указатель базового класса к моменту вызова функции. Таким образом, выбор функции осуществляется во время выполнения
-

Цель

10.10. Чистые виртуальные функции и абстрактные классы

Иногда вам может понадобиться создать базовый класс, который определяет только обобщенную форму, доступную всем производным классам, имея в виду, что она затем будет конкретизироваться в каждом производном классе. Такой класс определяет состав функций, которые должны быть реализованы в производных классах, но сам себе не предоставляет реализации всех этих функций или некоторой из них. Ситуация такого рода возникает, например, когда базовый класс не может предложить имеющую смысл реализацию функции. Именно это имело место в последнем варианте `TwoDShape`. Определ

ние функции `area()` было просто шаблоном. Этот вариант функции не вычислит и не выведет на экран площадь какой бы то ни было фигуры.

Как вы убедитесь сами, если будете создавать собственные библиотеки классов, такая ситуация, когда функция в контексте базового класса не может иметь разумной реализации, встречается довольно часто. Она может быть разрешена двумя способами. Один способ, продемонстрированный в нашем примере, заключается в реализации функции, которая будет просто выводить предупреждающее сообщение. Хотя в некоторых ситуациях — например, при отладке программы — такой подход может быть полезен, обычно к нему не прибегают. Вы можете столкнуться с функциями, которые должны быть обязательно переопределены в производном классе, чтобы этот класс имел смысл. Вспомним класс `Triangle`. Он не имеет смысла, пока в нем не определена функция `area()`. В таких случаях вы должны иметь какой-то способ, который позволит вам убедиться, что производный класс действительно переопределил все необходимые функции. Решение этой проблемы в C++ заключается в использовании *чистых виртуальных функций*.

Чистая виртуальная функция — это функция, объявленная в базовом классе и не имеющая в этом классе своего определения. В этом случае каждый производный класс должен обязательно определить собственный вариант функции — он не может использовать функцию, которая не имеет определения. Для объявления чистой виртуальной функции используется такая общая форма:

```
virtual тип имя-функции(список-параметров) = 0;
```

Здесь *тип* представляет собой тип возвращаемого функцией значения, а *имя-функции* — ее имя.

Используя чистые виртуальные функции, вы можете улучшить класс `TwoDShape`. Поскольку для неопределенной двумерной фигуры нельзя предложить разумный способ вычисления площади, в приведенном ниже варианте предыдущей программы функция `area()` объявлена внутри класса `TwoDShape` как чистая виртуальная. Это, разумеется, означает, что все классы, производные от `TwoDShape`, должны переопределять `area()`.

```
// Использование чистой виртуальной функции."
```

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

```
// Класс для двумерных объектов.
class TwoDShape {
    // эти члены закрыты
    double width;
    double height;
```

```
// добавим поле для названия фигуры
char name[20];
public:

// Конструктор по умолчанию.
TwoDShape() {
    width = height = 0.0;
    strcpy(name, "неизвестно");
}

// Конструктор для TwoDShape.
TwoDShape(double w, double h, char *n) {
    width = w;
    height = h;
    strcpy(name, n);
}

// Конструктор для объекта с равными шириной и высотой.
TwoDShape(double x, char *n) {
    width = height = x;
    strcpy(name, n);
}

void showDim() {
    cout << "Ширина и высота составляют " <<
        width << " и " << height << "\n";
}

// функции доступа
double getWidth() { return width; }
double getHeight() { return height; }
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
char *getName() { return name; }

// area() теперь чистая виртуальная функция ← area() теперь яв-
virtual double area() = 0;                       ляется чистой вир-
};                                                  туальной функцией

// Triangle является производным от TwoDShape.
class Triangle : public TwoDShape {
    char style[20]; // теперь private
public:
    /* Конструктор по умолчанию. Он автоматически вызывает
       конструктор по умолчанию TwoDShape. */
    Triangle() {
```

```

    strcpy(style, "неизвестно");
}

// Конструктор с тремя параметрами.
Triangle(char *str, double w,
         double h) : TwoDShape(w, h, "треугольник") {
    strcpy(style, str);
}

// Конструируем равнобедренный треугольник.
Triangle(double x) : TwoDShape(x, "треугольник") {
    strcpy(style, "равнобедренный");
}

// Теперь это переопределяет area(), объявленную в
// TwoDShape.
double area() {
    return getWidth() * getHeight() / 2;
}

void showStyle() {
    cout << "Треугольник " << style << "\n";
}
};

// Производный от TwoDShape класс для прямоугольников.
class Rectangle : public TwoDShape {
public:

    // Конструируем прямоугольник.
    Rectangle(double w, double h) :
        TwoDShape(w, h, "прямоугольник") { }

    // Конструируем квадрат.
    Rectangle(double x) :
        TwoDShape(x, "прямоугольник") { }

    bool isSquare() {
        if(getWidth() == getHeight()) return true;
        return false;
    }

    // Это другое переопределение функции area().
    double area() {
        return getWidth() * getHeight();
    }
};

```

```
int main() {
    // объявим массив указателей на объекты TwoDShape.
    TwoDShape *shapes[4];

    shapes[0] = &Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);

    for(int i=0; i < 4; i++) {
        cout << "объект представляет собой " <<
            shapes[i]->getName() << "\n";
        cout << "Площадь равна " <<
            shapes[i]->area() << "\n";

        cout << "\n";
    }

    return 0;
}
```

Если класс имеет хотя бы одну чистую виртуальную функцию, про него говорят, что это *абстрактный класс*. Абстрактный класс имеет одну важную черту: нельзя создать объект такого класса. Чтобы убедиться в справедливости этого утверждения, попробуйте удалить переопределение `area()` из класса `Triangle` последней программы. При попытке создать экземпляр `Triangle` вы получите сообщение об ошибке. Абстрактный класс может быть использован только в качестве базового, от которого будут наследовать другие классы. Причина невозможности объявить объект абстрактного класса заключается в том, что одна или несколько его функций не имеют определений. Именно поэтому массив `shapes` в последней программе уменьшен до четырех элементов и не создается обобщенный объект `TwoDShape`. Как это видно из программы, если базовый класс является абстрактным, вы, тем не менее, можете объявить указатель этого класса, который будет затем использоваться для указания на объекты производных классов.

✓ Вопросы для самопроверки

1. Класс, который наследуется, называется _____ классом. Класс, который наследует, называется _____ классом
2. Имеет ли базовый класс доступ к членам производного класса? Имеет ли производный класс доступ к членам базового класса?

3. Создайте производный от **TwoDShape** класс с именем **Circle** (круг). Включите в него функцию **area()**, вычисляющую площадь круга.
4. Каким образом предотвратить доступ производного класса к члену базового класса?
5. Приведите общую форму конструктора, который вызывает конструктор базового класса.
6. Пусть имеется такая иерархия классов:

```
class Alpha { ...
```

```
class Beta : public Alpha { ...
```

```
class Gamma : public Beta { ...
```

В каком порядке вызываются конструкторы этих классов, когда создается объект **Gamma**?

7. Откуда можно обращаться к членам класса с атрибутом **protected**?
8. Указатель базового класса может указывать на объект производного класса. Объясните, почему это важно применительно к переопределению функций.
9. Что такое чистая виртуальная функция? Что такое абстрактный класс?
10. Можно ли создать экземпляр объекта абстрактного класса?
11. Объясните, каким образом чистые виртуальные функции помогают реализации принципа полиморфизма “один интерфейс, множество методов”.

Модуль

11

C++ и система ВВОДА-ВЫВОДА

Цели, достигаемые в этом модуле

- 11.1 Понять, что такое потоки ввода-вывода
- 11.2 Узнать о иерархии классов ввода-вывода
- 11.3 Освоить перегрузку операторов << и >>
- 11.4 Изучить форматирование ввода-вывода с помощью функций-членов ios
- 11.5 Научиться форматировать ввод-вывод с помощью манипуляторов
- 11.6 Начать создавать собственные манипуляторы
- 11.7 Научиться открывать и закрывать файлы
- 11.8 Освоить чтение и запись текстовых файлов
- 11.9 Освоить чтение и запись двоичных файлов
- 11.10 Больше узнать о функциях ввода-вывода
- 11.11 Познакомиться со случайным доступом к файлам
- 11.12 Узнать, как можно получить состояние системы ввода-вывода

С самого начала этой книги вы использовали систему ввода-вывода C++, однако делали это без должного формального описания этой системы. Поскольку система ввода-вывода основывается на иерархии классов, не было возможности представить ее теорию и детали без предварительного обсуждения классов и наследования. Теперь наступило время изучить систему ввода-вывода в деталях.

Система ввода-вывода C++ весьма велика, и невозможно описать в этой книге все ее классы, функции и средства; здесь мы только познакомим вас с наиболее важными и используемыми чаще других элементами этой системы. В частности, вы узнаете, как можно перегрузить операторы << и >>, чтобы получить возможность вводить и выводить объекты разработанных вами классов. Вы также познакомитесь с форматами и манипуляторами ввода-вывода. В конце модуля мы обсудим файловый ввод-вывод.

Старая и новая системы ввода-вывода

В настоящее время используются две версии объектно-ориентированной библиотеки ввода-вывода C++: более старая, основанная на исходной спецификации C++, и новая, определенная в стандартном C++. Старая библиотека ввода-вывода поддерживается заголовочным файлом `<iostream.h>`. Новая библиотека ввода-вывода поддерживается заголовком `<iostream>`. Обе библиотеки с точки зрения программиста представляются в значительной степени одинаковыми. Это объясняется тем, что новая библиотека ввода-вывода по существу является просто модернизированной и улучшенной версией старой. Фактически большая часть различий в этих библиотеках лежит под поверхностью и определяется тем, как они реализованы, а не тем, как они используются.

С точки зрения программиста между старой и новой библиотекой имеются два основных различия. Во-первых, новая библиотека ввода-вывода содержит несколько дополнительных средств и определяет некоторые новые типы данных. Таким образом, новая библиотека определена надмножеством старой. Практически все программы, написанные для старой библиотеки ввода-вывода, будут компилироваться без существенных изменений в случае использования новой библиотеки. Во-вторых, старая библиотека ввода-вывода находилась в глобальном пространстве имен. Новая же библиотека находится в пространстве имен `std`. (Пространство имен `std` используется всеми библиотеками стандартного C++.) Поскольку старая библиотека ввода-вывода теперь устарела, в этой книге описывается только новая библиотека, хотя большая часть приводимых данных будет с равным успехом относиться и к старой.

Цель

11.1. Потоки C++

Важнейшей чертой системы ввода-вывода C++ является то, что она оперирует над *потоками*. Поток является абстрактным объектом, который либо создает, либо поглощает информацию. Поток связывается с физическим устройством посредством системы ввода-вывода C++. Все потоки ведут себя одинаковым образом, одни и те же функции и операторы ввода-вывода применимы практически ко всем типам устройств. Например, метод, с помощью которого вы выводите данные на экран, может быть использован для записи данных на диск или печати их на принтере.

В наиболее общей форме поток является логическим интерфейсом с файлом. Согласно определению в C++, термин “файл” может относиться к клавиатуре, порту, файлу на ленте и т. д. Хотя файлы различаются по форме и возможностям, все потоки одинаковы. Преимущество такого подхода заключается в том, что для вас, программиста, одно аппаратное устройство будет выглядеть в значительной степени так же, как и любое другое. Поток обеспечивает единообразный интерфейс.

Поток связывается с файлом посредством операции открытия. Поток отсоединяется от файла посредством операции закрытия.

Имеются два типа потоков: *текстовые* и *двоичные*. Текстовый поток состоит из символов. При использовании текстового потока может иметь место то или иное преобразование символов. Например, при выводе символа новой строки он может быть преобразован в последовательность возврат каретки — перевод строки. По этой причине между тем, что посылается в поток, и тем, что записывается в файл, может и не быть точного соответствия. Двоичные потоки могут использоваться с данными любых типов. В этом случае никакого преобразования не происходит, и между тем, что посылается в поток, и тем, что фактически содержится в файле, имеется полное соответствие.

Еще одним важным понятием является *текущая позиция*. Текущая позиция — это позиция в файле, к которой будет выполняться очередное обращение при чтении или записи файла. Если, например, файл имеет длину 100 байт, и половина файла уже прочитана, следующая операция чтения обратится к байту 50, который и является в данный момент текущей позицией.

Подытожим. В C++ ввод-вывод осуществляется посредством логического интерфейса, называемого *поток*ом. Все потоки имеют схожие свойства, и операции над любым потоком выполняются одними и теми же функциями ввода-вывода, независимо от того, с какого рода файлом связан поток. Файл является физическим устройством, которое содержит данные. Хотя файлы и различаются, все потоки одинаковы. (Разумеется, конкретные устройства могут и не обеспечивать все

возможные операции, например, операции произвольного доступа, и тогда связанные с ними потоки также эти операции поддерживать не будут.)

Предопределенные потоки C++.

C++ содержит несколько предопределенных потоков, которые открываются автоматически, когда начинается выполняться ваша C++-программа. Эти потоки называются **cin**, **cout**, **cerr** и **clog**. Как вы уже знаете, **cin** — это поток, связанный со стандартным вводом, а **cout** — поток, связанный со стандартным выводом. Поток **cerr** также связан со стандартным выводом, как и **clog**. Разница между двумя этими потоками заключается в том, что **clog** буферизован, а **cerr** нет. Это означает, что любые данные, посылаемые в поток **cerr**, выводятся немедленно, а вывод в **clog** записывается на устройство только после заполнения буфера. Обычно потоки **cerr** и **clog** используются программой для записи отладочной информации и информации об ошибках.

C++ также открывает варианты стандартных потоков, предназначенных для работы с “широкими” (16-битовыми) символами: **wcin**, **wcout**, **wcerr** и **wclog**. Эти потоки призваны поддерживать такие языки, как, например, китайский, которые используют большие символьные наборы. Мы не будем использовать их в этой книге.

По умолчанию стандартные потоки C++ связаны с консолью, но ваша программа может перенаправить их на другие устройства или файлы. Они могут быть также перенаправлены операционной системой.

Цель

11.2. Потоковые классы C++

Как вы узнали из Модуля 1, C++ обеспечивает поддержку своей системы ввода-вывода в заголовке `<iostream>`. В нем определена довольно сложная система иерархий классов для поддержки операций ввода-вывода. Классы ввода-вывода начинаются с системы шаблонных классов. Как вы узнаете из Модуля 12, шаблон определяет форму класса без полной спецификации данных, над которыми он будет производить операции. После того, как шаблонный класс определен, могут быть созданы конкретные экземпляры этого класса. Если говорить о библиотеке ввода-вывода, стандартный C++ создает два специфических варианта этих шаблонных классов: один для 8-битовых символов и другой для “широких”. Эти специфические варианты действуют, как и любые другие классы, и для полноцен-

ного использования системы ввода-вывода C++ нет необходимости знать о шаблонах.

Система ввода-вывода C++ построена на двух связанных, но различных иерархиях шаблонных классов. Первая наследуется от низкоуровневого класса ввода-вывода, называемого **basic_streambuf**. Этот класс обеспечивает операции элементарного, низкоуровневого ввода-вывода и предоставляет базовую поддержку для всей системы ввода-вывода C++. Использовать класс **basic_streambuf** непосредственно вам может понадобиться, только если вы будете заниматься системным программированием ввода-вывода. Иерархия классов, с которой вам чаще всего придется сталкиваться, наследуется от класса **basic_ios**. Это класс ввода-вывода высокого уровня, который предоставляет форматирование, обнаружение ошибок и информацию о состоянии при работе с потоковым вводом-выводом. (Базовый класс для **basic_ios** называется **ios_base**; в нем определены некоторые характерные средства, используемые классом **basic_ios**.) Класс **basic_ios** используется в качестве базового для нескольких производных классов, включая **basic_istream**, **basic_ostream** и **basic_iostream**. Эти классы используются для создания потоков, обеспечивающих ввод, вывод и ввод-вывод соответственно.

Как уже говорилось, библиотека ввода-вывода содержит два специфических варианта иерархий классов ввода-вывода: одна для 8-битовых символов, а другая для “широких”. В этой книге обсуждаются только классы для 8-битовых символов, поскольку именно они используются наиболее часто. Ниже приведен список соответствий имен шаблонных классов их символьно-ориентированных вариантов.

| Имя шаблонного класса | Эквивалентное имя символьно-ориентированного класса |
|------------------------|---|
| basic_streambuf | streambuf |
| basic_ios | ios |
| basic_istream | istream |
| basic_ostream | ostream |
| basic_iostream | iostream |
| basic_fstream | fstream |
| basic_ifstream | ifstream |
| basic_ofstream | ofstream |

На протяжении всей оставшейся части этой книги будут использоваться символьно-ориентированные имена, поскольку именно этими именами вы будете пользоваться в своих программах. Эти же самые имена использовались и в старом варианте библиотеки ввода-вывода. Именно поэтому старая и новая библиотеки ввода-вывода совместимы на уровне исходных текстов программ.

Последнее замечание: класс `ios` содержит большое количество функций-членов и переменных для управления и контроля фундаментальных потоковых операций. Мы будем часто обращаться к этому классу. Не забывайте, что если вы включаете в свою программу заголовок `<iostream>`, вы получаете доступ к этому важному классу.

Минутная тренировка

1. Что такое поток? Что такое файл?
 2. Какой поток связан со стандартным выводом?
 3. Ввод-вывод C++ поддерживается сложным набором иерархий классов. Правильно ли это?
1. Поток называется логический абстрактный объект, который либо производит, либо потребляет информацию. Файл – это физический объект, содержащий данные
 2. Поток, связанный со стандартным выводом, называется `cout`
 3. Правильно Ввод-вывод C++ поддерживается сложным набором иерархий классов
-

Цель

11.3. Перегрузка операторов ввода-вывода

В предыдущих модулях, когда программе нужно было вывести или ввести данные, связанные с классом, создавались функции-члены, единственным назначением которых был вывод или ввод данных для этого класса. Хотя в этом подходе нет ничего неправильного, C++ предлагает гораздо лучший способ выполнения операций ввода-вывода для классов: перегрузку операторов ввода-вывода `<<` и `>>`.

В языке C++ оператор `<<` называется *оператором вставки*, потому что он вставляет данные в поток. Аналогично, оператор `>>` называется *оператором извлечения*, потому что он извлекает данные из потока. Операторные функции, перегружающие операторы вставки и извлечения, называют соответственно функциями вставки и извлечения или просто функциями вывода и ввода.

В заголовке `<iostream>` имеются перегруженные операторы вставки и извлечения для всех встроенных типов C++. Здесь вы увидите, как можно перегрузить эти операторы применительно к создаваемым вами классам.

Создание операторных функций вывода

В качестве простого примера давайте создадим оператор вывода (вставки) для приведенного ниже варианта класса `ThreeD`:

```
class ThreeD {
public:
    int x, y, z; // трехмерные координаты
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};
```

Чтобы создать функцию вывода для объекта типа **ThreeD**, необходимо перегрузить для него оператор <<. Вот как это делается:

```
// Вывод координат X, Y, Z - функция вывода для ThreeD.
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возврат потока stream
}
```

Рассмотрим приведенный фрагмент внимательно, так как многие его детали будут повторяться во всех функция вывода. Прежде всего заметим, что наша функция, в соответствии с ее объявлением, возвращает объект типа **ostream**. Такое объявление делает возможным объединять операторы вывода этого типа в составных выражениях ввода-вывода. Далее, функция имеет два параметра. Первый представляет собой ссылку на поток, который указывается с левой стороны оператора <<. Второй параметр — это объект, указываемый с правой стороны оператора. (Этот параметр может также представлять собой ссылку на объект, если вам это больше нравится.) Внутри функции выводятся в поток три значения, содержащиеся в объекте типа **ThreeD**, и локальная переменная **stream** возвращается.

Вот короткая программа, демонстрирующая создание и использование оператора вывода:

```
/* Демонстрация перегруженного оператора вывода
   для конкретного класса.*/

#include <iostream>
using namespace std;

class ThreeD {
public:
    int x, y, z; // трехмерные координаты
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
};

// Вывод координат X, Y, Z - оператор вывода для ThreeD.
```



```
ostream &operator<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возврат stream
}

int main()
{
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c;

    return 0;
}
```

Оператор вывода для класса **ThreeD**.

Используем оператор вывода класса **ThreeD** для вывода координат

Вывод этой программы выглядит таким образом:

```
1, 2, 3
3, 4, 5
5, 6, 7
```

Если вы удалите код, специфический для класса **ThreeD**, останется шаблон для функции вывода:

```
ostream &operator<(ostream &stream, class_type obj)
{
    // здесь располагается код, специфический для данного
    // класса
    return stream; // возврат stream
}
```

Разумеется, объект **obj** вполне допустимо передать по ссылке.

Действия, выполняемые функцией вывода, фактически могут быть какими угодно. Однако хороший стиль программирования требует, чтобы ваша функция вывода организовывала разумный вывод. Не забывайте, что она должна возвращать локальную переменную, названную в наших примерах **stream**.

Использование дружественных функций для перегрузки операторов вывода

В предыдущей программе функция перегрузки оператора вывода не являлась членом **ThreeD**. В самом деле, ни функции ввода, ни

функции вывода не могут быть членами класса. Причина такого запрета заключается в том, что если функция **operator** является членом класса, ее левый операнд (передаваемый неявно посредством указателя **this**) должен быть объектом того класса, который вызывает эту функцию. Изменить это требование нельзя. Однако, когда мы перегружаем операторы ввода, левым операндом является поток, а правым — объект класса, для которого осуществляется операция вывода. Поэтому перегруженные операторы вывода должны быть функциями-не членами.

То, что перегруженные операторы вывода не могут быть членами класса, для которого они определяются, вызывает серьезный вопрос: каким образом перегруженный оператор вывода может обратиться к закрытым элементам класса? В предыдущей программе переменные *x*, *y* и *z* были сделаны открытыми, чтобы оператор вывода мог к ним обратиться. Однако сокрытие данных является важной частью ООП, и вынужденное открытие данных является серьезным нарушением его принципов. Для решения возникшей проблемы имеется решение: перегруженный оператор вывода может быть дружественной функцией класса. Как друг класса, для которого он определяется, этот оператор имеет доступ к закрытым данным. Ниже приведен новый вариант класса **ThreeD** и использующей его программы, где перегруженный оператор вывода объявлен дружественной функцией:

// Использование дружественной функции для перегрузки <<.

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

class ThreeD {
    int x, y, z; // трехмерные координаты - теперь private
public:
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
    friend ostream &operator<<(ostream &stream, ThreeD obj) <←
};

// Вывод координат X, Y, Z - оператор вывода для ThreeD.
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возврат stream
}

int main()
```

Оператор вывода класса **ThreeD**
теперь является другом и имеет
доступ к закрытым данным.

```

{
    ThreeD a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c;

    return 0;
}

```

Обратите внимание на переменные *x*, *y* и *z*, которые теперь объявлены закрытыми, но, тем не менее, непосредственно доступны для перегруженного оператора вывода. Объявление перегруженных операторов вывода (и ввода) друзьями классов, для которых они определяются, хорошо согласуется с принципами инкапсуляции ООП.

Перегрузка операторов ввода

Для перегрузки оператора ввода используется тот же самый общий подход, продемонстрированный выше применительно к операции вывода. Например, приведенный ниже перегруженный оператор ввода вводит трехмерные координаты. Обратите внимание на то, что он также выводит запрос пользователю:

```

// Получение трехмерных координат - оператор ввода для
// ThreeD.
istream &operator>>(istream &stream, ThreeD &obj)
{
    cout << "Введите значения X,Y,Z: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}

```

Оператор ввода должен возвращать ссылку на объект типа **istream**. Кроме того, в качестве первого параметра должна использоваться ссылка на объект типа **istream**. Это объект потока, указываемый с левой стороны оператора **>>**. Второй параметр представляет собой ссылку на переменную, которая будет принимать ввод. Поскольку это ссылка, второй параметр может модифицироваться в процессе ввода данных.

Шаблон для перегруженного оператора ввода выглядит так:

```

istream &operator>>(istream &stream, object_type &obj)
{
    // здесь размещается код вашей функции ввода
}

```

```
    return stream;
}
```

Приведенная ниже программа демонстрирует создание и использование оператора ввода для объектов типа **ThreeD**:

// Демонстрация прикладного перегруженного оператора ввода.

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

```
class ThreeD {
    int x, y, z; // трехмерные координаты
public:
    ThreeD(int a, int b, int c) { x = a; y = b; z = c; }
    friend ostream &operator<<(ostream &stream, ThreeD obj);
    friend istream &operator>>(istream &stream, ThreeD &obj);
};
```

// Вывод координат X, Y, Z – оператор вывода для ThreeD.

```
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // возврат stream
}
```

// Получение трехмерных координат – оператор ввода для ThreeD.

```
istream &operator>>(istream &stream, ThreeD &obj)
{
    cout << "Введите значения X,Y,Z: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

←
Перегруженный оператор
ввода для **ThreeD**

```
int main()
{
    ThreeD a(1, 2, 3);

    cout << a;

    cin >> a;
    cout << a;

    return 0;
}
```

Вывод одного из прогонов программы выглядит так:

1, 2, 3

Введите значения X, Y, Z: 5 6 7

5, 6, 7

Как и перегруженные операторы вывода, перегруженные операторы ввода не могут быть членами класса, для операций с которым они предназначены. Они могут быть либо друзьями класса, либо просто независимыми функциями.

За исключением того, что вы должны вернуть ссылку на объект типа **istream**, вы можете внутри функции ввода делать все, что вам заблагорассудится. Однако ради ясности и удобства чтения программы лучше использовать перегруженные операторы ввода исключительно для операций ввода.

Минутная тренировка

1. Что такое функция вывода?
 2. Что такое функция ввода?
 3. Почему для создания перегруженных операторов ввода и вывода часто используются дружественные функции?
1. Функция вывода помещает данные в поток
 2. Функция ввода извлекает данные из потока
 3. Дружественные функции часто используются для создания перегруженных операторов ввода и вывода, потому что они имеют доступ к закрытым данным класса
-

Форматированный ввод-вывод

До сих пор при вводе и выводе мы неявным образом использовали форматные данные, поставляемые системой ввода-вывода C++ по умолчанию. Вы, однако, можете детально управлять форматом ваших данных, для чего имеются два способа. Первый использует функции-члены класса **ios**. Второй работает со специальными функциями, называемыми *манипуляторами*. Мы начнем с рассмотрения функций-членов **ios**.

Цель

11.4. Форматирование с помощью функций-членов **ios**

С каждым потоком связан набор флагов формата, которые задают способ форматирования данных в потоке. В классе **ios** объявлен перечисли-

мый тип битовой маски с именем **fmtflags**, в котором определены приведенные ниже значения. (Формально они определены внутри **ios_base**, который, как уже отмечалось, является базовым классом для **ios**.)

| adjustfield | basefield | boolalpha | dec |
|-------------|------------|-----------|------------|
| fixed | floatfield | hex | internal |
| left | oct | right | scientific |
| showbase | showpoint | showpos | skipws |
| unitbuf | uppercase | | |

С помощью этих значений устанавливаются и сбрасываются флаги формата. Некоторые старые компиляторы могут не распознавать перечислимый тип **fmtflags**; в этом случае флаги формата кодируются в виде длинных целых чисел.

Если флаг **skipws** установлен, то при вводе в поток отбрасываются ведущие пробельные символы (символы пробела, табуляции и новой строки). Если флаг **skipws** сброшен, то пробельные символы не отбрасываются.

Если флаг **left** установлен, то вывод выравнивается по левому краю. Если установлен **right**, вывод выравнивается по правому краю. При установке флага **internal** числовые значения дополняются пробелами до указанной ширины поля, причем пробелы включаются между знаком числа и самим числом. Если ни один из этих флагов не установлен, вывод по умолчанию выравнивается по правому краю.

Числовые значения выводятся по умолчанию в десятичной форме. Однако имеется возможность изменить систему счисления. Установка флага **oct** приводит к выводу чисел в восьмеричной системе; флаг **hex** устанавливает шестнадцатеричную систему, а для возврата к десятичной форме вывода надо установить флаг **dec**.

Установка **showbase** включает в вывод обозначение системы счисления. Например, если установлена шестнадцатеричная система вывода, то значение 1F будет отображаться как 0x1F.

Если вывод чисел осуществляется в экспоненциальной форме, то по умолчанию выводится строчное **e**. Кроме того, строчной будет буква **x** при выводе шестнадцатеричных чисел. Если установить флаг **uppercase**, эти символы выводятся прописными буквами.

Установка **showpos** выводит знак плюс перед положительными значениями.

Установка **showpoint** приводит к выводу десятичной точки и завершающих нулей при выводе любых чисел с плавающей точкой, даже если в этом нет фактической необходимости.

При установке флага **scientific** числа с плавающей точкой будут выводиться в экспоненциальной форме. Если установлен **fixed**, числа с плавающей точкой выводятся обычным образом. Если ни один из этих флагов не установлен, компилятор сам выбирает подходящий способ вывода.

Если установлен **unitbuf**, после каждой операции вставки буфер очищается.

При установленном **boolalpha** булевы переменные вводятся и выводятся с использованием ключевых слов **true** и **false**.

Поскольку часто приходится ссылаться на поля **oct**, **dec** и **hex**, для их совокупности введено обозначение **basefield**. Аналогично поля **left**, **right** и **internal** обозначаются **adjustfield**. Наконец, к полям **scientific** и **fixed** можно обращаться с помощью ключевого слова **floatfield**.

Установка и сброс флагов формата

Для установки флага используется функция **setf()**, являющаяся членом класса **ios**. Ее прототип выглядит так:

```
fmtflags setf(fmtflags флаги);
```

Эта функция возвращает предыдущее значение флагов формата и устанавливает флаги, указанные параметром *флаги*. Например, чтобы установить флаг **showbase**, следует использовать такое предложение:

```
поток.setf(ios::showbase);
```

Здесь *поток* — это тот поток, на который вы хотите воздействовать. Обратите внимание на то, как **ios::** квалифицирует **showbase**. Поскольку **showbase** представляет собой константу перечислимого типа, определенного в классе **ios**, ее необходимо квалифицировать, т. е. указывать принадлежность к классу **ios** при любом ее использовании. Этот принцип приложим ко всем флагам формата.

Следующая программа использует **setf()** для установки флагов **showpos** и **scientific**:

```
// Использование setf().

#include <iostream>
using namespace std;

int main()
{
    // Установка флагов showpos и scientific.
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << " ";

    return 0;
}
```

Установка флагов формата
посредством **setf()**

Программа выведет следующее:

```
+123 +1.232300e+002
```

С помощью операции ИЛИ можно объединять в одном вызове `setf()` любое число флагов формата. Например, объединив флаги **scientific** и **showpos**, как это показано ниже, вы сможете ограничиться одним вызовом функции `setf()`:

```
cout.setf(ios::scientific | ios::showpos);
```

Для сброса флагов используется функция `unsetf()` со следующим прототипом:

```
void unsetf(fmtflags флаги);
```

Флаги, указанные параметром *флаги*, сбрасываются. (Все остальные флаги остаются без изменения.)

Иногда оказывается полезным узнать текущее состояние флагов формата. Вы можете определить текущие значения флагов с помощью функции `flags()` со следующим прототипом:

```
fmtflags flags( );
```

Эта функция возвращает текущее значение флагов в вызвавшем ее потоке.

Другая форма той же функции устанавливает флаг, заданный параметром *флаги*, и возвращает предыдущее значение этого флага:

```
fmtflags flags(fmtflags флаги);
```

Приведенная ниже программа демонстрирует использование `flags()` и `unset()`:

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

int main()
{
    ios::fmtflags f;

    f = cout.flags(); ←————— Получим флаги формата

    if(f & ios::showpos)
        cout << "showpos установлен для cout.\n";
    else
        cout << "showpos сброшен для cout.\n";
```



```

cout << "\nУстановим showpos для cout.\n";
cout.setf(ios::showpos); ← Установим флаг showpos

f = cout.flags();

if(f & ios::showpos)
    cout << "showpos установлен для cout.\n";
else
    cout << "showpos сброшен для cout.\n";

cout << "\nСбросим showpos для cout.\n";
cout.unsetf(ios::showpos); ← Сбросим флаг showpos.

f = cout.flags();
if(f & ios::showpos)
    cout << "showpos установлен для cout.\n";
else
    cout << "showpos сброшен для cout.\n";

return 0;
}

```

Программа выводит следующее:

showpos сброшен для cout.

Установим showpos для cout.
showpos установлен для cout.

Сбросим showpos для cout.
showpos сброшен для cout.

Заметьте, что в программе при объявлении переменной **f** тип **fmtflags** предварен указанием на класс **ios::**. Это необходимо делать, поскольку тип **fmtflags** определен в классе **ios**. Вообще, когда вы используете имя типа или перечислимой константы, которое определено в классе, вы должны квалифицировать его именем класса.

Установка ширины поля, точности и символа-заполнителя

Помимо флагов формата, имеются еще три функции-члена, определенные в **ios**, которые устанавливают следующие дополнительные характеристики формата: ширину поля, точность и символ-заполнитель. Эти характеристики устанавливаются функциями **width()**, **precision()** и **fill()** соответственно. Рассмотрим эти функции по очереди.

Когда осуществляется вывод значения, то по умолчанию оно занимает столько места, сколько требуется для отображения содержащего в нем числа символов. Однако вы можете указать минимальную ширину поля с помощью функции `width()`. Ее прототип выглядит так:

```
streamsize width(streamsize w);
```

Здесь `w` становится шириной поля, а возвращаемое значение представляет собой предыдущее значение ширины. В некоторых реализациях ширину поля необходимо устанавливать перед каждым выводом, иначе будет использоваться ширина по умолчанию. Тип `streamsize` определяется компилятором как некоторая форма целого.

После того, как вы установили минимальную ширину поля, происходит следующее. Если значение использует меньше символов, чем установленная ширина, то оставшаяся часть поля вывода будет заполнена текущим символом-заполнителем (по умолчанию пробелом). Если размер значения превышает минимальную ширину поля, поле будет расширено. Никакие значения не усекаются.

При выводе значений с плавающей точкой в экспоненциальной форме вы можете задать число цифр, выводимых после десятичной точки, с помощью функции `precision()`. Ее прототип выглядит так:

```
streamsize precision(streamsize p);
```

Здесь число цифр (точность) устанавливается равным `p`, и функция возвращает старое значение. Число цифр по умолчанию равно 6. В некоторых реализациях точность необходимо устанавливать перед каждым выводом значения с плавающей точкой. Если точность не установлена, используется значение по умолчанию.

По умолчанию, если поле нуждается в заполнении, оно заполняется пробелами. С помощью функции `fill()` вы можете задать символ-заполнитель. Прототип этой функции:

```
char fill(char ch);
```

После вызова `fill()` `ch` становится новым символом-заполнителем, а старое значение возвращается функцией.

Ниже приведена программа, демонстрирующая использование этих трех функций:

```
// Демонстрация функций width(), precision() и fill().
```

```
#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);    // в поле шириной 10 символов
    cout << 123 << " ";
    cout.width(10);    // установим ширину 10
    cout << 123.23 << "\n";

    cout.fill('#'); // заполнитель #
    cout.width(10); // в поле шириной 10 символов
    cout << 123 << " ";
    cout.width(10); // установим ширину 10
    cout << 123.23;

    return 0;
}
```

Установим точность
↓
Установим ширину поля
←
Установим символ-заполнитель

Программа выводит на экран следующее:

```
+123 +1.232300e+002
      +123 +1.23e+002
#####+123 +1.23e+002
```

В некоторых реализациях ширину поля необходимо устанавливать перед каждым выводом значения с плавающей точкой. Именно поэтому в приведенной программе функция **width()** вызывается многократно.

Функции **width()**, **precision()** и **fill()** имеют перегруженные формы, которые позволяют получать, но не изменять текущие установки. Эти формы приведены ниже:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Минутная тренировка

1. Для чего предназначен флаг **boolalpha**?
2. Для чего предназначена функция **setf()**?
3. Какая функция устанавливает символ-заполнитель?

- 1 Если флаг **boolalpha** установлен, булевы значения вводятся и выводятся с использованием слов **true** и **false**
- 2 **set(f)** устанавливает один или более флагов формата
- 3 Символ-заполнитель устанавливается функцией **fill()**

Цель

11.5. Использование манипуляторов ввода-вывода

Система ввода-вывода C++ предлагает и другой способ изменения параметров формата потока. В этом способе используются специальные функции, называемые *манипуляторами*, которые можно включать в выражения ввода-вывода. Стандартные манипуляторы перечислены в табл. 11-1. Для использования манипуляторов, принимающих аргументы, вы должны включить в программу заголовок `<iomanip>`.

Таблица 11-1. Манипуляторы ввода-вывода C++

| Манипулятор | Назначение | Ввод или вывод |
|--------------------------------|---|----------------|
| <code>boolalpha</code> | Устанавливает флаг boolalpha | Ввод/вывод |
| <code>dec</code> | Устанавливает флаг dec | Ввод/вывод |
| <code>endl</code> | Выводит символ новой строки и очищает поток | Вывод |
| <code>ends</code> | Выводит <code>null</code> | Вывод |
| <code>fixed</code> | Устанавливает флаг fixed | Вывод |
| <code>flush</code> | Очищает поток | Вывод |
| <code>hex</code> | Устанавливает флаг hex | Ввод/вывод |
| <code>internal</code> | Устанавливает флаг internal | Вывод |
| <code>left</code> | Устанавливает флаг left | Вывод |
| <code>noboolalpha</code> | Сбрасывает флаг boolalpha | Ввод/вывод |
| <code>noshowbase</code> | Сбрасывает флаг showbase | Вывод |
| <code>noshowpoint</code> | Сбрасывает флаг showpoint | Вывод |
| <code>noshowpos</code> | Сбрасывает флаг showpos | Вывод |
| <code>noskipws</code> | Сбрасывает флаг skipws | Ввод |
| <code>nounitbuf</code> | Сбрасывает флаг unitbuf | Вывод |
| <code>nouppercase</code> | Сбрасывает флаг uppercase | Вывод |
| <code>oct</code> | Устанавливает флаг oct | Ввод/вывод |
| <code>resetios (fmts f)</code> | Сбрасывает флаги, заданные в <i>f</i> | Ввод/вывод |
| <code>right</code> | Устанавливает флаг right | Вывод |

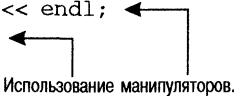
Окончание табл. 11-1

| Манипулятор | Назначение | Ввод или вывод |
|------------------------------|---|----------------|
| scientific | Устанавливает флаг scientific | Вывод |
| setbase(int <i>base</i>) | Устанавливает систему счисления <i>base</i> | Ввод/вывод |
| setfill(int <i>ch</i>) | Устанавливает символ-заполнитель <i>ch</i> | Вывод |
| setiosfmts(<i>f</i>) | Устанавливает флаги, заданные в <i>f</i> | Ввод/вывод |
| setprecision (int <i>p</i>) | Устанавливает число цифр точности | Вывод |
| setw(int <i>w</i>) | Устанавливает ширину поля <i>w</i> | 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 лидирующие пробельные символы | Ввод |

Манипулятор используется как элемент большего выражения ввода-вывода. Вот простая программа, использующая манипуляторы для управления форматом своего вывода:

```
// Демонстрация манипуляторов ввода-вывода.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(2) << 1000.243 << endl;
    cout << setw(20) << "Hello there.";
    return 0;
}
```



Вывод этой программы:

```
1e+003
      Hello there.
```

Обратите внимание, как манипуляторы включаются в цепочку операций ввода-вывода. Также заметьте, что когда манипулятор не принимает аргументов (например, **endl** в программе), за ним не указываются круглые скобки.

Программа, приведенная ниже, использует функцию **setiosflags ()** для установки флагов **scientific** и **showpos**:

```
// Использование setiosflags().

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

int main()
{
    cout << setiosflags(ios::showpos) << ← Использование setiosflags( )
         setiosflags(ios::scientific) <<
         123 << " " << 123.23;

    return 0;
}
```

Следующая программа показывает использование **ws** для удаления лидирующих пробельных символов при вводе строки в **s**:

```
// Удаление лидирующих пробельных символов.

#include <iostream>
using namespace std;

int main()
{
    char s[80];

    cin >> ws >> s; Использование ws
    cout << s;

    return 0;
}
```

Цель

11.6. Создание собственных манипуляторных функций

Вы можете создать свои собственные манипуляторные функции. Эти функции имеют две разновидности: с параметрами и без них. Создание параметризованных манипуляторов требует знакомства с вопросами, которые выходят за рамки этой книги. С другой стороны, создание манипуляторов без параметров осуществляется весьма просто и мы его здесь опишем.

Все манипуляторные функции вывода без параметров имеют такой шаблон:

```
ostream &manip_name(ostream &stream)
{
    // ваш код размещается здесь
    return stream;
}
```

Здесь **manip_name** — это имя манипулятора. Важно понимать, что хотя манипулятор имеет в качестве единственного параметра указатель на поток, которым он манипулирует, при использовании манипулятора в выражении вывода никакие аргументы не указываются.

В приведенной ниже программе создается манипулятор с именем **setup()**, который устанавливает выравнивание влево, ширину поля в 10 символов и знак доллара в качестве символа-заполнителя:

```
// Создание манипулятора вывода
#include <iostream>
#include <iomanip>
using namespace std;

ostream &setup(ostream &stream) ← Собственный манипулятор вывода
{
    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, вам может понадобиться выполнить операцию ввода-вывода на устройстве, с которым не работает ни один из предопределенных манипуляторов — например, плоттере. В этом случае создание собственных манипуляторов облегчит вывод на это устройство. Во-вторых, бывает, что вы повторяете много раз одну и ту же последовательность операций. Вы можете собрать эти операции в один манипулятор, как это иллюстрирует следующая программа.

Все манипуляторы ввода без параметров имеют такой шаблон:

```
istream &manip_name(istream &stream)
{
    // ваш код размещается здесь

    return stream;
}
```

В качестве примера приведенная ниже программа создает манипулятор **prompt()**. Он выводит сообщение-запрос и затем конфигурирует ввод так, чтобы принимались шестнадцатеричные цифры:

```
// Создание манипулятора ввода.

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

istream &prompt(istream &stream) ← Прикладной манипулятор ввода.
{
    cin >> hex;
    cout << "Введите число в шестнадцатеричном формате: ";

    return stream;
}

int main()
{
    int i;

    cin >> prompt >> i;
    cout << i;

    return 0;
}
```

Не забывайте, что существенным условием для манипуляторов является возврат локальной переменной, названной в приведенном примере *stream*. Если этого не сделать, созданный вами манипулятор нельзя будет использовать в цепочечных операциях ввода или вывода.

Минутная тренировка

1. Для чего служит **endl**?
2. Для чего служит **ws**?
3. Используется ли манипулятор ввода-вывода как часть больших выражений ввода-вывода?

- 1 **endl** выводит символ новой строки
- 2 **ws** приводит к пропуску при вводе лидирующих пробельных символов.
- 3 манипулятор ввода-вывода используется как часть больших выражений ввода-вывода

Файловый ввод-вывод

Вы можете использовать систему ввода-вывода C++ для выполнения операций ввода-вывода над файлами. Для этого в программу следует включить заголовок **<fstream>**. В нем определяется несколько важных классов и значений.

Цель

11.7. Открытие и закрытие файла

В C++ файл открывается путем связывания его с потоком. Как вы уже знаете, существуют три типа потоков: ввода, вывода и ввода-вывода. Для того, чтобы открыть поток ввода, вы должны объявить поток класса **ifstream**. Для того, чтобы открыть поток вывода, вы должны объявить поток класса **ofstream**. Поток, который будет выполнять обе операции, и ввода, и вывода, должен быть объявлен как объект класса **fstream**. Например, этот фрагмент создает один поток ввода, один поток вывода и один поток, способный делать и то, и другое:

```
ifstream in; // ввод
ofstream out; // вывод
fstream both; // ввод и вывод
```

Создав поток, вы можете связать его с файлом с помощью функции **open()**. Эта функция входит как член в каждый из трех потоковых классов. Прототипы трех вариантов этой функции выглядят так:

```
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 *filename* обозначает имя файла; в него может входить описание пути. Значение *mode* задает режим работы открываемого файла. Этот аргумент должен иметь значение, определенное типом **openmode**, который представляет собой перечислимый тип, определенный в **ios**

(посредством его базового класса `ios_base`). Возможные значения *mode* приведены ниже:

```
ios::app  
ios::ate  
ios::binary  
ios::in  
ios::out  
ios::trunc
```

Вы можете комбинировать два или несколько значений с помощью побитового ИЛИ (`|`).

Указание **`ios::app`** приводит к тому, что весь вывод записывается в конец файла. Это значение можно использовать только с файлами, допускающими запись в них. Если в объявлении присутствует **`ios::ate`**, то при открытии файла осуществляется поиск его конца. Хотя наличие **`ios::ate`** активизирует начальный поиск конца файла, тем не менее операции ввода-вывода могут осуществляться, начиная с любого места файла.

Значение **`ios::in`** указывает, что файл открывается для ввода. Значение **`ios::out`** указывает, что файл открывается для вывода.

Значение **`ios::binary`** открывает файл в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. В этом режиме могут иметь место различные преобразования символов, например, последовательности возврат каретки — перевод строки преобразовываться в символы новых строк. Однако когда файл открывается в двоичном режиме, никаких преобразований не производится. Следует понимать, что любой файл, содержит ли он форматированный текст или необработанные данные, может быть открыт и в двоичном, и в текстовом режиме. Единственным различием будет наличие или отсутствие преобразования символов.

Значение **`ios::trunc`** приводит к уничтожению исходного содержимого существующего файла с указанным именем, и к усечению файла до нулевой длины. Такая операция усечения автоматически выполняется над любым существующим файлом с указанным именем при открытии потока вывода класса **`ofstream`**.

Следующий фрагмент открывает текстовый файл для вывода:

```
ofstream mystream;  
mystream.open("test");
```

Поскольку параметр *mode* функции **`open()`** по умолчанию принимает значение, соответствующее типу открываемого потока, часто нет необходимости задавать его явным образом, как это и сделано в последнем примере. (Некоторые компиляторы не задают для **`fstream::open()`** по умолчанию значение параметра *mode* in

| **out**; в этом случае вам придется задать этот параметр в явной форме.)

Если функция **open()** завершилась неуспешно, поток, если его использовать в булевом выражении, примет значение **false**. Вы можете использовать это обстоятельство для подтверждения успешного открытия файла с помощью предложения такого рода:

```
if(!mystream) {  
    cout << "Не могу открыть файл.\n";  
    // обработка ошибки  
}
```

Как правило, перед тем, как пытаться обратиться к файлу, вы должны проверить результат выполнения функции **open()**.

Вы можете также проверить, открылся ли файл, с помощью функции **is_open()**, которая является членом классов **fstream**, **ifstream** и **ofstream**. Вот ее прототип:

```
bool is_open( ),
```

Функция возвращает **true**, если поток связан с открытым файлом, и **false** в обратном случае. Следующий фрагмент проверяет, открыт ли в настоящий момент **mystream**:

```
if(!mystream.is_open()) {  
    cout << "Файл не открыт.\n";  
    // ...  
}
```

Хотя использование функции **open()** для открытия файла — вполне допустимая операция, вы, скорее всего, не будете прибегать к ней, поскольку в классах **ifstream**, **ofstream** и **fstream** имеются конструкторы, которые автоматически открывают файл. Конструкторы имеют те же самые параметры и аргументы по умолчанию, что и функция **open()**. Поэтому самый распространенный способ открытия файла выглядит так:

```
ifstream mystream("myfile"); // открыть файл для ввода
```

Если по какой-то причине файл не открылся, значение связанной с файлом переменной потока будет иметь значение **false**.

Для закрытия файла используйте функцию-член **close()**. Например, чтобы закрыть файл, связанный с потоком, который назван **mystream**, вы используете такое предложение:

```
mystream.close();
```

Функция **close()** не принимает параметров и ничего не возвращает.

Цель

11.8. Чтение и запись текстовых файлов

Простейшим способом чтения из текстового файла или записи в него является использование операторов << и >>. Например, приведенная ниже программа записывает целочисленную переменную, число с плавающей точкой и строку в файл с именем **test**:

```
// Запись в файл.

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

int main()
{
    ofstream out("test"); ← Создаем и открываем файл с именем "test" для текстового вывода
    if(!out) {
        cout << "Не могу открыть файл.\n";
        return 1;
    }

    out << 10 << " " << 123.23 << "\n"; ← Вывод в файл
    out << "Это короткий текстовый файл.";

    out.close(); ← Закрытие файла

    return 0;
}
```

Приведенная ниже программа читает целое число, число типа **float**, символ и строку из файла, созданного в предыдущей программе:

```
// Прочитаем из файла.

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

int main()
{
    char ch;
    int i;
```

```

float f;
char str[80];

ifstream in("test"); ← Открываем файл для текстового ввода.
if(!in) {
    cout << "Не могу открыть файл.\n";
    return 1;
}

in >> i;
in >> f;
in >> ch;
in >> str; ← Читаем из файла

cout << i << " " << f << " " << ch << "\n";
cout << str;

in.close(); ← Закрываем файл

return 0;
}

```

Не забывайте, что при использовании оператора `>>` для чтения текстовых файлов осуществляется определенное преобразование символов. Например, убираются пробельные символы. Если вы хотите предотвратить какое бы то ни было преобразование символов, вы должны открывать файл в режиме двоичного доступа. Не забывайте также, что при чтении строки с помощью оператора `>>` ввод останавливается, когда встречается первый пробельный символ.

Минутная тренировка

1. Какой класс создает файл для ввода?
 2. Какая функция открывает файл?
 3. Можете ли вы читать и записывать файл с помощью `<<` и `>>`?
 - 1 Чтобы открыть файл для ввода, следует использовать **ifstream**
 - 2 Чтобы открыть файл, используйте конструктор класса или функцию **open()**
 - 3 Да, вы можете использовать `<<` и `>>` для чтения и записи файла
-

Спросим у эксперта

Вопрос: Как вы уже объясняли в Модуле 1, C++ является надмножеством C. Я знаю, что C определяет собственную систему ввода-вывода. Доступна ли для программистов на C++ система ввода-вывода C? Если да, то следует ли использовать ее в C++-программах?

Ответ: Ответ на первый вопрос: да. Система ввода-вывода языка C доступна программистам на C++. Ответ на второй вопрос: определенно нет. Система ввода-вывода C не является объектно-ориентированной. По этой причине почти всегда оказывается, что система ввода-вывода C++ более совместима с C++-программами. Однако, система ввода-вывода C все еще широко используется; она весьма проста и приводит к незначительным издержкам в программах. Поэтому для узко специализированных программ система ввода-вывода C может оказаться полезной. Сведения о системе ввода-вывода C можно найти в моей книге: *C++: The Complete Reference* (Osborne/McGraw-Hill).

Цель

11.9. Неформатированный и двоичный ввод-вывод

Хотя читать и записывать форматированные текстовые файлы весьма просто, такой способ не всегда оказывается самым эффективным. Кроме того, вам может понадобиться записывать в файл не текст, а неформатированные (необработанные) данные. Здесь описываются функции, позволяющие вам выполнять такого рода операции.

Если вы собираетесь выполнять над файлом двоичные операции, удостоверьтесь, что вы открыли его с помощью описателя режима **ios::binary**. Хотя функции обслуживания неформатированного файла будут работать с файлом, открытым в текстовом режиме, однако при этом будут происходить преобразования некоторых символов. Преобразование символов противоречит самой идее двоичных файловых операций.

В принципе имеются два способа записывать и читать неформатированные двоичные данные в файл или из файла. Во-первых, вы можете записать байт с помощью функции-члена **put ()** и прочитать байт с помощью функции-члена **get ()**. Во-вторых, вы можете использовать блочные функции ввода-вывода: **read ()** и **write ()**. Ниже будут рассмотрены оба способа.

Использование **get ()** и **put ()**

Функция **get ()** имеет много форм, но чаще всего используется вариант, приведенный (вместе с аналогичным вариантом **put ()**) ниже:

```
istream &get(char &ch);  
ostream &put(char ch);
```

Функция `get()` читает один символ из связанного с файлом потока и помещает его значение в `ch`. Функция возвращает ссылку на поток. Это значение может быть нулем, если достигнут конец файла. Функция `put()` записывает `ch` в поток и возвращает ссылку на поток.

Приведенная ниже программа (предполагается, что вы дали ей имя PR) выведет на экран содержимое любого файла. Она использует функцию `get()`:

// Чтение из файла с помощью `get()` и отображение на экране.

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

```
int main(int argc, char *argv[])
{
    char ch;
```

```
if(argc!=2) {
    cout << "Использование: PR <имя-файла>\n";
    return 1;
}
```

Открываем файл для
двоичных операций

```
ifstream in(argv[1], ios::in | ios::binary);
if(!in) {
    cout << "Не могу открыть файл.\n";
    return 1;
}
```

Читаем данные из файла, пока
не будет достигнут конец файла

```
while(in) { // in будет false, когда достигнут eof
    in.get(ch);
    if(in) cout << ch;
}
```

```
in.close();
```

```
return 0;
}
```

Когда поток `in` достигнет конца файла, его значение станет `false`, и цикл `while` завершится.

В действительности имеется более компактный способ описать цикл чтения и отображения файла:

```
while(in.get(ch))
    cout << ch;
```

Это вполне допустимая форма, так как `get()` возвращает поток `in`, а `in` станет равным `false`, когда будет достигнут конец файла.

Приведенная ниже программа использует `put()` для записи строки в файл:

```
// Использование put() для записи в файл.
```

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

```
int main()
{
    char *p = "Привет всем!\n";

    ofstream out("test", ios::out | ios::binary);
    if(!out) {
        cout << "Не могу открыть файл.\n";
        return 1;
    }
```

```
    // Запись символов, пока не будет достигнут завершающий ноль.
    while(*p) out.put(*p++);
```

```
    out.close();
```

```
    return 0;
}
```

Запись строки в файл с помощью `put()` Не выполняются никакие преобразования символов

После выполнения этой программы файл `test` будет содержать строку “Привет всем!”, за которой будет записан символ новой строки. Никакие преобразования символов не осуществляются.

Чтение и запись блоков данных

Для чтения и записи блоков двоичных данных используйте функции-члены `read()` и `write()`. Они имеют такие прототипы:

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

Функция `read()` читает `num` байтов из потока, связанного с файлом, и заносит их в буфер, на который указывает `buf`. Функция `write()` записывает `num` байтов в поток из буфера, на который указывает `buf`. Как уже упоминалось, `streamsize` — это имя типа той или

инной формы целого числа, определенный в библиотеке C++. Переменные этого типа могут содержать значение, соответствующее максимальному числу байтов, которое может быть передано в одной операции ввода-вывода.

Приведенная ниже программа записывает, а затем читает массив целых чисел:

```
// Использование 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 могу открыть файл.\n";
        return 1;
    }

    out.write((char *) &n, sizeof n); ← Запись блока данных

    out.close();

    for(i=0; i<5; i++) // очистим массив
        n[i] = 0;

    ifstream in("test", ios::in | ios::binary);
    if(!in) {
        cout << "Не могу открыть файл.\n";
        return 1;
    }

    in.read((char *) &n, sizeof n); ← Чтение блока данных

    for(i=0; i<5; i++) // показывает значения, прочитанные из
                        // файла
        cout << n[i] << " ";

    in.close();

    return 0;
}
```

Заметьте, что приведение типа, выполняемое внутри вызовов функций `read()` и `write()`, необходимо, если вы используете буфер, определенный не как символьный массив.

Если конец файла достигается до того, как будет прочитано *num* символов, функция `read()` попросту останавливается, а буфер будет содержать столько символов, сколько удалось прочитать. Узнать, сколько символов было прочитано, можно с помощью другой функции-члена, `gcount()`, имеющий такой прототип:

```
streamsize gcount( );
```

Функция `gcount()` возвращает число символов, прочитанных в последней операции ввода.

Минутная тренировка

1. Какой описатель режима вы должны использовать, чтобы читать или записывать двоичные данные?
 2. Для чего предназначена функция `get()`? Для чего предназначена функция `put()`?
 3. Какая функция читает блок данных?
 - 1 Чтобы читать или записывать двоичные данные, вы должны использовать описатель режима `ios::binary`.
 2. Функция `get()` читает символ. Функция `put()` записывает символ.
 3. Для чтения блока данных используйте `read()`.
-

Цель

11.10. Больше о функциях ввода-вывода

Система ввода-вывода C++ определяет и другие относящиеся к вводу-выводу функции, некоторые из которых могут вам пригодиться. Мы их здесь рассмотрим.

Другие варианты `get()`

Функция `get()` имеет целый ряд перегруженных форм помимо той, которая была рассмотрена выше. Приведем прототипы трех наиболее употребительных вариантов:

```
istream &get(char *buf, streamsize num);  
istream &get(char *buf, streamsize num, char delim);  
int get( );
```

Первый вариант читает символы в массив, на который указывает *buf*, до тех пор, пока не будет введено *num* – 1 символов, или обнаружится символ новой строки, или будет достигнут конец файла. Функция **get()** в таком варианте поместит в массив, на который указывает *buf*, завершающий ноль. Если во входном потоке обнаруживается символ новой строки, он *не* извлекается, а остается в потоке до следующей операции ввода.

Второй вариант читает символы в массив, на который указывает *buf*, до тех пор, пока не будет введено *num* – 1 символов, или обнаружится разделительный символ, заданный параметром *delim*, или встретится конец файла. Функция **get()** в таком варианте поместит в массив, на который указывает *buf*, завершающий ноль. Если во входном потоке обнаруживается разделительный символ *delim*, он *не* извлекается, а остается в потоке до следующей операции ввода.

Третья перегруженная форма **get()** возвращает из потока следующий символ. Если достигнут конец файла, функция возвращает **EOF** (символическая константа, обозначающее конец файла). Значение **EOF** определено в `<iostream>`.

Одно из полезных применений функции **get()** – чтение строки, содержащей пробелы. Как вы знаете, если строка читается с помощью `>>`, чтение прекращается на первом пробельном символе. В результате оператор `>>` оказывается бесполезным при чтении строк, содержащих пробелы. Однако вы можете решить эту задачу с помощью варианта функции **get(buf, num)**, как это показано в следующей программе:

```
// Использование get( ) для чтения строки, содержащей пробелы.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[80];

    cout << "Введите свое имя: ";
    cin.get(str, 79); ← Использование get( ) для чтения строки, содержащей пробельные символы.

    cout << str << '\n';

    return 0;
}
```

Здесь используется разделительный символ по умолчанию, который соответствует символу новой строки. В результате **get()** действует практически так же, как и стандартная функция **gets()**.

getline()

Есть еще одна функция, выполняющая ввод — это **getline()**. Она является членом класса потока ввода; ее прототипы выглядят так:

```
istream &getline(char *buf, streamsize num);  
istream &getline(char *buf, streamsize num, char delim);
```

Первый вариант читает символы в массив, на который указывает *buf*, до тех пор, пока не будет введено *num* — 1 символов, или обнаружен символ новой строки, или встретится конец файла. Функция **getline()** в таком варианте помещает в массив, на который указывает *buf*, завершающий ноль. Если во входном потоке обнаруживается символ новой строки, он извлекается, но не помещается в буфер *buf*.

Второй вариант читает символы в массив, на который указывает *buf*, до тех пор, пока не будет введено *num* — 1 символов, или обнаружится разделительный символ, заданный параметром *delim*, или встретится конец файла. Функция **getline()** в таком варианте помещает в массив, на который указывает *buf*, завершающий ноль. Если во входном потоке обнаруживается разделительный символ, он извлекается, но не помещается в буфер *buf*.

Легко видеть, что два варианта **getline()** практически идентичны вариантам **get(buf, num)** и **get(buf, num, delim)** функции **get()**. Обе функции читают символы из потока ввода и помещают их в массив, на который указывает *buf*, до тех пор, пока не будет введено *num* — 1 символов или обнаружится разделительный символ. Разница между **get()** и **getline()** заключается в том, что **getline()** читает и извлекает из потока ввода разделительный символ; **get()** не делает этого.

Обнаружение символа EOF

Для того, чтобы узнать, не достигнут ли конец файла, можно использовать функцию-член **eof()** со следующим прототипом:

```
bool eof( );
```

Функция возвращает **true**, если был достигнут конец файла, и **false** в противном случае.

peek() и putback()

Имеется возможность получить следующий символ в потоке ввода без извлечения его из потока. Для этого предусмотрена функция **peek()**. Она имеет следующий прототип:

```
int peek( );
```

peek() возвращает следующий символ в потоке или **EOF**, если достигнут конец файла. Символ содержится в младшем байте возвращаемого значения.

Можно, наоборот, вернуть в поток последний прочитанный из него символ. Для этого используется функция **putback()**. Ее прототип выглядит так:

```
istream &putback(char c);
```

Здесь *c* — последний прочитанный символ.

```
flush( )
```

Когда выполняется вывод, данные не записываются немедленно на физическое устройство, связанное с потоком. Вместо этого данные запоминаются во внутреннем буфере до его заполнения. Только тогда содержимое буфера переписывается на диск. Однако вы можете принудительно сбросить содержимое буфера на диск, даже если он еще не заполнен, вызвав функцию **flush()**. Ее прототип выглядит так:

```
ostream &flush( );
```

Вызов **flush()** может быть оправдан, если программа будет эксплуатироваться в неблагоприятных условиях (например, при частых нарушениях питания).



Замечание

.....

Закрытие файла или завершение программы также приводит к сбросу всех буферов на диск.

Проект 11-1 Утилита сравнения файлов

В этом проекте разрабатывается простая, но весьма полезная утилита сравнения. Она открывает оба сравниваемых файла, а затем читает и сравнивает соответствующие наборы байтов. Если наблюдается расхождение, значит, файлы различаются. Если достигнут

конец каждого файла, а расхождений не было обнаружено, значит, файлы совпадают.

Шаг за шагом

1. Создайте файл с именем **CompFiles.cpp**.
2. Начните с добавления в **CompFiles.cpp** этих строк:

```
/*
    Проект 11-1

    Разработка утилиты сравнения файлов.
*/

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

int main(int argc, char *argv[])
{
    register int i;
    int numread;

    unsigned char buf1[1024], buf2[1024];

    if(argc!=3) {
        cout << "Использование: compfiles <файл1> <файл2>\n";
        return 1;
    }
```

Обратите внимание на то, что имена сравниваемых файлов вводятся в качестве параметров командной строки.

3. Добавьте код, который открывает файлы для двоичных операций ввода, как это показано ниже:

```
ifstream f1(argv[1], ios::in | ios::binary);
if(!f1) {
    cout << "Не могу открыть первый файл.\n";
    return 1;
}

ifstream f2(argv[2], ios::in | ios::binary);
if(!f2) {
    cout << "Не могу открыть второй файл.\n";
    return 1;
}
```

Файлы открываются для двоичных операций, чтобы предотвратить возможные преобразования символов.

4. Добавьте код, который фактически сравнивает файлы:

```
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 разную длину.\n";
        f1.close();
        f2.close();
        return 0;
    }

    // сравнение содержимого буферов
    for(i=0; i<f1.gcount(); i++)
        if(buf1[i] != buf2[i]) {
            cout << "Файлы различаются.\n";
            f1.close();
            f2.close();
            return 0;
        }

} while(!f1.eof() && !f2.eof());

cout << "Файлы совпадают.\n";
```

В этом фрагменте с помощью функции `read()` из каждого файла последовательно читаются порции данных размером в выделенные для них буферы. Затем выполняется сравнение содержимого буферов. Если содержимое буферов различается, файлы закрываются, на экран выводится сообщение "Файлы различаются." и программа завершается. В противном случае операции чтения в буферы и сравнения их содержимого повторяются, пока не будет достигнут конец одного (или обоих) файлов. Поскольку в конце файлов последние порции читаемых данных могут не заполнить буферы целиком, программа использует функцию `gcount()` для определения точного числа символов, прочитанных в буферы. Если один из файлов короче другого, то при достижении конца более короткого файла значения, возвращаемые `gcount()`, будут различаться. В этом случае выводится сообщение "Файлы имеют разную длину.". Наконец, если файлы совпадают, то при достижении конца одного файла будет достигнут конец и другого, что

подтверждается вызовом функции `eof()` для каждого потока. Если файлы оказываются полностью совпадающими, выводится соответствующее сообщение.

5. Завершите программу закрытием обоих файлов:

```
f1.close();
f2.close();

return 0;
}
```

6. Ниже приведен полный текст программы **CompFiles.cpp**:

```
/*
    Проект 11-1

    Разработка утилиты сравнения файлов.
*/

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

int main(int argc, char *argv[])
{
    register int i;
    int numread;

    unsigned char buf1[1024], buf2[1024];

    if(argc!=3) {
        cout << "Использование: compfiles <файл1> <файл2>\n";
        return 1;
    }

    ifstream f1(argv[1], ios::in | ios::binary);
    if(!f1) {
        cout << "Не могу открыть первый файл.\n";
        return 1;
    }
    ifstream f2(argv[2], ios::in | ios::binary);
    if(!f2) {
        cout << "Не могу 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 разную длину.\n";
        f1.close();
        f2.close();
        return 0;
    }

    // сравнение содержимого буферов
    for(i=0; i<f1.gcount(); i++)
        if(buf1[i] != buf2[i]) {
            cout << "Файлы различаются.\n";
            f1.close();
            f2.close();
            return 0;
        }

} while(!f1.eof() && !f2.eof());

cout << "Файлы совпадают.\n";

f1.close();
f2.close();

return 0;
}
```

7. Чтобы испытать программу, сначала скопируйте файл **CompFiles.cpp** в файл под именем **temp.txt**. Затем введите такую командную строку:

```
CompFiles CompFiles.cpp temp.txt
```

Программа должна сообщить вам, что файлы совпадают. Далее, сравните **CompFiles.cpp** с каким-нибудь другим файлом, например, с другой программой из этого модуля. Вы увидите, что **CompFiles.cpp** сообщает о несовпадении файлов.

8. Попробуйте самостоятельно усложнить **CompFiles.cpp**. Например, добавьте режим, в котором программа перестанет различать строчные и прописные буквы. Другое усовершенствование может заключаться в том, чтобы **CompFiles.cpp** выводила на экран позицию внутри файла, где найдено расхождение.

Цель

11.11. Произвольный доступ

До сих пор чтение и запись файлов осуществлялись в последовательном режиме. Однако вы можете обращаться к произвольным местам файла в случайном порядке. В системе ввода-вывода C++ произвольный доступ к файлу осуществляется с помощью функций `seekg()` и `seekp()`. Их наиболее употребительные формы выглядят таким образом:

```
istream &seekg(off_type offset, seekdir origin);  
ostream &seekp(off_type offset, seekdir origin);
```

Здесь `off_type` — это целочисленный тип, определенный в `ios`. Переменные этого типа имеют длину, достаточную для размещения максимального значения, которое может иметь `offset`. `seekdir` является перечислимым типом, который может принимать следующие значения:

| Значение | Описание |
|-----------------------|-----------------|
| <code>ios::beg</code> | Начало файла |
| <code>ios::cur</code> | Текущая позиция |
| <code>ios::end</code> | Конец файла |

Система ввода-вывода C++ поддерживает два указателя, связанных с файлом. Один является указателем ввода; он указывает на текущую позицию ввода, т. е. в каком месте файла будет выполняться очередная операция ввода. Другой является указателем вывода; он указывает на текущую позицию вывода, т. е. в каком месте файла будет выполняться очередная операция вывода. Каждый раз, когда выполняется операция ввода или вывода, соответствующий операции указатель автоматически перемещается. Используя функции `seekg()` и `seekp()`, мы можем перемещать эти указатели и тем самым обеспечивать обращение к произвольным местам файла.

Функция `seekg()` перемещает указатель ввода связанного с потоком файла на `offset` байтов от места, указанного параметром `origin`. Функция `seekp()` перемещает указатель вывода связанного с потоком файла на `offset` байтов от места, указанного параметром `origin`.

В принципе операции ввода-вывода с произвольным доступом должны выполняться только над файлами, открытыми для двоичных операций. Символьные преобразования, которые могут происходить при работе с текстовыми файлами, могут привести к тому, что запрос на установку определенной позиции в файле не будет соответствовать фактическому содержанию файла.

Приведенная ниже программа демонстрирует использование функции **seekp()**. Программа (предполагается, что вы дали ей имя **CHANGE**) позволяет задать на командной строке имя файла, вслед за которым указывается номер конкретного байта файла, которой мы хотим изменить. Программа записывает символ **X** в указанный байт файла. Заметьте, что файл должен быть открыт для операций чтения/записи.

// Демонстрация произвольного доступа.

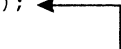
```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Использование: CHANGE <имя-файла> <номер-байта>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out) {
        cout << "Не могу открыть файл.\n";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);
    out.put('X');
    out.close();

    return 0;
}
```



Поиск указанного байта в файле. Эта операция смещает указатель вывода

Следующая программа использует функцию **seekg()**. Она выводит на экран содержимое файла, начиная с того места, которое вы указали на командной строке. Слово **NAME** в строке

```
cout << "Использование: NAME <имя файла> <начальная позиция>\n";
```

следует заменить именем файла с программой.

// Вывод файла начиная с заданного места.

```
#include <iostream>
#include <fstream>
```

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Использование: NAME <имя файла> <начальная
                позиция>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Не могу открыть файл.\n";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg); ← Это предложение пере-
                                         мещает указатель ввода

    while(in.get(ch))
        cout << ch;

    return 0;
}
```

Вы можете определить текущую позицию каждого из указателей в файле с помощью таких функций:

```
pos_type tellg( );
pos_type tellp( );
```

Здесь **pos_type** — это целочисленный тип, определенный в **ios**. Переменные этого типа имеют достаточную длину для размещения максимального значения, которое могут возвращать эти функции.

Имеются перегруженные варианты **seekg()** и **seekp()**, которые перемещают указатели в файле в позиции, полученные с помощью предварительного вызова функций **tellg()** и **tellp()**. Их прототипы:

```
istream &seekg(pos_type position);
ostream &seekp(pos_type position);
```

Минутная тренировка

1. Какая функция позволяет определить достижение конца файла?

2. Для чего предназначена функция **getline()**?
3. Какие функции позволяют устанавливать позиции для произвольного доступа?

- 1 Функция **eof()** позволяет определить достижение конца файла
- 2 **getline()** читает строку текста
- 3 Для установки позиции для произвольного доступа используйте функции **seekg()** и **seekp()**

Цель

11.12. Определение состояния ввода-вывода

Система ввода-вывода C++ поддерживает информацию о результатах каждой операции ввода-вывода. Биты состояния потока ввода-вывода описаны в объекте типа **iostate**, который представляет собой перечислимый тип, определенный в **ios** и включающий следующие члены:

| Имя | Описание |
|---------------------|--|
| ios::goodbit | Не установлены никакие биты ошибок |
| ios::eofbit | 1 если достигнут конец файла, 0 в противном случае |
| ios::failbit | 1 если (возможно) возникла нефатальная ошибка ввода-вывода; 0 в противном случае |
| ios::badbit | 1 если возникла фатальная ошибка ввода-вывода, 0 в противном случае |

Вы можете получить информацию о состоянии ввода-вывода двумя способами. Первый способ — вызвать функцию **rdstate()**. Ее прототип

```
iostate rdstate( );
```

Функция возвращает текущее состояние флагов ошибок. Как вы можете догадаться, взглянув на приведенный выше перечень флагов, если ошибок нет, **rdstate()** возвращает **goodbit**. В противном случае устанавливается флаг ошибки.

Другой способ обнаружения ошибки ввода-вывода заключается в использовании одной или нескольких из следующих функций-членов **ios**:

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

Функция `eof()` уже обсуждалась ранее. Функция `bad()` возвращает `true`, если установлен бит `badbit`. Функция `fail()` возвращает `true`, если установлен бит `failbit`. Функция `good()` возвращает `true`, если ошибок не было. В противных случаях все эти функции возвращают `false`.

Если произошла ошибка, то вам может понадобиться перед продолжением программы сбросить ее флаг. Для этого можно воспользоваться функцией-членом `ios::clear()`, имеющей такой прототип:

```
void clear(iostate flags = ios::goodbit);
```

Если `flags = ios::goodbit`, как это имеет место по умолчанию, все флаги ошибок сбрасываются. В противном случае в переменной `flags` следует указать интересующие вас флаги.

Перед тем, как вы продолжите изучение этой книги, можно порекомендовать вам поэкспериментировать с использованием этих функций обнаружения ошибок и добавить в предшествующие примеры более детальную обработку возможных ошибок.

✓ Вопросы для самопроверки

1. Как называются четыре предопределенных потока?
2. Определяет ли C++ символьные потоки и для 8-битовых, и для “широких” символов?
3. Приведите общую форму перегруженного оператора вставки.
4. Каково назначение `ios::scientific`?
5. Каково назначение `width()`?
6. Манипулятор ввода-вывода используется внутри выражений ввода-вывода. Справедливо ли это утверждение?
7. Покажите, как открыть файл для ввода текста.
8. Покажите, как открыть файл для вывода текста.
9. Каково назначение `ios::binary`?
10. Когда достигается конец файла, переменная потока становится равной `false`. Справедливо ли это утверждение?
11. Покажите, как прочитать файл до конца, если он связан с потоком ввода по имени `strm`.
12. Напишите программу, которая копирует файлы. Пользователь должен задавать имена входного и выходного файлов в командной строке. Удостоверьтесь, что ваша программа копирует как текстовые, так и двоичные файлы.
13. Напишите программу, которая объединяет два текстовых файла. Пользователь должен задавать имена двух файлов в команд-

ной строке в том порядке, в котором они должны войти в выходной файл. Кроме того, пользователь должен задавать имя выходного файла. Таким образом, если программа имеет имя **merge**, то командная строка, которая объединит файлы **MyFile1.txt** и **MyFile2.txt** в файл **Target.txt**, будет выглядеть следующим образом:

merge MyFile1.txt MyFile2.txt Target.txt

14. Покажите, как с помощью предложения **seekg()** найти 330-й байт в потоке с именем **MyStrm**.

Модуль 12

Исключения, шаблоны и другие дополнительные темы

Цели, достигаемые в этом модуле

- 12.1 Изучить основы обработки исключений
- 12.2 Понять, как создаются родовые функции
- 12.3 Понять, как создаются родовые классы
- 12.4 Познакомиться с операторами new и delete динамического выделения памяти
- 12.5 Начать использовать пространства имен
- 12.6 Рассмотреть использование статических членов классов
- 12.7 Научиться идентифицировать типы во время выполнения программы
- 12.8 Освоить дополнительные операторы приведения

С того момента, как вы начали изучать эту книгу, вы много узнали и многому научились. Теперь, в заключительном модуле, вы познакомитесь с некоторыми важными разделами C++, включая обработку исключений, шаблоны, динамическое выделение памяти и пространства имен. Вы узнаете также об идентификаторах времени выполнения и операторах приведения. Имейте в виду, что C++ — это сложный профессиональный язык программирования с большим количеством самых разнообразных средств, и в этом пособии для начинающих нельзя описать все его дополнительные возможности, специальные приемы или нюансы программирования. Однако, закончив изучение этого модуля, вы освоите базовые элементы языка и сможете приступить к разработке реальных программ.

Цель

12.1. Обработка исключений

Исключительной ситуацией, или исключением, называется ошибка, возникающая во время выполнения. С помощью подсистемы обработки исключений C++ вы можете систематическим и управляемым образом обрабатывать ошибки времени выполнения. Если в вашей программе используется обработка исключений, то при возникновении исключения в ней автоматически вызывается программа его обработки. Принципиальным достоинством обработки исключений является возможность автоматического включения в большую программу значительной части программного кода, предназначенного для обработки ошибок, который в противном случае пришлось бы вводить вручную.

Основы обработки исключений

Обработка исключения в C++ основана на трех ключевых словах: **try**, **catch** и **throw**. Коротко говоря, программные предложения, в которых вы хотите отслеживать исключения, заключаются в блок *попытки* **try**. Если внутри блока **try** возникает исключение (т. е. ошибка), оно *выбрасывается* (с помощью **throw**). Исключение *ловится* с помощью **catch** и обрабатывается. Ниже этот процесс будет описан с большими деталями.

Код, в котором вы хотите контролировать исключения, должен входить в состав блока **try**. (Функция, вызываемая из блока **try**, тоже контролируется.) Исключения, выбрасываемые контролируемым кодом, ловятся предложением **catch**, которое следует непосредственно за предложением **try**, выбросившем исключение. Общая форма конструкции **try-catch** выглядит таким образом:

```
try {  
    // блок try  
}  
catch (тип1 аргумент) {  
    // блок catch  
}  
catch (тип2 аргумент) {  
    // блок catch  
}  
catch (тип3 аргумент) {  
    // блок catch  
}  
// ...  
catch (типN аргумент) {  
    // блок catch  
}
```

Блок **try** должен содержать ту часть вашей программы, в которой вы хотите отслеживать ошибки. Эта программная секция может быть очень короткой и содержать всего несколько предложений внутри одной функции, или быть такой всеобъемлющей, как блок **try**, заключающий в себе весь код функции **main()** (что приведет к контролю исключений во всей вашей программе).

Выброшенное исключение ловится соответствующим предложением **catch**, которое обрабатывает это исключение. К каждому блоку **try** может относиться несколько предложений **catch**. Какое из этих предложений **catch** будет использоваться, определяется типом исключения. Если тип данных, указанный в предложении **catch**, соответствует типу исключения, будет выполняться именно это предложение **catch** (все остальные предложения **catch** обходятся). Когда исключение поймано, *аргумент* блока **catch** принимает его значение. Пойманы могут быть любые типы данных, включая классы, которые вы создаете сами.

Общая форма предложения **throw** выглядит так:

throw *исключение*

Предложение **throw** выбрасывает исключение, указанное параметром *исключение*. Если это исключение требуется поймать, тогда **throw** должны быть выполнено либо внутри самого блока **try**, либо внутри любой функции, вызываемый из этого блока **try** (непосредственно или косвенным образом).

Если вы выбрасываете исключение, для которого не предусмотрено соответствующего предложения **catch**, возникает аварийное завершение программы. Это значит, что программа прекратит свое выполнение неконтролируемым образом. Таким образом, необходимо ловить все выбрасываемые исключения.

Ниже приведен простой пример программы, в которой использован механизм обработки исключений C++:

// Простой пример обработки исключений.

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

int main()
{
    cout << "Старт\n";

    try { // начинается блок try ←————— Начало блока try
        cout << "Внутри блока try\n";
        throw 99; // выбросим ошибку ←————— Выбрасывание исключения
        cout << "Это выполняться не будет";
    }
    catch (int i) { // поймаем ошибку ←————— Улавливание исключения
        cout << "Поймано исключение - значение равно: ";
        cout << i << "\n";
    }

    cout << "Конец";

    return 0;
}
```

Эта программа выведет следующее:

```
Старт
Внутри блока try
Поймано исключение - значение равно: 99
Конец
```

Рассмотрим текст этой программы. Как вы видите, в ней есть блок **try**, включающий в себя три предложения, а также предложение **catch(int i)**, которое обрабатывает исключение типа **int**. Внутри блока **try** только два из трех предложений будут выполняться: первое предложение **cout** и **throw**. После того, как исключение будет выброшено, управление передается на выражение **catch**, а блок **try** завершается. Другими словами, **catch** не вызывается. Вместо этого программное управление *передается* на предложение **catch**. (Для выполнения этого действия программный стек при необходимости автоматически устанавливается в исходное состояние.) Таким образом, предложение **cout**, следующее за **throw**, никогда выполняться не будет.

Обычно код внутри предложения **catch** пытается исправить ошибку с помощью соответствующих действий. Если ошибка может

быть ликвидирована, тогда выполнение программы продолжится с тех предложений, которые стоят вслед за предложением **catch**. В противном случае выполнение программы будет завершено контролируемым образом.

Как уже упоминалось выше, тип исключения должен соответствовать типу, указанному в предложении **catch**. Если, например, в предыдущей программе вы измените тип в предложении **catch** на **double**, тогда исключение не будет поймано, и произойдет аварийное завершение программы. Такой пример приведен ниже:

```
// Эта программа работать не будет.

#include <iostream>
using namespace std;

int main()
{
    cout << "Старт\n";
    try { // начало блока try
        cout << "Внутри блока try\n";
        throw 99; // Выбросим ошибку
        cout << " Это выполняться не будет";
    }
    catch (double i) { // не будет работать для исключения int
        cout << "Поймано исключение – значение равно: ";
        cout << i << "\n";
    }

    cout << "Конец";

    return 0;
}
```

↑
Это предложение не может
поймать исключение **int**!

Вывод этой программы приведен ниже. Его содержимое говорит о том, что целочисленное исключение не может быть поймано предложением **catch(double i)**. Учтите, что заключительное сообщение, говорящее об аварийном завершении программы, может изменяться от компилятора к компилятору.

```
Старт
Внутри блока try
Abnormal program termination
```

Исключение, выброшенное функцией, которая вызывается изнутри блока **try**, может быть поймано этим блоком **try**. Например, приведенная ниже программа вполне правильна:

```

/* Выбрасывание исключения из функции,
   вызванной из блока try. */

#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "Внутри Xtest, пробное значение равно: " << test <<
    "\n";
    if(test) throw test; ← Это исключение улавливается пред-
                           ложением catch в main( ).
}

int main()
{
    cout << "Старт\n";

    try { // начинается блок try
        cout << "Внутри блока try\n";
        Xtest(0);
        Xtest(1); ← Поскольку Xtest( ) вызвана изнутри блока try,
                   ее код также контролируется на ошибку.
        Xtest(2);
    }
    catch (int i) { // поймано ошибку
        cout << "Поймано исключение - значение равно: ";
        cout << i << "\n";
    }

    cout << "Конец";

    return 0;
}

```

Программа выводит на экран следующее:

```

Старт
Внутри блока try
Внутри Xtest, пробное значение равно: 0
Внутри Xtest, пробное значение равно: 1
Поймано исключение - значение равно: 1
Конец

```

Как подтверждается выводом программы, исключение, выброшенное в `Xtest()`, было поймано обработчиком исключений в `main()`.

Блок `try` может быть локализован в функции. В этом случае при каждом входе в функцию обработчик исключений, связанный с этой функцией, приводится в исходное состояние. Посмотрим на следующую программу.

```
// Блок try может быть локализован в функции.

#include <iostream>
using namespace std;

// try/catch переустанавливается при каждом входе в функцию.
void Xhandler(int test)
{
    try{ ←—————Этот блок try локализован в Xhandler.
        if(test) throw test;
    }
    catch(int i) {
        cout << "Поймали! Исключение #: " << i << '\n';
    }
}

int main()
{
    cout << "Старт\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "Конец";

    return 0;
}
```

Программа выдает на экран следующее:

```
Старт
Поймали! Исключение #: 1
Поймали! Исключение #: 2
Поймали! Исключение #: 3
Конец
```

В этом примере выбрасываются три исключения. После каждого исключения осуществляется возврат из функции. Когда функция вызывается в следующий раз, обработчик исключений приводится в исходное состояние.

В принципе блок **try** переустанавливается каждый раз, когда происходит вход в него. Таким образом, если блок **try** является частью цикла, он будет переустанавливаться в каждом новом шаге цикла.

Минутная тренировка

1. Что называется исключением в C++?
 2. На каких трех ключевых словах основывается обработки исключений?
 3. Улавливание исключения основывается на его типе. Правильно ли это?
1. Исключением называется ошибка времени выполнения
 2. Обработка исключений основывается на ключевых словах **try**, **catch** и **throw**
 3. Правильно, улавливание исключения основывается на его типе
-

Использование группы предложений catch

Как уже отмечалось, с одним предложением **try** можно связать несколько предложений **catch**. В действительности обычно так и делают. Однако эти предложения **catch** должны ловить исключения разных типов. Например, приведенная ниже программа ловит исключения типа **int** и указателя на символы:

```
// Использование группы предложений catch.

#include <iostream>
using namespace std;

// Можно ловить исключения разных типов.
void Xhandler(int test)
{
    try{
        if(test) throw test; // выбрасываем int
        else throw "Значение равно нулю"; // выбрасываем char *
    }
    catch(int i) { ←————— Это предложение ловит исключения int
        cout << "Поймали! Исключение #: " << i << '\n';
    }
    catch(char *str) { ←————— Это предложение ловит исключения char*
        cout << "Поймали строку: ";
        cout << str << '\n';
    }
}

int main()
{
    cout << "Старт\n";

    Xhandler(1);
    Xhandler(2);
}
```

```
Xhandler(0);
Xhandler(3);

cout << "Конец";

return 0;
}
```

Программа выводит следующее:

```
Старт
Поймали! Исключение #: 1
Поймали! Исключение #: 2
Поймали строку: Значение равно нулю
Поймали! Исключение #: 3
Конец
```

Как видите, каждое предложение **catch** откликается только на его собственный тип.

В общем случае выражения **catch** проверяются в том порядке, в каком они включены в программу. Выполняется только предложение того же типа. Все остальные блоки **catch** игнорируются.

Улавливание исключений базового класса

Существует важное правило использования группы предложений **catch**, имеющее отношение к производным классам. Предложение **catch** для базового класса будет также соответствовать любому классу, производному от этого. Поэтому, если вы хотите ловить исключения как типа базового класса, так и типа производного класса, поместите производный класс в начале последовательности **catch**. Если вы поступите наоборот, **catch** базового класса будет ловить также исключения всех производных классов. Рассмотрим в качестве примера следующую программу:

// Улавливание производного класса. Программа неверна!

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

class B {
};

class D: public B {
};
```



```

int main()
{
    D derived;

    try {
        throw derived;
    }
    catch(B b) { ← Этот список catch написан в не-
        cout << "Пойман базовый класс.\n";           правильном порядке! Производ-
    }                                                    ные классы должны ловиться пе-
    catch(D d) { ← ред базовым классом
        cout << "Это не будет выполняться.\n";
    }

    return 0;
}

```

Из-за того, что у объекта **derived** базовым классом является класс **B**, он будет пойман первым предложением **catch**, а второе предложение **catch** никогда выполняться не будет. Некоторые компиляторы отметят это предложение и выведут предупреждающее сообщение. Другие могут вывести сообщение об ошибке и прекратить компиляцию. В любом случае, чтобы программа работала правильно, вы должны изменить порядок предложений **catch**.

Улавливание всех исключений

В некоторых случаях вам может понадобиться обработчик исключений, который будет ловить все исключения, а не исключения одного определенного типа. Для построения такого обработчика вы должны использовать следующую форму **catch**:

```

catch(...) {
    //обработка всех исключений
}

```

Здесь многоточие — символ улавливания всех исключений. Приведенная ниже программа иллюстрирует форму **catch(...)**.

```

// В этом примере ловятся все исключения.

#include <iostream>
using namespace std;

void Xhandler(int test)
{

```

```
try{
    if(test==0) throw test; // выбросить int
    if(test==1) throw 'a'; // выбросить char
    if(test==2) throw 123.23; // выбросить double
}
catch(...) { // ловим все исключения ←———— Ловит все исключения
    cout << "Поймали!\n";
}
}

int main()
{
    cout << "Старт\n";
    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "Конец";

    return 0;
}
```

Программа выводит следующее:

```
Старт
Поймали!
Поймали!
Поймали!
Конец
```

Xhandler выбрасывает исключения трех типов: **int**, **char** и **double**. Все они ловятся с помощью предложения **catch(...)**.

Весьма полезным приемом применения **catch(...)** является его использование в качестве последнего **catch** в группе таких предложений. В этом случае оно предоставляет удобный вариант по умолчанию. Используя **catch(...)** в качестве предложения по умолчанию, вы получаете способ улавливания всех исключений, которые вы не хотите обрабатывать явным образом. Кроме того, улавливая все исключения, вы предотвращаете аварийное завершение программы в случае выбрасывания необрабатываемого исключения.

Задание исключений, выбрасываемых функцией

Вы можете задать тип исключений, которые могут быть выброшены функцией, вне этой функции. Вы можете также запретить функции выбрасывать какие-либо исключения. Для реализации

этих ограничений необходимо добавить предложение **throw** к определению функции. Общая форма такого определения выглядит следующим образом:

```
тип-возврата имя-функции(список-аргументов) throw (список-типов)
{
    //...
}
```

При таком определении функции она может выбрасывать исключения только тех типов, которые содержатся в перечне *список-типов*. Выбрасывание исключения любого другого типа приведет к аварийному завершению программы. Если вы хотите запретить функции выбрасывать какие бы то ни было исключения, укажите вместо списка типов пустой список.



Замечание

К моменту написания этой книги Visual C++ фактически не предохранял функцию от выбрасывания исключений, не перечисленных в предложении **throw**. Однако такое поведение является нестандартным. Вы по-прежнему можете включить в определение функции предложение **throw**, однако такое предложение будет носить чисто информационный характер.

Приведенная ниже программа показывает, как задавать типы исключений, которые могут быть выброшены функцией.

```
// Ограничение типов исключений, выбрасываемых функцией.

#include <iostream>
using namespace std;

// Эта функция может выбрасывать только int, char и double.
void Xhandler(int test) throw(int, char, double)
{
    if(test==0) throw test;    // выбросить int
    if(test==1) throw 'a';    // выбросить char
    if(test==2) throw 123.23; // выбросить double
}

int main()
{
    cout << "Старт\n";
```

↑
Укажите исключения, которые могут выбрасываться функцией **Xhandler**

```
try{
    Xhandler(0); // попробуйте также передать функции
                // Xhandler() 1 и 2
}
catch(int i) {
    cout << "Поймано int\n";
}
catch(char c) {
    cout << "Поймано char\n";
}
catch(double d) {
    cout << "Поймано double\n";
}

cout << "end";

return 0;
}
```

В этой программе функция **Xhandler** может выбрасывать только исключения типов **int**, **char** и **double**. Если она попытается выбросить исключение какого-либо другого типа, произойдет аварийное завершение программы. Чтобы наблюдать такое поведение программы, удалите **int** из списка и запустите программу заново (предварительно перекомпилировав ее). Вы получите предупреждающее сообщение компилятора и ошибку при выполнении программы. (Как уже отмечалось, в настоящее время Visual C++ не ограничивает типы исключений, выбрасываемых функцией.)

Важно отдавать себе отчет в том, что функция может быть ограничена в выбрасывании исключений только в тот блок **try**, который ее вызвал. Блок **try** *внутри* функции может выбрасывать исключения любых типов, при условии, что эти исключения будут пойманы *внутри* этой функции. Ограничения относятся только к выбрасыванию исключений из функции наружу.

Вторичное выбрасывание исключения

Вы можете выбросить исключение изнутри обработчика исключения, вызвав **throw** само по себе, без указания типа исключения. В результате текущее исключение будет передано внешнему предложению **try/catch**. Наиболее вероятным применением такого вызова **throw** является ситуация, когда несколько обработчиков исключений должны иметь доступ к одному исключению. Скажем, один обработчик исключения занимается одним аспектом этого исключения, а второй обработчик — другим аспектом. Исключение может быть выброшено

только изнутри блока **catch** (или из любой функции, вызванной изнутри этого блока). Когда вы вторично выбрасываете исключение, оно не будет поймано тем же предложением **catch**. Оно перейдет к следующему предложению **catch**. Приводимая ниже программа иллюстрирует вторичное выбрасывание исключения. В ней вторично выбрасывается исключение **char***.

// Пример вторичного выбрасывания исключения.

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

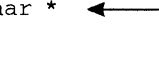
void Xhandler()
{
    try {
        throw "Привет"; // Выбрасываем char *
    }
    catch(char *) { // ловим char *
        cout << "Поймано char * внутри Xhandler\n";
        throw ; // вторичное выбрасывание char * ←
                // из функции наружу
    }
}

int main()
{
    cout << "Старт\n";

    try{
        Xhandler();
    }
    catch(char *) {
        cout << "Поймано char * внутри main\n";
    }

    cout << "Конец";

    return 0;
}
```



Вторичное выбрасывание исключения

Программа выводит следующее:

```
Старт
Поймано char * внутри XHandler
Поймано char * внутри main
Конец
```

Минутная тренировка

1. Покажите, как поймать все исключения.
 2. Как указать тип исключений, которые могут быть выброшены из функции?
 3. Каким образом можно вторично выбросить исключение?
 1. Для того чтобы поймать все исключения, используйте `catch(...)`.
 2. Для указания типов исключений, которые могут быть выброшены из функции, используйте предложение `throw`.
 3. Для вторичного выбрасывания исключения вызовите `throw` без значения.
-

Спросим у эксперта

Вопрос: Получается, что у функции есть два способа сообщить об ошибке: выбросить исключение или вернуть код ошибки. В каких случаях лучше пользоваться тем или другим подходом?

Ответ: Вы правы, имеются два общих подхода к сообщению об ошибках: выбрасывание исключений и возврат кода ошибки. На сегодня специалисты по языку предпочитают использовать исключения, а не коды ошибок. Например, языки Java и C# плотно привязаны к исключениям, которые они используют для сообщений о большинстве распространенных ошибок, таких, как ошибка открытия файла или арифметическое переполнение. Поскольку C++ происходит от C, он использует для сообщений об ошибках и коды ошибок, и исключения. Так, многие ошибочные ситуации, возникающие при использовании библиотечных функций C++, приводят к возврату в программу кода ошибки. Однако, в новых программах, которые вы будете создавать, вам следует использовать для сообщений об ошибках механизм исключений. Это путь, по которому идет современное программирование.

Шаблоны

Шаблон является одним из наиболее сложных и мощных средств C++. Шаблонов не было в исходных спецификациях C++; они были добавлены лишь несколько лет назад и поддерживаются всеми современными компиляторами C++. Шаблоны помогают вам достичь одной из самых трудноуловимых целей в программировании: созданию повторно используемого кода.

Использование шаблонов позволяет создавать родовые (типовые, обобщенные) функции и классы. В родовой функции или классе тип данных, с которыми работает функция или класс, задается с помощью параметра. Таким образом, вы можете использовать одну функцию или класс для обработки данных различных типов вместо того, чтобы создавать специфические варианты кода для каждого типа данных. Здесь мы коснемся и родовых функций, и родовых классов.

Цель

12.2. Родовые функции

Родовая функция определяет обобщенный набор операций, которые будут применены к различным типам данных. Тип данных, с которыми будет работать функция, передается ей через параметр. Посредством родовой функции единую обобщенную процедуру можно приложить к широкому диапазону данных. Как вы, возможно, знаете, многие алгоритмы логически остаются неизменными, независимо от того, данные какого типа они обслуживают. Например, алгоритм упорядочения Quicksort остается одним и тем же, будет ли он применен к массиву целых чисел или к массиву чисел с плавающей точкой. Различаются в этом случае лишь типы обрабатываемых данных. Создавая родовую функцию, вы определяете существо алгоритма независимо от каких-либо данных. После того, как вы это сделаете, компилятор будет автоматически генерировать правильный код для того типа данных, который фактически используется при выполнении этой функции. По существу, создавая родовую функцию, вы создаете функцию, которая может автоматически перегружать саму себя.

Родовая функция создается с помощью ключевого слова **template**. Обычное значение этого слова (шаблон в переводе на русский язык) точно отражает его использование в C++. С помощью **template** создается шаблон или каркас, который описывает, что должна делать эта функция, оставляя для компилятора задачу заполнение этого каркаса требуемыми деталями. Общая форма определения родовой функции выглядит так:

```
template <class тип> тип-возврата имя-функции(список-параметров)
{
    //тело функции
}
```

Здесь *тип* — это условное имя обобщенного, родового типа обрабатываемых данных. Это имя затем используется внутри определения функции для объявления типа данных, с которыми будет работать функция. Компилятор, создавая конкретный вариант функции, автоматически заменит *тип* на тип фактически используемых данных. Хотя использование ключевого слова **class** для задания родового типа в объявлении шаблона стало традиционным, вы можете также использовать ключевое слово **typename**.

В приведенной ниже программе создается родовая функция, которая обменивает значения двух переменных, для которых она вызывается. Поскольку алгоритм обмена двух значений независим от типа переменных, такая процедура удачно подходит в качестве кандидата на роль родовой функции.

```
// пример родовой функции.
```

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

Родовая функция, которая обменивает значения своих аргументов. Здесь **X** — это родовой тип данных.

```
// Это родовая функция.
```

```
template <class X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

```
int main()
```

```
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';
```

```
    cout << "Исходные i, j: " << i << ' ' << j << '\n';
    cout << "Исходные x, y: " << x << ' ' << y << '\n';
    cout << "Исходные a, b: " << a << ' ' << b << '\n';
```

```
    swapargs(i, j); // обменяем целые
    swapargs(x, y); // обменяем значения
                    // с плавающей точкой
    swapargs(a, b); // обменяем символы
```

Компилятор автоматически создает варианты функции **swapargs()**, которые используют типы данных своих аргументов.

```
    cout << "После обмена i, j: " << i << ' ' << j << '\n';
    cout << "После обмена x, y: " << x << ' ' << y << '\n';
    cout << "После обмена a, b: " << a << ' ' << b << '\n';
```

```
    return 0;
```

```
}
```

Рассмотрим детально эту программу. Строка

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору, что, во-первых, создается шаблон и, во-вторых, что начинается родовое объявление. В этой строке **X** — это условное имя родового типа. Далее объявляется функция **swapargs()**, которая использует **X** как тип данных для обмениваемых значений. В **main()** функция **swapargs()** вызывается с аргументами трех различных типов:

int, **float** и **char**. Поскольку **swapargs()** является родовой функцией, компилятор автоматически создает три варианта **swapargs()**: один, который будет обменивать целые числа, другой для обмена чисел с плавающей точкой и третий для символов. Таким образом, одна и та же родовая функция **swapargs()** может быть использована для обмена аргументов любых типов данных.

Приведем несколько важных терминов, имеющих отношение к шаблонам. Во-первых, родовые функции (т. е. функции, объявления которых начинаются со предложения **template**) называют также *шаблонными функциями*. Оба этих термина будут использоваться в нашей книге. Когда компилятор создает конкретный вариант этой функции, говорят, что создается *специализация*. Ее также называют *порожденной функцией*. Про акт порождения функции говорят, что создается *экземпляр* родовой функции. В целом можно сказать, что порожденная функция является специализированным экземпляром родовой функции.

Функция с двумя родовыми типами

В предложении **template** можно определить более одного родового типа данных, если использовать список с разделяющими запятыми. Например, приведенная ниже программа создает шаблонную функцию, использующую два родовых типа:

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

template <class Type1, class Type2>
void myfunc(Type1 x, Type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "hi");

    myfunc(0.23, 10L);

    return 0;
}
```

В этом примере условные имена типов **Type1** и **Type2** заменяются компилятором на типы данных **int** и **char***, а также **double** и **long** соответственно, когда компилятор создает специфические экземпляры функции **myfunc()** в **main()**.

Явная перегрузка родовых функций

Хотя родовые функции сами перегружают себя по мере необходимости, вы также можете перегружать их явным образом. Формально это называется *явной специализацией*. Если вы перегружаете родовую функцию, тогда эта перегруженная функция замещает (скрывает) родовую функцию относительно этого специфического варианта. Рассмотрим, например, эту программу – модифицированный вариант уже известной вам программы обмена значений:

```
// Специализация родовой функции.

#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Внутри шаблонной функции swapargs.\n";
}

// Эта функция замещает родовой вариант swapargs() для int.
void swapargs(int &a, int &b) ← Явная перегрузка swapargs( )
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Внутри специализации swapargs для int.\n";
}

int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "Исходные i, j: " << i << ' ' << j << '\n';
    cout << " Исходные x, y: " << x << ' ' << y << '\n';
    cout << " Исходные a, b: " << a << ' ' << b << '\n';

    swapargs(i, j); // вызывается явно перегруженная swapargs()
```

Здесь вызывается перегруженный явно вариант **swapargs()**.

```

swapargs(x, y); // вызывается родовая swapargs()
swapargs(a, b); // вызывается родовая swapargs()

cout << "После обмена i, j: " << i << ' ' << j << '\n';
cout << "После обмена x, y: " << x << ' ' << y << '\n';
cout << "После обмена a, b: " << a << ' ' << b << '\n';

return 0;
}

```

Программа выводит на экран следующее:

```

Исходные i, j: 10 20
Исходные x, y: 10.1 23.3
Исходные a, b: x z
Внутри специализации swapargs для int.
Внутри шаблонной функции swapargs.
Внутри шаблонной функции swapargs.
После обмена i, j: 20 10
После обмена x, y: 23.3 10.1
После обмена a, b: z x

```

Как показывают комментарии к строкам программы, при вызове **swapargs(i, j)** активизируется перегруженный явным образом вариант родовой функции **swapargs()**, потому что родовая функция замещена явной перегрузкой.

Относительно недавно был предложен альтернативный вариант синтаксиса для обозначения явно перегруженной специализации функции. Этот более новый подход использует ключевое слово **template**. При использовании синтаксиса нового стиля для специализации, перегруженная функция **swapargs()** из последней программы будет выглядеть таким образом:

```

// Использование нового синтаксиса для специализации.
template void swapargs<int>(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Внутри специализации swapargs для int.\n";
}

```

Как видите, новый синтаксис использует конструкцию **template** для указания на специализацию. Тип данных, для которого создается специализация, помещается внутри угловых скобок вслед за именем

функции. Этот же синтаксис используется для специализации родовой функции любого типа. Хотя на сегодня у нового стиля синтаксиса специализации нет видимых преимуществ перед старым, однако в будущем, видимо, новый стиль окажется лучшим подходом.

Явная специализация шаблона позволяет вам подгонять вариант родовой функции под конкретные условия — например, чтобы получить выигрыш в производительности для некоторого определенного типа данных. Однако в принципе, если вам нужны различающиеся варианты функции для различных типов данных, лучше использовать перегруженные функции, а не шаблоны.

Цель

12.3. Родовые классы

В дополнение к родовым функциям вы можете также определить родовой класс. В этом случае вы создаете класс, который определяет все алгоритмы, используемые этим классом; однако конкретный тип обрабатываемых данных будет задан в виде параметра при создании объектов этого класса.

Родовые классы оказываются полезными в тех случаях, когда класс использует логику, поддающуюся обобщению. Например, алгоритм, обслуживающий очередь целых чисел, точно так же будет работать с очередью символов, а механизм управления связанным списком адресов рассылки так же хорошо справится с управлением связанным списком автомобильных деталей. Создаваемый вами родовой класс может выполнять определяемые вами операции, например, управление очередью или связанным списком, для любого типа данных. Компилятор автоматически сгенерирует правильный тип объекта, исходя из типа, который вы задаете при создании объекта.

Общая форма объявления родowego класса выглядит следующим образом:

```
template <class тип> class имя-класса {  
    // тело класса  
}
```

Здесь *тип* — это условное имя обобщенного, родowego типа, который будет задан при создании экземпляра класса. При необходимости вы с помощью списка с разделяющими запятыми можете определить более одного родowego типа данных.

После того, как вы создали родовой класс, вы создаете конкретный экземпляр этого класса посредством следующей общей формы:

```
имя-класса <тип> объект;
```

Здесь *тип* — это имя типа данных, с которыми будет работать класс. Функции-члены родового класса автоматически делаются родовыми. Вам не нужно использовать ключевое слово **template** для явного указания на их обобщенность.

Вот пример родового класса:

```
// Простой родовой класс.

#include <iostream>
using namespace std;

template <class T> class MyClass { ← Объявление родового класса.
    T x, y;                               Здесь T – это родовой тип.
public:
    MyClass(T a, T b) {
        x = a;
        y = b;
    }
    T div() { return x/y; }
};

int main()
{
    // Создание варианта MyClass для double.
    MyClass<double> d_ob(10.0, 3.0 ); ←
    cout << "деление чисел типа double: " << d_ob.div() << "\n";

    // Создание варианта MyClass для int.
    MyClass<int> i_ob(10, 3);
    cout << "деление чисел типа int: " << i_ob.div() << "\n";

    return 0;
}
```

Создание конкретного экземпляра
родового класса

Вот вывод этой программы:

```
Деление чисел типа double: 3.33333
Деление чисел типа int: 3
```

Как показывает этот вывод, объект **double** выполнил деление с плавающей точкой, а объект **int** — целочисленное деление.

Когда объявляется конкретный экземпляр класса **MyClass**, компилятор автоматически генерирует вариант функции **div()**, а также переменные **x** и **y** для хранения фактических данных. В этом примере объявляются два различных типа объектов. Первый, **d_ob**, обрабатывает данные типа **double**. Это значит, что **x** и **y** являются значениями типа **double**, и тип результата деления, как и возвращаемый функцией **div()**

тип — тоже **double**. Второй объект, **i_ob**, обрабатывает данные типа **int**. Поэтому **x**, **y** и возвращаемое из **div()** значение имеют тип **int**. Обратите особое внимание на эти объявления:

```
MyClass<double> d_ob(10.0, 3.0);
MyClass<int> i_ob(10, 3);
```

Посмотрите, как требуемый тип данных передается внутри угловых скобок. Изменяя тип данных, указываемый при создании объектов **MyClass**, мы можете изменять тип данных, обрабатываемых классом **MyClass**.

Шаблонный класс может иметь более одного родового типа данных. Просто объявите все требуемые классом типы данных посредством списка с разделяющими запятыми внутри спецификации **template**. Например, следующий пример создает класс, использующий два родовых типа данных:

```
/* Этот пример использует два родовых типа данных
   в определении класса. */

#include <iostream>
using namespace std;

template <class T1, class T2> class MyClass
{
    T1 i;
    T2 j;
public:
    MyClass(T1 a, T2 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 проверка");

    ob1.show(); // вывести int, double
    ob2.show(); // вывести char, char *

    return 0;
}
```

Программа выводит следующее:

```
10 0.23
X Это проверка
```

Программа объявляет два вида объектов. **ob1** использует данные типов **int** и **double**. **ob2** использует символ и указатель на символы. В обоих случаях компилятор автоматически генерирует подходящие данные и функции для обслуживания создаваемых объектов.

Явные специализации класса

Как и в случае шаблонных функций, вы можете создать специализацию родового класса. Для этого надо использовать конструкцию **template**, как это вы делали при создании явных специализаций шаблонных функций. Вот пример:

```
// Демонстрация специализации класса.

#include <iostream>
using namespace std;

template <class T> class MyClass {
    T x;
public:
    MyClass(T a) {
        cout << "Внутри родового класса MyClass\n";
        x = a;
    }
    T getx() { return x; }
};

// Явная специализация для int.
template class MyClass<int> { ← Это явная специализация класса MyClass
    int x;
public:
    MyClass(int a) {
        cout << "Внутри 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); ← Здесь используется явная специализация класса MyClass
    cout << "int: " << i.getx() << "\n";
}
```

```
    return 0;  
}
```

Эта программа выводит следующее:

Внутри родового класса MyClass
double: 10.1

Внутри специализации MyClass<int>
int: 25

В этой программе обратите особое внимание на строку:

```
template class MyClass<int> {
```

Она говорит компилятору, что создается явная целочисленная специализация класса **MyClass**. Тот же самый синтаксис используется при создании специализации любого типа для родового класса.

Явная специализация класса расширяет возможности применения родовых классов, поскольку она позволяет вам без труда контролировать один или два особых случая, в то время как все остальные автоматически обрабатываются компилятором. Конечно, если оказывается, что вы вынуждены создавать слишком много специализаций, тогда вам, видимо, лучше вообще не связываться с шаблонным классом.

Минутная тренировка

1. Какое ключевое слово используется для объявления родовой функции или класса?
 2. Можно ли явно перегрузить родовую функцию?
 3. Выполняется ли автоматическое объявление родовыми функций-членов родового класса?
- 1 Для объявления родовой функции или класса используется ключевое слово **template**
 - 2 Да, родовую функцию можно перегрузить явным образом
 - 3 Да, в родовом классе все его функции-члены автоматически делаются родовыми
-

Проект 12-1 Создание родового класса очереди

В проекте 8-2 вы создали класс **Queue**, поддерживающий очередь символов. В этом проекте вы преобразуете **Queue** в родовую класс, который сможет управлять данными любых типов. **Queue** является удачным кандидатом на преобразование в родовую класс, потому что его

логика не зависит от типа данных, с которыми он работает. Тот же механизм, что обслуживает очередь, скажем, целых чисел, годится и для очереди значений с плавающей точкой, и даже для объектов ваших собственных классов. Как только вы создали родовой класс `Queue`, вы сможете использовать его во всех случаях, когда вам понадобится очередь каких-либо объектов.

Шаг за шагом

1. Начните с копирования класса `Queue` из Проекта 8-2 в файл с именем `GenericQ.cpp`.
2. Измените объявление `Queue` на объявление шаблона, как показано ниже:

```
template <class QType> class Queue {
```

Здесь родовой тип данных назван `QType`.

3. Замените тип данных массива `q` на тип `QType`:

```
QType q[maxQsize]; // массив для хранения очереди
```

Поскольку `q` теперь принадлежит родовому типу, эту переменную можно будет использовать с любым типом данных, объявленным объектом `Queue`.

4. Замените тип данных параметра функции `put()` на `QType`, как показано ниже:

```
// Поместим данные в очередь.
void put(QType data) {
    if(putloc == size) {
        cout << " - Очередь полна.\n";
        return;
    }

    putloc++;
    q[putloc] = data;
}
```

5. Замените тип возврата функции `get()` на `QType`, как показано ниже:

```
// Извлечем данные из очереди.
QType get() {
    if(getloc == putloc) {
        cout << " - Очередь пуста.\n";
        return 0;
    }
}
```

```
    }  
    getloc++;  
  
    return q[getloc];  
}
```

6. Ниже приведена полная программа родового класса Queue вместе с функцией **main()**, демонстрирующей его использование:

```
/*  
    Проект 12-1  
  
    Шаблонный класс очереди.  
*/  
#include <iostream>  
using namespace std;  
  
const int maxQsize = 100;  
  
// Далее создается родовой класс очереди.  
template <class QType> class Queue {  
    QType q[maxQsize]; // массив для хранения очереди  
    int size; // максимальное число элементов,  
               // которые могут находиться в очереди  
    int putloc, getloc; // индексы "положить" и "взять"  
public:  
  
    // Сконструируем очередь конкретной длины.  
    Queue(int len) {  
        // Размер очереди должен быть меньше max и положителен.  
        if(len > maxQsize) len = maxQsize;  
        else if(len <= 0) len = 1;  
  
        size = len;  
        putloc = getloc = 0;  
    }  
    // Поместим данное в очередь.  
    void put(QType data) {  
        if(putloc == size) {  
            cout << " - Очередь полна.\n";  
            return;  
        }  
  
        putloc++;  
        q[putloc] = data;  
    }  
};
```

```
// Извлечем данные из очереди.
QType get() {
    if(getloc == putloc) {
        cout << " -- Очередь пуста.\n";
        return 0;
    }
    getloc++;
    return q[getloc];
}
};

// Демонстрация родового класса Queue.
int main()
{
    Queue<int> iQa(10), iQb(10); // создадим две очереди
                                // целых чисел

    iQa.put(1);
    iQa.put(2);
    iQa.put(3);

    iQb.put(10);
    iQb.put(20);
    iQb.put(30);

    cout << "Содержимое очереди целых чисел iQa: ";
    for(int i=0; i < 3; i++)
        cout << iQa.get() << " ";
    cout << endl;

    cout << "Содержимое очереди целых чисел iQb: ";
    for(int i=0; i < 3; i++)
        cout << iQb.get() << " ";
    cout << endl;

    Queue<double> dQa(10), dQb(10); // создадим две очереди
                                    // чисел с плавающей точкой

    dQa.put(1.01);
    dQa.put(2.02);
    dQa.put(3.03);

    dQb.put(10.01);
    dQb.put(20.02);
    dQb.put(30.03);
```

```
cout << "Содержимое очереди чисел с плавающей точкой dQa: ";
for(int i=0; i < 3; i++)
    cout << dQa.get() << " ";
cout << endl;

cout << "Содержимое очереди чисел с плавающей точкой dQb: ";
for(int i=0; i < 3; i++)
    cout << dQb.get() << " ";
cout << endl;

return 0;
}
```

Вот вывод этой программы:

```
Содержимое очереди целых чисел iQa: 1 2 3
Содержимое очереди целых чисел iQb: 10 20 30
Содержимое очереди чисел с плавающей точкой dQa: 1.01 2.02 3.03
Содержимое очереди чисел с плавающей точкой dQb: 10.01 20.02 30.03
```

7. Как видно из этого примера, родовые функции и классы являются мощным средством оптимизации процесса разработки программ, потому что они позволяют вам определять общую форму объекта, которую затем можно использовать применительно к любому типу данных. Вы избавляетесь от утомительной процедуры создания отдельных реализаций вашего алгоритма для каждого типа данных, с которыми будет использоваться этот алгоритм. Компилятор автоматически создает для вас конкретный вариант класса.

Цель

12.4. Динамическое выделение памяти

Имеются два основных способа, с помощью которых C++-программы могут сохранять информацию в основной памяти компьютера. Первый способ заключается в использовании переменных. Память, выделяемая под переменные, фиксируется во время компиляции и не может быть изменена в процессе выполнения программы. Второй способ хранения информации использует систему *динамического выделения памяти* C++. При таком способе память под данные выделяется по мере необходимости из объема свободной памяти, которая располагается между вашей программой (вместе с ее областью хранения постоянных данных) и стеком. Эта область памяти называется *кучей*. (На рис. 12-1 показано как в принципе выглядят C++-программы в памяти.)

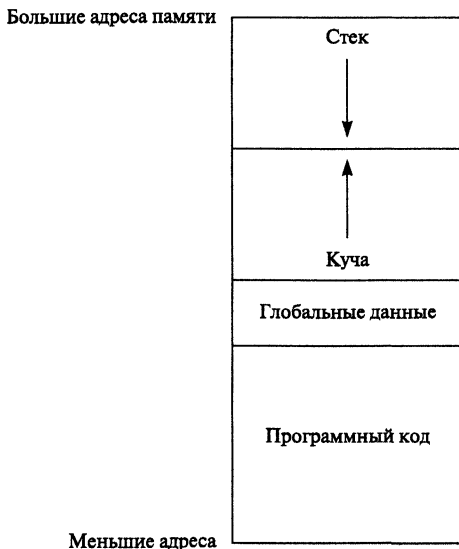


Рис. 12-1. Принципиальная схема распределения памяти для C++-программы

Динамическое выделение памяти осуществляется во время выполнения программы. Таким образом, динамическое выделение дает возможность вашей программе создавать переменные в процессе выполнения программы. Программа может создать столько переменных, сколько ей требуется в зависимости от ситуации. Динамическое выделение памяти часто используется для поддержки таких структур данных, как связанные списки, двоичные деревья и разреженные массивы. Разумеется, вы можете использовать динамическое выделение памяти во всех случаях, когда это будет иметь смысл. Динамическое выделение памяти с той или иной целью является важнейшей частью практически всех реальных программ.

Память для динамического выделения берется из кучи. Нетрудно догадаться, что в определенных крайних ситуациях эта память может исчерпаться. Таким образом, хотя динамическое выделение памяти обеспечивает значительное повышение гибкости, оно имеет свои ограничения.

C++ предоставляет два оператора для динамического выделения памяти: **new** и **delete**. Оператор **new** выделяет память и возвращает указатель на ее начало. Оператор **delete** освобождает память, предварительно выделенную посредством оператора **new**. Общая форма этих операторов приведена ниже:

```
p_var = new mun;  
delete p_var;
```

Здесь *p_var* представляет собой переменную-указатель, принимающую адрес памяти, объем которой достаточен для хранения элемента данных типа *тип*.

Поскольку куча имеет ограниченный объем, она может исчерпаться. Если в системе не хватит наличной памяти для выполнения всех запросов на ее выделение, оператор **new** не выделит память и выбросит исключение **bad_alloc**. Это исключение определено в заголовке **<new>**. Вашей программе следует обрабатывать это исключение с целью принятия соответствующих действий в случае отказа в выделении памяти. Если исключение **bad_alloc** вашей программой не обрабатывается, оно приведет к аварийному завершению программы.

Действия **new** в случае нехватки памяти, описанные выше, определены стандартным C++. Однако некоторые старые компиляторы реализуют **new** несколько иначе. Когда C++ только появился, **new** в случае отказа возвращал нулевой указатель. Позже алгоритм его выполнения был изменен, и теперь **new** при невозможности выделения памяти выбрасывает исключение, как это описано в предыдущих строках. Если вы используете старый компилятор, сверьтесь с документацией к нему, чтобы узнать, как в точности он реализует **new**.

Поскольку стандартный C++ определяет, что при отказе **new** генерирует исключение, именно такой алгоритм предполагается в программных примерах этой книги. Если ваш компилятор по-иному обрабатывает отказ выделения памяти, вам придется внести в программы соответствующие изменения.

Ниже приведена программа, которая выделяет память для хранения переменной типа **int**:

```
// Демонстрация new и delete.
```

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

```
int main()
```

```
{
```

```
    int *p;
```

```
    try {
```

```
        p = new int; // выделение места в памяти под int
```

```
    } catch (bad_alloc xa) {
```

```
        cout << "Отказ в выделении памяти\п";
```

```
        return 1;
```

```
    }
```

```
    *p = 100;
```

```
    cout << "По адресу " << p << " ";
```

Выделение памяти под **int**.

Ловим ошибку выделения памяти.

```
cout << "находится значение " << *p << "\n";

delete p; ←————— Освобождение выделенной памяти.

return 0;
}
```

Эта программа присваивает указателю **p** адрес области памяти из кучи, достаточной для хранения целого числа. Затем она записывает в эту память число 100 и выводит содержимое этой памяти на экран. После этого выделенная память освобождается.

Оператор **delete** должен использоваться только с указателем правильного типа и имеющим значение, которое было получено при выделении памяти оператором **new**. Использование с **delete** указателя любого другого типа не определено в языке и почти наверняка породит серьезные проблемы, возможно, фатальный отказ системы.

Спросим у эксперта

Вопрос: Мне встречались программы на C++, которые использовали для динамического выделения памяти функции **malloc()** и **free()**. Что это за функции?

Ответ: Язык C не поддерживает операторы **new** и **delete**. Для динамического выделения памяти C использует функции **malloc()** и **free()**. **malloc()** выделяет память, а **free()** ее освобождает. C++ также поддерживает эти функции, так что вы можете иногда увидеть **malloc()** и **free()** в C++-программе, особенно, если для нее за основу был взят старый код на C. Однако использовать **malloc()** и **free()** в ваших программах не следует. Операторы **new** и **delete** не только предоставляют более удобный способ управления динамическим выделением памяти, но они также предотвращают некоторые виды ошибок, весьма типичных при использовании **malloc()** и **free()**. Еще одно предупреждение: хотя и нет установленных правил запрета на использование **malloc()** и **free()**, совместное с **new** и **delete** в одной программе, однако мешать их не следует. Нет полной гарантии, что они взаимно совместимы.

Инициализация выделенной памяти

Вы можете инициализировать выделенную память каким-либо известным значением, указав значение-инициализатор после имени типа в предложении **new**. Вот общая форма **new**, если этот оператор используется вместе с инициализатором:

```
p_var = new тип (инициализатор);
```

Конечно, тип инициализатора должен быть совместим с типом *тип* данного, для которого выделяется память.

Приведенная ниже программа выделяет память под целочисленную переменную и инициализирует ее значением 87:

```
// Инициализация памяти.

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

int main()
{
    int *p;

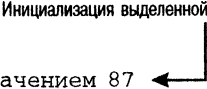
    try {
        p = new int (87); // инициализируем значением 87
    } catch (bad_alloc xa) {
        cout << "Ошибка выделения\n";
        return 1;
    }

    cout << "По адресу " << p << " ";
    cout << "находится значение " << *p << "\n";

    delete p;

    return 0;
}
```

Инициализация выделенной памяти



Выделение памяти под массивы

Вы можете выделить память под массив с помощью **new**, используя следующую общую форму:

```
p_var = new тип-массива [размер];
```

Здесь *размер* определяет число элементов в массиве.

Для освобождения памяти, выделенной под массив, следует использовать такую форму **delete**:

```
delete [ ] p_var;
```

Здесь квадратные скобки [] сообщают оператору **delete**, что необходимо освободить память, занимаемую массивом.

В качестве примера приведенная ниже программа выделяет память под 10-элементный массив целых чисел:

```
// Выделение памяти под массив.

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

int main()
{
    int *p, i;

    try {
        p = new int [10]; // выделим память под 10 чисел int
    } catch (bad_alloc xa) {
        cout << "Ошибка выделения\n";
        return 1;
    }

    for(i=0; i<10; i++ )
        p[i] = i;

    for(i=0; i<10; i++)
        cout << p[i] << " ";

    delete [] p; // освободим память

    return 0;
}
```

Выделение памяти под массив `int` ←

← Освобождение памяти массива

Обратите внимание на предложение **delete**. Как только что было упомянуто, если освобождается память, выделенная под массив с помощью **new**, то оператору **delete** необходимо сообщить, что освобождается массив, для чего и используются квадратные скобки []. (Как вы увидите в следующем подразделе, это особенно важно в тех случаях, когда память выделяется под массив объектов.)

Одно ограничение при выделении памяти под массивы: им нельзя присвоить начальное значение. Другими словами, выделяя память под массив, вы не можете указать инициализатор.

Выделение памяти под объекты

Используя оператор **new**, вы можете динамически выделять память под объекты. В этом случае создается объект и возвращается указатель

на него. Динамически созданный объект ведет себя подобно любому другому объекту. Когда он создается, вызывается его конструктор (если у объекта есть конструктор). Когда память объекта освобождается, вызывается его деструктор.

Приведенная ниже программа создает класс с именем **Rectangle**, который инкапсулирует ширину и высоту прямоугольника. Этот объект уничтожается перед завершением программы.

```
// Выделение памяти под объект.

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

class Rectangle {
    int width;
    int height;
public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
        cout << "Конструируем прямоугольник размером " << width <<
            " на " << height << ".\n";
    }
    ~Rectangle() {
        cout << "Уничтожаем прямоугольник размером " << width <<
            " на " << height << ".\n";
    }

    int area() {
        return width * height;
    }
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle(10, 8);
    } catch (bad_alloc xa) {
        cout << "Ошибка выделения\n";
        return 1;
    }

    cout << "Площадь равна " << p->area();
```

Выделение памяти под объект класса **Rectangle**. При этом вызывается конструктор **Rectangle**

```

cout << "\n";

delete p; ← Освобождение памяти, занятой объектом.
           При этом вызывается деструктор Rectangle

return 0;
}

```

Вывод программы выглядит таким образом:

Конструируем прямоугольник размером 10 на 8.

Площадь равна 80

Уничтожаем прямоугольник размером 10 на 8.

Обратите внимание на то, что в строке создания объекта значения аргументов для конструктора указаны в скобках после типа, как это делается при любого рода инициализации. Также заметьте, что, поскольку **p** содержит указатель на объект, для вызова функции **area()** используется оператор-стрелка (а не оператор-точка).

Вы можете выделить память под массив объектов, однако здесь есть скрытая ловушка. Поскольку при выделении памяти под массив с помощью **new** нельзя указать инициализатор, вы должны быть уверены, что если в классе определены конструкторы, среди них должен быть один без параметров. Если такого конструктора нет, компилятор не найдет нужного конструктора, когда вы попытаетесь выделить память под массив объектов, и программа компилироваться не будет.

В приведенном ниже варианте предыдущей программы в класс включен конструктор без параметров, чтобы можно было выделить память под массив объектов **Rectangle**. Кроме того, к составу класса добавлена функция **set()**, которая устанавливает размеры каждого прямоугольника.

// Выделение памяти под массив объектов.

```

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

class Rectangle {
    int width;
    int height;
public:
    Rectangle() { ← Добавим конструктор без параметров.
        width = height = 0;
        cout << "Конструируем прямоугольник размером " << width <<
            " на " << height << ".\n";
    }
    Rectangle(int w, int h) {
        width = w;

```

```
    height = h;
    cout << "Конструируем прямоугольник размером " << width <<
          " на " << height << ".\n";
}

~Rectangle() {
    cout << "Уничтожаем прямоугольник размером " << width <<
          " на " << height << ".\n";
}

void set(int w, int h) { ←————— Добавим функцию set( )
    width = w;
    height = h;
}

int area() {
    return width * height;
}
};

int main()
{
    Rectangle *p;

    try {
        p = new Rectangle [3];
    } catch (bad_alloc &a) {
        cout << "Ошибка выделения\n";
        return 1;
    }

    cout << "\n";

    p[0].set(3, 4);
    p[1].set(10, 8);
    p[2].set(5, 6);

    for(int i=0; i << 3; i++)
        cout << "Площадь равна " << p[i].area() << endl;

    cout << "\n";

    delete [] p; ←————— В этом месте для каждого объекта в массиве
                           вызывается деструктор.

    return 0;
}
```

Ниже приведен вывод этой программы:

Конструируем прямоугольник размером 0 на 0.
Конструируем прямоугольник размером 0 на 0.
Конструируем прямоугольник размером 0 на 0.

Площадь равна 12
Площадь равна 80
Площадь равна 30

Уничтожаем прямоугольник размером 5 на 6.
Уничтожаем прямоугольник размером 10 на 8.
Уничтожаем прямоугольник размером 3 на 4.

Поскольку указатель **p** освобождается с помощью **delete []**, деструктор вызывается для каждого объекта массива, как это и видно из вывода программы. Обратите внимание также на вызов функции **set()**. Индексированный указатель обозначает имя элемента массива, и поэтому при обращении к членам **Rectangle** используется оператор-точка.

Минутная тренировка

1. Какой оператор выделяет память? Какой оператор освобождает память?
 2. Что происходит, если запрос на выделение памяти не может быть выполнен?
 3. Можно ли инициализировать память при ее выделении?
- 1 Выделяет память оператор **new** Освобождает память оператор **delete**
 - 2 Если запрос на выделение памяти не может быть выполнен, выбрасывается исключение **bad_alloc**
 - 3 Да, память можно инициализировать при ее выделении
-

Цель

12.5. Пространства имен

Пространства имен уже были кратко описаны в Модуле 1. Здесь мы займемся ими более детально. Назначение пространства имен заключается в локализации имен идентификаторов с целью избежать конфликта имен. В среде программирования на C++ произошло буквально обвальное возрастание числа переменных, функций и имен классов. Перед изобретением пространств имен все эти имена боролись за места в глобальном пространстве имен, в результате чего часто возникали конфликты. Например, если ваша программа определила функцию с именем **toupper()**, она могла (в зависимости от своего списка па-

раметров) заместить стандартную библиотечную функцию `toupper()`, потому что оба имени сохранялись бы в глобальном пространстве имен. Проблемы конфликта имен усугублялись, если для одной и той же программы использовались две или больше библиотек сторонних производителей. В этом случае было возможно — даже весьма вероятно — что имена, определенные в одной библиотеке, стали бы конфликтовать с именами, определенными в другой. К особенно большим неприятностям приводили конфликты имен классов. Если, например, ваша программа определяла класс с именем `Stack`, а класс с таким же именем был уже определен в библиотеке, используемой вашей программой, неминуемо возникал конфликт.

Ответом на все эти проблемы стало создание ключевого слова `namespace`. Пространство имен, локализуя видимость объявленных в нем имен, позволяет одному и тому же имени использоваться в разных контекстах без возбуждения конфликта. Возможно, максимальную выгоду от использования пространств имен получила стандартная библиотека C++. Перед введением в оборот понятия пространства имен все библиотека C++ была определена в глобальном пространстве имен (которое, конечно, было единственным пространством). Введение `namespace` позволило определить для библиотеки C++ собственное пространство имен, названное `std`, что существенно уменьшило вероятность конфликтов имен. Вы можете также создать собственное пространство имен для локализации видимости любых имен, которые, на ваш взгляд, могут привести к конфликтам. Это становится особенно важным, если вы создаете библиотеки классов или функций.

Основы использования пространств имен

Ключевое слово `namespace` позволяет вам разделить на части глобальное пространство имен путем создания декларативного района. По существу `namespace` определяет область видимости. Общая форма объявления пространства имен выглядит таким образом:

```
namespace имя {  
    // объявления  
}
```

Все, что определяется внутри предложения `namespace`, находится в области видимости этого пространства имен.

Вот пример создания пространства имен. В нем локализуются имена, используемые для реализации простого класса счетчика для обратного счета. В пространстве имен определен класс `counter`, который реализует такой счетчик, а также переменные `upperbound` и `lowerbound`, которые содержат верхнюю и нижнюю границы, применимые ко всем счетчикам.

// Демонстрация пространства имен.

```
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;
        }
    };
}
```

← Создание пространства имен с именем **CounterNameSpace**

В этой программе **upperbound**, **lowerbound**, и класс **counter** являются частью области видимости, определенной пространством имен **CounterNameSpace**.

Внутри пространства имен обращение к объявленным в нем идентификаторам осуществляется непосредственно, без квалификации их именем пространства имен. Например, внутри **CounterNameSpace** функция **run()** может обращаться непосредственно к переменной **lowerbound** в предложении

```
if(count > lowerbound) return count--;
```

Однако, поскольку **namespace** определяет область видимости, для обращения к объектам, объявленным в пространстве имен, извне этого пространства, вы должны использовать оператор разрешения видимости **::**. Например, для присваивания значения 10 переменной **upperbound** из кода, не входящего в **CounterNameSpace**, вы должны использовать такое предложение:

```
CounterNameSpace::upperbound = 10;
```

Или, для объявления объекта типа **counter** вне **CounterNameSpace**, придется использовать предложение вроде этого:

```
CounterNameSpace::counter ob;
```

В общем случае для обращения к членам пространства имен вне этого пространства предваряйте имя члена именем пространства имен с оператором разрешения видимости.

Ниже приведена программа, демонстрирующая использование пространства имен **CounterNameSpace**:

```
// Демонстрация 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 {
```

Полностью квалифицированное обращение к членам **CounterNameSpace**. Обратите внимание на использование оператора разрешения видимости.


```

    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 объявление объекта **counter** и ссылки на переменные **upperbound** и **lowerbound** квалифицированы указанием **CounterNameSpace**. Однако после того, как объект типа **counter** объявлен, нет необходимости в дальнейшем квалифицировать его самого или какие-либо его члены. Таким образом, **ob1.run()** может быть вызвана непосредственно; пространство имен уже разрешено.

Допустимо использовать более одного объявления одного и того же пространства имен. В этом случае пространства имен складываются. Это дает возможность расщепить пространство имен по нескольким файлам или даже использовать части пространства имен в одном файле. Например, во фрагменте

```

namespace NS {
    int i;
}

// ...

namespace NS {
    int j;
}

```

пространство имен **NS** расщеплено на две части, но содержимое обеих частей входит в одно и то же пространство имен **NS**.

Последнее замечание: пространства имен могут вкладываться друг в друга. Другими словами, одно пространство имен может быть объявлено внутри другого.

Предложение using

Если ваша программа часто ссылается на члены пространства имен, то требование указывать имя пространства имен вместе с оператором разрешения видимости каждый раз, когда вы обращаетесь к его членам, становится весьма утомительным. Предложение **using** было придумано для облегчения этой процедуры. Предложение **using** имеет две общие формы:

```
using namespace имя;  
using имя::член;
```

В первом предложении *имя* обозначает имя пространства имен, к которому мы хотим получить доступ. Все члены, определенные внутри указанного пространства имен, становятся видимы (т. е. они становятся частью текущего пространства имен) и могут использоваться без дополнительной квалификации. Во втором предложении сделан видимым только один член пространства имен. Например, при наличии описанного выше пространства имен **CounterNameSpace** следующие предложения вполне правильны:

```
using CounterNameSpace::lowerbound; // только lowerbound видим  
lowerbound = 10; // ОК потому что lowerbound видим  
using namespace CounterNameSpace; // все члены видимы  
upperbound = 100; // ОК потому что все члены видимы
```


Приведенная ниже программа иллюстрирует использование ключевого слова **using** в примере из предыдущего подраздела:

```
// Демонстрация 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()
{
    // используем только upperbound из CounterNameSpace
    using CounterNameSpace::upperbound;  Использование конкретного члена CounterNameSpace

    // теперь для установки upperbound квалификация не требуется
    upperbound = 100;

    // квалификация все еще нужна для lowerbound и др.
    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 CounterNameSpace
    using namespace CounterNameSpace;  Использование всего пространства имен 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;
}
```

Программа иллюстрирует один важный момент: использование одного пространства имен не замещает другого. Когда вы подключаете пространство имен к области видимости, оно просто добавляет свои имена к любым другим пространствам имен, которые уже действуют. Таким образом, к концу программы к глобальному пространству имен оказались добавлены `std` и `CounterNameSpace`.

Безымянные пространства имен

Имеется специальный тип пространства имен, называемый *безымянным* (или *анонимным*) *пространством имен*, который позволяет вам создавать идентификаторы, уникальные для данного файла. Такое пространство имен имеет следующую общую форму:

```
namespace {
    // объявления
}
```

Безымянные пространства имен позволяют вам установить уникальные идентификаторы, которые будут известны только в области видимости одного файла. Другими словами, в файле, который содержит безымянное пространство имен, члены этого пространства могут использоваться непосредственно, без квалификации. Однако вне файла эти идентификаторы неизвестны.

Как уже упоминалось ранее, одним из способов ограничить область видимости глобального имени является объявление его как `static`. Хотя использование статических глобальных объявлений разрешено в C++, лучшим способом достичь той же цели является использование безымянного пространства имен.

Пространство имен `std`

Стандартный C++ определяет всю библиотеку в ее собственном пространстве имен с именем `std`. Именно по этой причине большинство программ этой книги включали такое предложение:

```
using namespace std;
```

В результате все пространство имен `std` подключалось к текущему пространству имен, что давало вам возможность обращаться к именам функций и классов, определенных в библиотеке, без квалификации их каждый раз посредством `std::`.

Конечно, вы можете при желании квалифицировать каждое имя библиотеки префиксом `std::`. Например, можно явным образом квалифицировать `cout` таким образом:

```
std::cout << "Явно квалифицируем cout посредством std::.";
```

Вам может оказаться нежелательным включение стандартной библиотеки C++ в глобальное пространство имен, если ваша программа мало использует эту библиотеку, или если ее использование может породить конфликты имен. Однако, если ваша программа содержит сотни ссылок на библиотечные имена, включение `std` в текущее пространство имен будет значительно более простой операцией, чем квалификация каждого имени индивидуально.

Минутная тренировка

1. Что такое пространство имен? С помощью какого ключевого слова можно создать пространство имен?
 2. Складываются ли пространства имен?
 3. Для чего предназначено ключевое слово `using`?
- 1 Пространство имен — это декларативное пространство, которое определяет область видимости. Оно создается с помощью ключевого слова **namespace**
 - 2 Да, пространства имен складываются
 - 3 **using** включает члены пространства имен в текущую область видимости
-

Цель

12.6. Статические члены классов

Из Модуля 7 вы узнали о ключевом слове **static**, которое там использовалось для модификации объявлений локальных и глобальных переменных. Помимо такого применения, описатель **static** может быть применен к членам класса. И переменные-члены, и функции-члены могут быть объявлены с описателем **static**. Ниже будут рассмотрены и те, и другие.

Статические переменные-члены

Если вы предворяете объявление переменной-члена словом **static**, вы сообщаете компилятору, что может существовать только

одна копия этой переменной, и что все объекты данного класса будут использовать эту единственную копию. В отличие от обычных данных-членов, для каждого объекта класса не создается индивидуальной копии статической переменной-члена. Сколько бы объектов класса вы не создали, будет существовать только один экземпляр статического данного-члена. В результате все объекты класса используют одну и ту же переменную. Когда создается первый объект, все статические переменные инициализируются нулем.

Если вы объявляете данное-член вида **static** внутри класса, вы не определяете его. Вы обязаны выполнить глобальное определение этой переменной где-то в другом месте, вне класса. При этом статическая переменная объявляется еще раз с использованием оператора разрешения видимости, чтобы указать, к какому классу она относится. Такое объявление выделяет под статическую переменную память.

Вот пример использования члена **static**:

```
// Использование статической переменной класса.

#include <iostream>
using namespace std;

class ShareVar {
    static int num; ← Объявление статического данного-члена. Его будут
                    использовать все экземпляры класса ShareVar
public:
    void setnum(int i) { num = i; };
    void shownum() { cout << num << " "; }
};

int ShareVar::num; // определим num ← Определение статического
                               данного-члена

int main()
{
    ShareVar a, b;

    a.shownum(); // выводит 0
    b.shownum(); // выводит 0

    a.setnum(10); // устанавливает num = 10

    a.shownum(); // выводит 10
    b.shownum(); // также выводит 10

    return 0;
}
```

Вот вывод этой программы:

```
0 0 10 10
```

Обратите внимание на то, что статическая целочисленная переменная **num** объявлена внутри класса **ShareVar** и определена как глобальная переменная. Как уже отмечалось выше, такая процедура необходима, потому что объявление **num** внутри **ShareVar** не выделяет для этой переменной память. C++ инициализирует **num** нулем, поскольку никакой другой инициализации нет. Именно поэтому оба первых вызова **shownum()** выводят 0. Далее объект **a** присваивает **num** значение 10. После этого оба объекта, и **a**, и **b**, используют **shownum()** для вывода значения **num**. Поскольку имеется только один экземпляр **num**, используемый и **a**, и **b**, оба вызова **shownum()** выводят 10.

Если статическая переменная описана в секции **public**, к ней можно обращаться с указанием имени класса, но без ссылки на конкретный объект. К ней также можно обратиться посредством объекта. Вот пример этого:

// Обращение к статической переменной с указанием имени класса.

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

```
class Test {
public:
    static int num;
    void shownum() { cout << num << endl; }
};
```

```
int Test::num; // определяем num
```

```
int main()
{
    Test a, b;
```

// Установим num, указав имя ее класса.

```
Test::num = 100; ←
```

Обращение к **num** посредством имени класса **Test**.

```
a.shownum(); // выводит 100
```

```
b.shownum(); // выводит 100
```

// Установим num, указав имя объекта.

```
a.num = 200; ←
```

Обращение к **num** посредством имени объекта **a**.

```
a.shownum(); // выводит 200
```

```
b.shownum(); // выводит 200

return 0;
}
```

Обратите внимание, как значение переменной `num` устанавливается с использованием имени ее класса:

```
Test::num = 100;
```

Эта переменная доступна также посредством объекта:

```
a.num = 200;
```

Оба способа допустимы.

Статические функции-члены

Функцию-член также можно объявить статической, хотя это используется довольно редко. Функция-член, объявленная как **static**, может обращаться только к другим статическим членам своего класса. (Разумеется, статическая функция-член имеет доступ к нестатическим глобальным данным и функциям.) Статическая функция-член не имеет указателя **this**. Виртуальными статические функции-члены быть не могут. Кроме того, их нельзя объявлять как **const** или **volatile**.

Статическая функция-член может быть активизирована объектом ее класса, или ее можно вызвать независимо от какого-либо объекта, используя имя класса и оператор разрешения видимости. Рассмотрим для примера приведенную ниже программу. В ней определена статическая переменная с именем **count**, которая хранит число объектов, существующих в каждый момент времени:

```
// Демонстрация статической функции-члена.
```

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

class Test {
    static int count;
public:

    Test() {
        count++;
        cout << "Конструируем объект " <<
```



```

        count << endl;
    }

    ~Test() {
        cout << "Уничтожаем объект " <<
            count << endl;
        count--;
    }

    static int numObjects() { return count; } ← Статическая функция-член
};

int Test::count;

int main() {
    Test a, b, c;

    cout << "Теперь имеются " <<
        Test::numObjects() <<
        " в наличии.\n\n";

    Test *p = new Test();
    cout << "После создания объекта Test " <<
        "имеются " <<
        Test::numObjects() <<
        " в наличии.\n\n";

    delete p;

    cout << "После удаления объекта" <<
        " имеются " <<
        a.numObjects() <<
        " в наличии.\n\n";

    return 0;
}

```

Вывод программы выглядит следующим образом:

```

Конструируем объект 1
Конструируем объект 2
Конструируем объект 3
Теперь имеются 3 в наличии.

```

```

Конструируем объект 4
После создания объекта Test имеются 4 в наличии.

```

Уничтожаем объект 4
После удаления объекта имеются 3 в наличии.

Уничтожаем объект 3
Уничтожаем объект 2
Уничтожаем объект 1

Обратите внимание, как в программе вызывается статическая функция-член `numObjects()`:

```
Test::numObjects()
```

В третьем вызове она активизируется посредством обычного синтаксиса с указанием имени объекта и оператора-точки.

Цель

12.7. Динамическая идентификация типов (RTTI)

Вы, возможно, до сих пор не сталкивались с динамической идентификацией типов (т. е. с определением типа во время выполнения программы), потому что это понятие отсутствует в непалиморфных языках, таких как С или традиционный BASIC. В непалиморфных языках нет необходимости в динамической идентификации типа, потому что тип каждого объекта всегда известен во время компиляции (т. е. когда программа пишется). Однако, в полиморфных языках, к которым относится и С++, могут быть ситуации, в которых тип объекта неизвестен во время компиляции, потому что природа объекта полностью определяется лишь во время выполнения программы. Как вы знаете, С++ реализует полиморфизм благодаря использованию иерархии классов, виртуальных функций и указателей на базовый класс. Указатель базового класса может быть использован для указания как на объекты базового класса, так и на *любые объекты, производные от этого базового класса*. Таким образом, не всегда возможно знать заранее, на объект какого типа будет указывать базовый указатель в каждый данный момент. Определение типа должно быть перенесено на время выполнения, для чего и используется механизм динамической идентификации типов (Runtime Type Identification, RTTI).

Для получения типа объекта предусмотрен оператор `typeid`. Для использования `typeid` необходимо подключить заголовок `<typeinfo>`. Чаще всего используется такая форма этого оператора:

```
typeid(объект)
```

Здесь *объект* является тем объектом, у которого требуется определить тип. Этот объект может быть любого типа, включая встроенные типы и типы создаваемых вами классов. **typeid** возвращает ссылку на объект типа **type_info**, который описывает тип объекта *объект*.

Класс **type_info** определяет следующие открытые 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 `!=` предоставляют возможность сравнения типов. Функция **before()** возвращает **true**, если вызывающий ее объект расположен (в порядке сравнения) до объекта, используемого в качестве параметра. (Эта функция в основном предназначена для внутреннего использования. Возвращаемое ею значение не имеет никакого отношения к наследованию или иерархии классов.) Функция **name()** возвращает указатель на имя типа.

Вот простой пример использования **typeid**:

```
// Простой пример использования typeid.
```

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

class MyClass {
    // ...
};

int main()
{
    int i, j;
    float f;
    MyClass ob;

    cout << "Тип i: " << typeid(i).name();
    cout << endl;
    cout << "Тип f: " << typeid(f).name();
    cout << endl;
    cout << "Тип ob: " << typeid(ob).name();
    cout << "\n\n";

    if(typeid(i) == typeid(j))
        cout << "Типы i и j совпадают\n";

    if(typeid(i) != typeid(f))
```

typeid позволяет получить тип объекта во время выполнения программы

```
    cout << "Типы i и f различны\n";

    return 0;
}
```

Программа выводит следующее:

```
Тип i: int
Тип f: float
Тип ob: MyClass

Типы i и j совпадают
Типы i и f различны
```

Пожалуй, наиболее важным является такое использование **typeid**, когда этот оператор применяется к указателю на базовый полиморфный класс (т. е. класс, включающий в себя по меньшей мере одну виртуальную функцию). В этом случае **typeid** автоматически возвращает тип объекта, на который фактически указывает указатель; это может быть объект базового класса или объект класса, производного от базового. (Вспомним, что указатель базового класса может указывать как на объекты базового класса, так и на объекты любого класса, производного от этого базового.) Таким образом, используя **typeid**, вы можете определить во время выполнения тип объекта, на который указывает указатель базового класса. Приведенная ниже программа иллюстрирует эту возможность:

```
// Пример использования typeid в иерархии полиморфных классов.

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

class Base {
    virtual void f() {}; // сделаем Base полиморфным
    // ...
};

class Derived1: public Base {
    // ...
};

class Derived2: public Base {
    // ...
};
```

```
int main()
{
    Base *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    p = &baseob;
    cout << "p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    return 0;
}
```

Вывод программы выглядит так:

```
p указывает на объект типа Base
p указывает на объект типа Derived1
p указывает на объект типа Derived2
```

Как показывает этот вывод, если оператор **typeid** применен к указателю базового полиморфного класса, то тип объекта, на который указывает этот указатель, будет определен динамически, во время выполнения программы.

Если **typeid** применен к базовому указателю непалиморфной иерархии классов, то во всех случаях будет получен базовый тип указателя. Другими словами, для непалиморфной иерархии не выполняется идентификация типа объекта, на который фактически указывает указатель. В качестве эксперимента прокомментируйте объявление виртуальной функции **f()** в **Base** и посмотрите на результат. Вы увидите, что типы всех объектов будут **Base**, потому что это и есть тип указателя.

Поскольку **typeid** обычно применяется к указателю со снятой ссылкой (т. е. к указателю, перед которым указан знак *****), предусмотрено специальное исключение для обработки ситуации, когда снимается ссылка с нулевого указателя. В этом случае **typeid** выбрасывает исключение **bad_typeid**.

Ссылки на объекты полиморфной иерархии классов действуют аналогично указателям. Если оператор **typeid** применен к ссылке на

объект полиморфного класса, он вернет тип объекта, на который фактически указывает эта ссылка, т. е., возможно, объекта производного класса. Чаще всего вам придется использовать такую возможность в ситуациях, когда объекты передаются в функции по ссылке.

Имеется и вторая форма **typeid**, которая в качестве аргумента использует имя типа:

```
typeid(имя-типа)
```

Например, следующее предложение вполне правильно:

```
cout << typeid(int).name();
```

Основное использование такой формы **typeid** — получить объект **type_info**, который описывает конкретный тип, чтобы его можно было использовать в предложении сравнения типов.

Минутная тренировка

1. Чем замечательна статическая переменная-член?
 2. Для чего предназначен оператор **typeid**?
 3. Какого типа объект возвращает **typeid**?
- 1 Статическая переменная-член существует в одном экземпляре и доступна всем объектам этого класса.
 - 2 Оператор **typeid** определяет тип объекта во время выполнения программы.
 - 3 Оператор **typeid** возвращает объект типа **type_info**
-

Цель

12.8. Операторы приведения типа

В C++ определены пять операторов приведения типа. Первый — это оператор традиционного стиля; он уже был описан в этой книге. Этот оператор был включен в C++ с самого начала разработки языка. Оставшиеся четыре оператора были добавлены лишь несколько лет назад; это: **dynamic_cast**, **const_cast**, **reinterpret_cast** и **static_cast**. Эти операторы предоставляют дополнительный контроль над способом приведения типа. Каждый из них кратко описан ниже.

dynamic_cast

Это, возможно, самый важный из дополнительных операторов приведения. Оператор **dynamic_cast** выполняет приведение типа во

время выполнения программы с контролем его допустимости. Если в тот момент, когда выполняется **dynamic_cast**, приведение недопустимо, оно не производится. Общая форма **dynamic_cast** выглядит следующим образом:

dynamic_cast <тип-мишени> (выражение)

Здесь *тип-мишени* задает тип, к которому выполняется приведение, а *выражение* — это выражение, которое приводится к новому типу. Тип мишени должен быть указателем или ссылкой, а приводимое выражение тоже должно давать в результате указатель или ссылку. Таким образом, **dynamic_cast** может использоваться только для приведения одного типа указателя к другому или одного типа ссылки к другому.

Оператор **dynamic_cast** используется для приведений полиморфных типов. Пусть, например, даны два полиморфных класса **B** и **D**, причем **D** является производным от **B**. **dynamic_cast** всегда может привести указатель **D*** к указателю **B***, так как базовый указатель всегда может указывать на производный объект. Однако **dynamic_cast** может привести указатель **B*** к указателю **D***, только если указываемый объект *действительно является* объектом **D**. В общем случае **dynamic_cast** успешно выполнится, если приводимый указатель (или ссылка) указывает (или ссылается) либо на объект типа мишени, либо на объект, производный от типа мишени. В противном случае приведение не состоится. В случае ошибки приведения могут быть два варианта. Если приведение выполняется над указателями, возвращается нулевой указатель. Если приведение осуществляется над ссылками, выбрасывается исключение **bad_cast**.

Ниже приведен простой пример. Предположим, что **Base** является полиморфным классом, а **Derived** — это класс, производный от **Base**.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // базовый указатель указывает на объект Derived
dp = dynamic_cast<Derived *> (bp); // приведение к
// производному указателю OK
if(dp) cout << "Приведение OK";
```

Здесь приведение базового указателя **bp** к производному указателю **dp** возможно, так как **bp** фактически указывает на объект класса **Derived**. В результате этот фрагмент выводит на экран сообщение “Приведение ОК”. Однако в следующем фрагменте приведение не выполняется, так как **bp** указывает на объект класса

Base, а приведение базового объекта к производному объекту не-допустимо:

```
bp = &b_ob; // базовый указатель указывает на объект Base
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout << "Ошибка приведения";
```

Поскольку приведение не состоялось, этот фрагмент выведет “Ошибка приведения”.

const_cast

Оператор **const_cast** используется для того, чтобы явным образом заместить **const** или **volatile** (или и то, и другое) в операции приведения. Тип мишени должен быть тем же самым, что и тип источника, за исключением отсутствия **const** или **volatile**. Чаше всего **const_cast** используется для снятия атрибута **const**. Общая форма **const_cast** такова:

const_cast <тип> (*выражение*)

Здесь *тип* задает тип мишени приведения, а *выражение* — это выражение, которое приводится к новому типу. Следует подчеркнуть, что использование **const_cast** для снятия атрибута **const** является потенциально опасным мероприятием. Применяйте его с осторожностью.

Еще одно замечание: только **const_cast** может снять атрибут **const**. Операторы **dynamic_cast**, **reinterpret_cast** и **static_cast** не могут этого сделать.

static_cast

Оператор **static_cast** выполняет непалиморфное приведение. Его можно использовать для любого стандартного преобразования типов. В этом случае не выполняется проверка типа во время выполнения программы. Таким образом, оператор **static_cast** просто является заменой исходного оператора приведения. Общая форма этого оператора такова:

static_cast <тип> (*выражение*)

Здесь *тип* задает тип мишени приведения, а *выражение* — это выражение, которое приводится к новому типу.

reinterpret_cast

Оператор `reinterpret_cast` преобразует один тип в принципиально другой тип. Например, он может преобразовать указатель в целое или целое в указатель. Этот оператор можно также использовать для приведения внутренне несовместимых типов указателей. Его общая форма:

`reinterpret_cast <тип> (выражение)`

Здесь *тип* задает тип мишени приведения, а *выражение* — это выражение, которое приводится к новому типу.

Что дальше?

Целью этой книги было познакомить вас с базовыми элементами языка. Это средства и приемы C++, используемые в каждом дне программирования. Со знаниями, которыми вы теперь обладаете, вы можете приступить к написанию реальных программ профессионального качества. Однако, C++ является весьма богатым языком, и в нем содержится много дополнительных средств, к обладанию которыми вам следует стремиться. К этим средствам относятся:

- стандартная библиотека шаблонов (Standard Template Library, STL);
- явные конструкторы;
- функции преобразования;
- функции-члены с описателем **const** и ключевое слово **mutable**;
- ключевое слово **asm**;
- перегрузка оператора индексирования массивов `[]`, оператора вызова функции `()` и операторов динамического выделения памяти **new** и **delete**.

Из перечисленного, возможно, самым важным средством является стандартная библиотека шаблонов. Это библиотека шаблонных классов, предоставляющая готовые решения для большого количества типичных задач организации и хранения данных. Например, STL определяет такие родовые структуры, как очереди, стеки и списки, которые часто используются в программах.

Вам также следует изучить библиотеку функций C++. Она содержит богатый набор подпрограмм, которые существенно упростят ваш труд по разработке собственных программ.

Для желающих продолжить изучение C++ я рекомендую свою книгу *C++: The Complete Reference*, Osborne/McGraw-Hill, Berkeley, California. В ней рассматриваются темы, перечисленные в приведен-

ном выше списке, а также многое, многое другое. Теперь вы обладаете достаточными знаниями, чтобы эффективно использовать этот достаточно глубокий путеводитель по C++.

✓ Вопросы для самопроверки

1. Объясните, как **try**, **catch** и **throw** совместно поддерживают обработку исключений.
2. Как следует организовать список **catch**, чтобы ловились исключения как базового, так и производных классов?
3. Покажите, как задать, что из функции с именем **func()**, которая возвращает **void**, может быть выброшено исключение **MyExcept**.
4. Определите исключение для родового класса **Queue**, рассмотренного в Проекте 12-1. Пусть **Queue** выбрасывает это исключение, когда происходит переполнение или потеря значимости. Протестируйте использование этого исключения.
5. Что такое родовая функция, и с помощью какого ключевого слова она создается?
6. Создайте родовой вариант функций **quicksort()** и **qs()**, описанных в Проекте 5-1. Протестируйте их использование.
7. Используя класс **Sample**, приведенный ниже, создайте очередь из трех объектов **Sample** с помощью родового класса **Queue**, рассмотренного в Проекте 12-1:

```
class Sample {  
    int id;  
public:  
    Sample() { id = 0; }  
    Sample(int x) { id = x; }  
    void show() { cout << id << endl; }  
};
```

8. Разработайте вариант вашего ответа на п. 7, в котором для объектов **Sample**, помещаемых в очередь, память выделяется динамически.
9. Покажите, как объявить пространство имен под названием **RobotMotion**.
10. Какое пространство имен содержит стандартную библиотеку C++?
11. Может ли статическая функция-член обратиться к нестатическим данным класса?
12. Какой оператор позволяет получить тип объекта во время выполнения программы?

13. Какой оператор приведения типов следует использовать, чтобы определить допустимость полиморфного приведения во время выполнения?
14. Для чего предназначен оператор `const_cast`?
15. Попробуйте самостоятельно поместить класс **Queue** из Проекта 12-1 в его собственное пространство имен под названием **QueueCode**, и в его собственный файл с именем **Queue.cpp**. Затем преобразуйте функцию `main()` так, чтобы она использовала предложение `using` для подключения **QueueCode** к области видимости.
16. Продолжите изучение C++. Этот язык является на сегодня самым мощным компьютерным языком. Его освоение сделает вас членом элитной лиги программистов.

Приложение А

Ответы на Вопросы для самопроверки

Модуль 1. Основы C++

1. C++ находится в центре современного мира программирования, потому что он произошел от C и является родителем языков Java и C#. Это четыре важнейших языка программирования.

2. Справедливо. Компилятор C++ создает код, который может быть непосредственно выполнен компьютером.

3. Тремя основными принципами объектно-ориентированного программирования являются инкапсуляция, полиморфизм и наследование.

4. Выполнение C++-программы начинается с функции `main()`.

5. Заголовок содержит информацию, используемую программой.

6. `<iostream>` представляет собой заголовок, поддерживающий ввод-вывод. Указанное предложение включает заголовок `<iostream>` в программу.

7. Пространство имен — это декларативная область, в которую могут быть помещены различные программные элементы. Элементы, объявленные в одном пространстве имен, отделены от элементов, объявленных в другом пространстве.

8. Переменная — это поименованная ячейка памяти. Содержимое переменной может быть изменено по ходу выполнения программы.

9. Неправильны имена `d` и `e`. Имена переменных не могут начинаться с цифры или совпадать с ключевыми словами C++.

10. Однострочный комментарий начинается с символов `//` и заканчивается в конце строки. Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`.

11. Общая форма предложения `if`:

`if (условие) предложение;`

Общая форма предложения `for`:

`for(инициализация; условие; приращение);`

12. Программный блок начинается символом `{` и заканчивается символом `}`.

13.

`// Таблица соответствия веса на Земле и Луне.`

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

```
int main() {
    double earthweight; // вес на Земле
    double moonweight; // вес на Луне

    int counter;
    counter = 0;

    for(earthweight = 1.0; earthweight <= 100.0; earthweight++)
    {
        moonweight = earthweight * 0.17;
        cout << earthweight << " земных фунтов эквивалентно " <<
            moonweight << " лунным фунтам.\n";
        counter++;
        if(counter == 25) {
            cout << "\n";
            counter = 0;
        }
    }

    return 0;
}
```

14.

// Преобразование юпитерианских лет в земные.

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

int main() {
    double e_years; // годы на Земле
    double j_years; // годы на Юпитере

    cout << "Введите число лет на Юпитере: ";
    cin >> j_years;

    e_years = j_years * 12.0;

    cout << "Эквивалентное число лет на Земле: " << e_years;

    return 0;
}
```

15. Когда вызывается функция, программное управление передается на эту функцию.

16.

```
// Среднее абсолютных значений 5 чисел.
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int i;
    double avg, val;

    avg = 0.0;
    for(i=0; i<5; i++) {
        cout << "Введите значение: ";
        cin >> val;

        avg = avg + abs(val);
    }
    avg = avg / 5;

    cout << "Среднее абсолютных значений: " << avg;

    return 0;
}
```

Модуль 2. Знакомимся с данными, типами и операторами

1. В C++ имеются следующие типы целых:

| | | |
|--------------|--------------------|-------------------|
| int | short int | long int |
| unsigned int | unsigned short int | unsigned long int |
| signed int | signed short int | signed long int |

Тип **char** также может быть использован в качестве целочисленного типа.

2. 12.2 имеет тип **double**.

3. Переменная **bool** может принимать значения **true** или **false**.

4. Длинному целому соответствует тип **long int**, или просто **long**.

5. Символу табуляции соответствует последовательность `\t`. Звуковой сигнал создается последовательностью `\b`.
6. Справедливо. Строка окружается двойными кавычками.
7. Шестнадцатеричные цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
8. Для инициализации переменной используйте следующую общую форму:

тип переменная = значение;

9. Оператор `%` выполняет деление по модулю. Он возвращает остаток от целочисленного деления. Этот оператор недопустимо использовать со значениями с плавающей точкой.

10. Если оператор инкремента предшествует своему операнду, C++ выполнит соответствующую операцию до получения значения операнда, которое он будет использовать в оставшейся части выражения. Если оператор следует за своим операндом, то C++ получит значение операнда до его инкремента.

11. A, C и E.

12. `x += 12;`

13. Приведение типа представляет собой явное преобразование типа.

14. Ниже показан один из возможных способов получения простых чисел в диапазоне от 1 до 100. Имеются и другие способы.

// Найдем простые числа в диапазоне от 1 до 100.

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

int main() {
    int i, j;
    bool isprime;

    for(i=1; i < 100; i++) {
        isprime = true;
        // посмотрим, делится ли это число без остатка
        for(j=2; j <= i/2; j++)
            // если да, оно не простое
            if((i%j) == 0) isprime = false;

        if(isprime)
            cout << i << " простое.\n";
    }

    return 0;
}
```


Модуль 3. Предложения управления программой

1.

```
// Посчитаем точки.
#include <iostream>
using namespace std;

int main() {
    char ch;
    int periods = 0;

    cout << "Введите $ для завершения.\n";

    do {
        cin >> ch;
        if(ch == '.') periods++;
    } while(ch != '$');

    cout << "Точки: " << periods << "\n";

    return 0;
}
```

2. Может. Если последовательность предложений ветви **case** не завершается предложением **break**, тогда выполнение программных строк продолжится в следующую ветвь **case**. Предложение **break** предохраняет от этого.

3.

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

4. Последнее **else** относится к внешнему **if**, которое является ближайшим **if** на том же уровне, что и **else**.

5.

```
for(int i = 1000; i >= 0; i -= 2) // ...
```

6. Нет. Согласно стандарту C++ ANSI/ISO, переменная *i* неизвестна вне цикла **for**, в котором она объявлена. (Заметьте, что некоторые компиляторы могут обрабатывать эту ситуацию по-иному.)

7. **break** приводит к завершению ближайшего к нему цикла или предложения **switch**.

8. После выполнения **break** на экран будет выведено “после while”.

9.

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

10.

```
/*
   Использование цикла для генерации прогрессии
   1 2 4 8 16, ...
*/
#include <iostream>
using namespace std;

int main() {

    for(int i = 1; i < 100; i += i)
        cout << i << " ";

    cout << "\n";

    return 0;
}
```

11.

// Изменение регистра букв.

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

int main() {
    char ch;
```

```
int changes = 0;

cout << "Введите точку для завершения.\n";

do {
    cin >> ch;
    if(ch >= 'a' && ch <= 'z') {
        ch -= (char) 32;
        changes++;
        cout << ch;
    }
    else if(ch >= 'A' && ch <= 'Z') {
        ch += (char) 32;
        changes++;
        cout << ch;
    }
} while(ch != '.');

cout << "\nИзменение регистра: " << changes << "\n";

return 0;
}
```

12. Предложением безусловного перехода в C++ является **goto**.

Модуль 4. Массивы, строки и указатели

1.

```
short int hightemps[31];
```

2. нуля

3.

```
// Найдем одинаковые числа
#include <iostream>
using namespace std;
```

```
int main()
{
    int nums[] = {1, 1, 2, 3, 4, 2, 5, 4, 7, 7};
```

```

for(int i=0; i < 10; i++)
    for(int j=i+1; j < 10; j++)
        if(nums[i] == nums[j])
            cout << "Одинаковые: " << nums[i] << "\n";

return 0;
}

```

4. Строка, завершающаяся нулем – это массив символов, который заканчивается байтом с нулевым значением.

5.

```

// Игнорирование регистра букв при сравнении строк.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    char str1[80];
    char str2[80];
    char *p1, *p2;

    cout << "Введите первую строку: ";
    cin >> str1;
    cout << "Введите вторую строку: ";
    cin >> str2;

    p1 = str1;
    p2 = str2;

    // цикл, пока p1 и p2 указывают на ненулевые символы
    while(*p1 && *p2) {
        if(tolower(*p1) != tolower(*p2)) break;
        else {
            p1++;
            p2++;
        }
    }

    /* строки совпадают, если и p1, и p2
       указывают на завершающие нули.
    */
    if(!*p1 && !*p2)
        cout << "Строки совпадают за исключением возможного " <<
            "несовпадения регистра букв.\n";
    else

```

```

cout << "Строки различаются \n";

return 0;
}

```

6. При использовании **strcat()** массив-приемник должен быть достаточно велик, чтобы вместить содержимое обеих строк.

7. В многомерном массиве каждый индекс указывается в собственной паре квадратных скобок.

8. `int nums[] = {5, 66, 88};`

9. Объявление массива неопределенной длины дает уверенность в том, что инициализированный массив всегда будет достаточно велик для размещения в нем всех указанных инициализаторов.

10. Указатель представляет собой объект, содержащий адрес памяти. Операторами указателей являются `&` и `*`.

11. Да, указатель можно индексировать так, как это делается с массивом. Да, к массиву можно обратиться посредством указателя.

12.

// Подсчет числа прописных букв.

```

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

int main()
{
    char str[80];
    int i;
    int count;

    strcpy(str, "This Is A Test");

    count = 0;
    for(i=0; str[i]; i++)
        if(isupper(str[i])) count++;

    cout << str << " содержит " << count << " прописных букв.";

    return 0;
}

```

13. Если один указатель указывает на другой указатель, то используется термин *вложенная косвенность*.

14. По соглашению нулевое значение указателя обозначает, что этот указатель не используется.

Модуль 5. Введение в функции

1. Общая форма определения функции такова:

```
тип-возврата имя-функции (список-параметров)
{
    // тело функции
}
```

2.

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

double hypot(double a, double b);

int main() {

    cout << "Гипотенуза прямоугольного треугольника 3 на 4: ";
    cout << hypot(3.0, 4.0) << "\n";

    return 0;
}

double hypot(double a, double b)
{
    return sqrt((a*a) + (b*b));
}
```

3. Да, функция может вернуть указатель. Нет, функция не может вернуть массив.

4.

```
// Собственный вариант функции strlen().
#include <iostream>
using namespace std;

int mystrlen(char *str);

int main()
{
    cout << "Длина строки Привет вам! равна: ";
    cout << mystrlen("Привет вам!");

    return 0;
}
```

```
}  
// Собственный вариант функции strlen().  
int mystrlen(char *str)  
{  
    int i;  
  
    for(i=0; str[i]; i++) ; // найдем конец строки  
  
    return i;  
}
```

5. Нет, значение локальной переменной теряется, когда происходит выход из функции. (Или, в более общем виде, значение локальной переменной теряется, когда происходит выход из ее блока.)

6. Основное преимущество глобальных переменных заключается в том, что они доступны всем остальным функциям в программе, и что они существуют в течение всего времени жизни программы. Их основной недостаток заключается в том, что они потребляют память все время, пока программа выполняется. Использование глобальной переменной, когда можно обойтись локальной, делает функцию менее универсальной, а использование большого числа глобальных переменных может привести к неожиданным побочным эффектам.

7.

```
#include <iostream>  
using namespace std;  
  
int seriesnum = 0;  
  
int byThrees();  
void reset();  
  
int main() {  
  
    for(int i=0; i < 10; i++)  
        cout << byThrees() << " ";  
  
    cout << "\n";  
  
    reset();  
  
    for(int i=0; i < 10; i++)  
        cout << byThrees() << " ";  
  
    cout << "\n";  
  
    return 0;
```

```

}

int byThrees()
{
    int t;

    t = seriesnum;
    seriesnum += 3;

    return t;
}

void reset()
{
    seriesnum = 0;
}

```

8.

```

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

int main(int argc, char *argv[])
{
    if(argc != 2) {
        cout << "Требуется пароль!\n";
        return 0;
    }

    if(!strcmp("mypassword", argv[1]))
        cout << "Доступ разрешен.\n";
    else
        cout << "Доступ запрещен.\n";

    return 0;
}

```

9. Справедливо. Прототип предотвращает вызов функции с неправильным числом аргументов.

10.

```

#include <iostream>
using namespace std;

void printnum(int n);

```



```
int main()
{
    printnum(10);

    return 0;
}
void printnum(int n)
{
    if(n > 1) printnum(n-1);
    cout << n << " ";
}
```

Модуль 6. Подробнее о функциях

1. Аргумент может быть передан в подпрограмму по значению и по ссылке.

2. Ссылка является неявным указателем. Чтобы создать параметр-ссылку, надо предварить имя параметра знаком &.

3. `f(ch, &i);`

4.

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

```
void round(double &num);
```

```
int main()
{
    double i = 100.4;

    cout << i << " округленное равно ";
    round(i);
    cout << i << "\n";

    i = -10.9;
    cout << i << " округленное равно ";
    round(i);
    cout << i << "\n";

    return 0;
}
```

Ответы на Вопросы для самопроверки

```
void round(double &num)
{
    double frac;
    double val;

    // разделим num на целую и дробную части
    frac = modf(num, &val);

    if(frac < 0.5) num = val;
    else num = val+1.0;
}
```

5.

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

// Обменять аргументы и вернуть меньший.
int &min_swap(int &x, int &y);

int main()
{
    int i, j, min;

    i = 10;
    j = 20;

    cout << "Исходные значения i и j: ";
    cout << i << ' ' << j << '\n';

    min = min_swap(j, i);

    cout << "Новые значения i и j: ";
    cout << i << ' ' << j << '\n';

    cout << "Меньшее значение: " << min << "\n";
    return 0;
}

// Обменять аргументы и вернуть меньший.
int &min_swap(int &x, int &y)
{
    int temp;
    // для обмена значений аргументов используем ссылки
    temp = x;
    x = y;
    y = temp;
}
```

```
// вернем ссылку на меньший аргумент
if (x < y) return x;
else return y;
}
```

6. Функция не должна возвращать ссылку на локальную переменную, потому что когда происходит выход из функции, локальная переменная выходит из области видимости (т. е. перестает существовать).

7. Перегруженные функции должны различаться типом или числом своих параметров (или и тем, и другим).

8.

```
/*
Проект 6-1 - модифицирован для раздела
"Вопросы для самопроверки"

Создание перегруженных функций println(),
которые выводят данные различных типов.
Этот вариант включает параметр отступа.
*/

#include <iostream>
using namespace std;

// Эти функции выводят символ новой строки.
void println(bool b, int ident=0);
void println(int i, int ident=0);
void println(long i, int ident=0);
void println(char ch, int ident=0);
void println(char *str, int ident=0);
void println(double d, int ident=0);

// Эти функции не выводят символ новой строки.
void print(bool b, int ident=0);
void print(int i, int ident=0);
void print(long i, int ident=0);
void print(char ch, int ident=0);
void print(char *str, int ident=0);
void print(double d, int ident=0);

int main()
{
    println(true, 10);
    println(10, 5);
    println("Это просто проверка ");
    println('x');
    println(99L, 10);
}
```

```

println(123.23, 10);

print("Вот несколько значений: ");
print(false);
print(88, 3);
print(100000L, 3);
print(100.01);
println(" Выполнено!");

return 0;
}
// Несколько функций println().
void println(bool b, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    if(b) cout << "true\n";
    else cout << "false\n";
}

void println(int i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i << "\n";
}

void println(long i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i << "\n";
}

void println(char ch, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << ch << "\n";
}

void println(char *str, int ident)

```

```
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << str << "\n";
}

void println(double d, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << d << "\n";
}

// Несколько функций print().
void print(bool b, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    if(b) cout << "true";
    else cout << "false";
}

void print(int i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i;
}

void print(long i, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << i;
}

void print(char ch, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << ch;
```

```

}

void print(char *str, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << str;
}

void print(double d, int ident)
{
    if(ident)
        for(int i=0; i < ident; i++) cout << ' ';

    cout << d;
}

```

9.

```

myfunc('x');
myfunc('x', 19);
myfunc('x', 19, 35);

```

10. Перегрузка функций может привести к неоднозначности, если компилятор не может определить, какой вариант функции надо вызывать. Это может случиться, когда выполняется автоматическое преобразование типов, а также когда используются аргументы по умолчанию.

Модуль 7. Подробнее о типах данных и операторах

1. `static int test = 100;`
2. Справедливо. Описатель **volatile** сообщает компилятору, что переменная может быть изменена какими-либо событиями вне программы.
3. Чтобы в многофайловом проекте один файл узнал о глобальных переменных, объявленных в другом файле, следует использовать описатель **extern**.
4. Наиболее важным свойством статической локальной переменной является то, что она сохраняет свое значение между вызовами функции.
- 5.

```

// Использование статической переменной
// для подсчета числа вызовов функции.

```

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

int counter();

int main()
{
    int result;

    for(int i=0; i<10; i++)
        result = counter();
    cout << "Функция вызывалась " <<
        result << " раз." << "\n";

    return 0;
}

int counter()
{
    static count = 0;

    count++;

    return count;
}
```

6. Максимальный выигрыш в производительности получится при объявлении регистровой переменной *x*, затем *y*, затем *z*. Так будет потому, что обращение к *x* осуществляется наиболее часто (поскольку она входит в цикл), реже осуществляется обращение к *y*, а *z* используется только когда цикл инициализируется.

7. *&* является побитовым оператором, который воздействует на индивидуальные биты внутри значения. *&&* является логическим оператором, который воздействует на значения *true/false*.

8. Указанное предложение умножает текущее значение *x* на 10 и присваивает результат той же переменной *x*. Предложение равносильно следующему:

```
x = x * 10;
```

9.

```
// Использование циклических сдвигов для шифрования сообщения.
```

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

```
using namespace std;

unsigned char rrotate(unsigned char val, int n);
unsigned char lrotate(unsigned char val, int n);
void show_binary(unsigned int u);

int main()
{
    char msg[] = "This is a test.";
    char *key = "xanadu";
    int klen = strlen(key);
    int rotnum;

    cout << "Исходное сообщение: " << msg << "\n";

    // Шифруем сообщение циклическими сдвигами влево.
    for(int i = 0 ; i < strlen(msg); i++) {
        /* Сдвигаем влево каждую букву на число битов,
           определенное с помощью ключа. */
        rotnum = key[i%klen] % 8;
        msg[i] = lrotate(msg[i], rotnum);
    }

    cout << "Зашифрованное сообщение: " << msg << "\n";

    // Расшифровываем сообщение циклическими сдвигами вправо.
    for(int i = 0 ; i < strlen(msg); i++) {
        /* Сдвигаем вправо каждую букву на число битов,
           определенное с помощью ключа. */
        rotnum = key[i%klen] % 8;
        msg[i] = rrotate(msg[i], rotnum);
    }

    cout << "Расшифрованное сообщение: " << msg << "\n";

    return 0;
}

// Циклический сдвиг байта влево на n позиций.
unsigned char lrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    for(int i=0; i < n; i++) {
```



```
t = t << 1;

/* Если бит выпадает, это будет бит 8
   целочисленной переменной t. Если это так,
   поместим этот бит с правой стороны. */
if(t & 256)
    t = t | 1; // поместим 1 с правой стороны
}

return t; // вернем младшие 8 бит.
}

// Циклический сдвиг байта вправо на n позиций.
unsigned char rrotate(unsigned char val, int n)
{
    unsigned int t;

    t = val;

    // Прежде всего, сдвинем значение на 8 бит влево.
    t = t << 8;
    for(int i=0; i < n; i++) {
        t = t >> 1;
        /* Если бит выпадает, это будет бит 7
           целочисленной переменной t. Если это так,
           поместим этот бит с левой стороны. */
        if(t & 128)
            t = t | 32768; // поместим 1 с правой стороны
    }

    /* Теперь сдвинем результат назад
       в младшие 8 бит переменной t. */
    t = t >> 8;

    return t;
}

// Выведем биты байта.
void show_binary(unsigned int u)
{
    int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";
    cout << "\n";
}
```

Модуль 8. Классы и объекты

1. Класс представляет собой логическую конструкцию, которая определяет форму объекта. Объект является экземпляром класса. Таким образом, объект физически существует в памяти.

2. Для определения класса используется ключевое слово **class**.

3. Каждый объект содержит собственную копию переменных-членов класса.

4.

```
class Test {
    int count;
    int max;
    // ...
}
```

5. Имя конструктора совпадает с именем класса. Имя деструктора также совпадает с именем класса, однако предваряется символом `~`.

6. Ниже приведены три способа создания объекта, при которых переменная `i` инициализируется числом 10:

```
Sample ob(10);
Sample ob = 10;
Sample ob = Sample(10);
```

7. Если функция-член объявлена внутри объявления класса, она, если возможно, делается встроенной.

8.

```
// Создание класса Triangle.
#include <iostream>
#include <cmath>
using namespace std;

class Triangle {
    double height;
    double base;
public:
    Triangle(double h, double b) {
        height = h;
        base = b;
    }

    double hypot() {
        return sqrt(height*height + base*base);
    }
}
```

```
double area() {
    return base * height / 2.0;
}

};

int main()
{
    Triangle t1(3.0, 4.0);
    Triangle t2(4.5, 6.75);

    cout << "Гипотенуза треугольника t1: " <<
        t1.hypot() << "\n";
    cout << "Площадь треугольника t1: " <<
        t1.area() << "\n";
    cout << "Гипотенуза треугольника t2: " <<
        t2.hypot() << "\n";
    cout << "Площадь треугольника t2: " <<
        t2.area() << "\n";

    return 0;
}
```

9.

```
/*
    Расширенный Проект 8-1

    Добавление к классу Help идентификатора пользователя.
*/

#include <iostream>
using namespace std;

// Класс, инкапсулирующий справочную систему.
class Help {
    int userID;
public:
    Help(int id) { userID = id; }

    ~Help() { cout << "Справочник завершается для #" <<
        userID << ".\n"; }

    int getID() { return userID; }
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};
```

```
// Вывод справочной информации.
void Help::helpon(char what) {
    switch(what) {
        case '1':
            cout << "Предложение if:\n\n";
            cout << "if(условие) предложение;\n";
            cout << "else предложение;\n";
            break;
        case '2':
            cout << "Предложение switch:\n\n";
            cout << "switch(выражение) {\n";
            cout << " case константа:\n";
            cout << " последовательность предложений\n";
            cout << " break;\n";
            cout << " // ... \n";
            cout << " }\n";
            break;
        case '3':
            cout << "Цикл for:\n\n";
            cout << "for(инициализация; условие; приращение) ";
            cout << " предложение;\n";
            break;
        case '4':
            cout << "Цикл while:\n\n";
            cout << "while(условие) предложение;\n";
            break;
        case '5':
            cout << "Цикл do-while:\n\n";
            cout << "do {\n";
            cout << " предложение;\n";
            cout << "} while (условие);\n";
            break;
        case '6':
            cout << "Предложение break:\n\n";
            cout << "break;\n";
            break;
        case '7':
            cout << "Предложение continue:\n\n";
            cout << "continue;\n";
            break;
        case '8':
            cout << "Предложение goto:\n\n";
            cout << "goto метка;\n";
            break;
    }
    cout << "\n";
}
```

```
}

// Вывод на экран меню справочной системы.
void Help::showmenu() {
    cout << "Справка по:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << " 3. for\n";
    cout << " 4. while\n";
    cout << " 5. do-while\n";
    cout << " 6. break\n";
    cout << " 7. continue\n";
    cout << " 8. goto\n";
    cout << "Выберите один из пунктов (q для завершения): ";
}

// Возвращает true, если выбор допустим.
bool Help::isvalid(char ch) {
    if(ch < '1' || ch > '8' && ch != 'q')
        return false;
    else
        return true;
}

int main()
{
    char choice;
    Help hlpob(27); // создание экземпляра класса Help.
    cout << "ID пользователя: " << hlpob.getID() <<
        ".\n";
    // Используем объект Help для вывода информации.
    for(;;) {
        do {
            hlpob.showmenu();
            cin >> choice;
        } while(!hlpob.isvalid(choice));

        if(choice == 'q') break;
        cout << "\n";

        hlpob.helpon(choice);
    }

    return 0;
}
```

Модуль 9. Подробнее о классах

1. Конструктор копий создает копию объекта. Он вызывается, когда один объект инициализирует другой. Вот его общая форма:

```
имя-класса (const имя-класса &объект) {
    //тело конструктора
}
```

2. Когда объект возвращается функцией, создается временный объект для использования его в качестве возвращаемого значения. После того, как значение возвращено, этот объект уничтожается деструктором объекта.

3.

```
int sum( ) {
    return this.i + this.j;
}
```

4. Структура — это класс, в котором все члены по умолчанию объявляются открытыми. Объединение — это класс, в котором все данные-члены располагаются в одной и той же памяти. Члены объединения также объявляются открытыми по умолчанию.

5. *this относится к объекту, для которого вызвана эта функция.

6. Дружественная функция — это функция-не член класса, которой предоставлен доступ к закрытым членам класса, по отношению к которому она является другом.

7.

```
тип имя-класса::operator#(тип операнд-2)
{
    // левый операнд передается посредством "this"
}
```

8. Для того чтобы могли выполняться операции между классовым и встроенным типами, вы должны использовать две дружественные операторные функции, одну с классовым типом в качестве первого параметра, а другую со встроенным типом в качестве первого параметра.

9. Нет, оператор ? не может быть перегружен. Нет, вы не можете изменить относительный приоритет оператора

10.

```
// Определим, является ли данное множество подмножеством другого.
bool Set::operator <(Set ob2) {
```

```
if(len > ob2.len) return false; // в ob1 больше элементов

for(int i=0; i < len; i++)
    if(ob2.find(members[i]) == -1) return false;
return true;
}

// Определим, является ли данное множество надмножеством другого.
bool Set::operator >(Set ob2) {
    if(len < ob2.len) return false; // в ob1 меньше элементов

    for(int i=0; i < ob2.len; i++)
        if(find(ob2.members[i]) == -1) return false;
    return true;
}
```

11.

```
// Установим пересечение.
Set Set::operator &(Set ob2) {
    Set newset;

    // Добавим элементы, общие для обоих множеств.
    for(int i=0; i < len; i++)
        if(ob2.find(members[i]) != -1) // добавим, если элемент
                                         // есть в обоих множествах
            newset = newset + members[i];

    return newset; // return set
}
```

Модуль 10. Наследование, виртуальные функции и полиморфизм

1. Класс, который наследуется, называется базовым классом. Класс, который наследует, называется производным классом.

2. Базовый класс *не имеет* доступа к членам производных классов, потому что базовый класс ничего не знает о производных классах. Производный класс *имеет* доступ к незакрытым членам базового класса (или классов).

3.

```
// Класс круга.
class Circle : public TwoDShape {
public:
    Circle(double r) : TwoDShape(r) { } // зададим радиус

    double area() {
        return getWidth() * getHeight() * 3.1416;
    }
};
```

4. Для предотвращения доступа производного класса к члену базового класса объявите этот член закрытым в базовом классе.

5. Вот общая форма конструктора, который вызывает конструктор базового класса:

```
производный-класс( ) : базовый-класс( ) { //...
```

6. Конструкторы вызываются в порядке наследования. Поэтому, когда создается объект **Gamma**, конструкторы вызываются в таком порядке: **Alpha**, **Beta**, **Gamma**.

7. К защищенному (**protected**) члену базового класса может обратиться его собственный класс и производные классы.

8. Когда виртуальная функция вызывается посредством указателя базового класса, то тип объекта, на который фактически указывает этот указатель, определяет, какой вариант функции будет вызван.

9. Чистая виртуальная функция — это функция, у которой в ее базовом классе нет тела. Поэтому чистая виртуальная функция должна быть переопределена в производных классах. Абстрактным называется класс, который содержит по крайней мере одну чистую виртуальную функцию.

10. Нет, абстрактный класс нельзя использовать для создания объекта.

11. Чистая виртуальная функция представляет собой родовое определение, которому должны следовать все реализации этой функции. Во фразе “один интерфейс, множество методов” чистая виртуальная функция представляет *интерфейс*, а ее индивидуальные реализации представляют *методы*.

Модуль 11. C++ и система ввода-вывода

1. Предопределенными потоками являются **cin**, **cout**, **cerr** и **clog**.

2. Да, C++ определяет символьные потоки и для 8-битовых, и для “широких” символов.

3. Общая форма перегруженного оператора вставки выглядит следующим образом:

```
ostream &operator<(ostream &stream, тип_класса obj)
{
    // Здесь размещается специфический код класса
    return stream; // возврат stream
}
```

4. **ios::scientific** определяет вывод числовых значений в экспоненциальной форме.

5. Функция **width()** устанавливает ширину поля вывода.

6. Справедливо. Манипулятор ввода-вывода используется внутри выражений ввода-вывода.

7. Вот один из способов открытия файла для ввода текста:

```
ifstream in("test");
if(!in) {
    cout << "Не могу открыть файл.\n";
    return 1;
}
```

8. Вот один из способов открытия файла для вывода текста:

```
ofstream out("test");

f(!out) {
    cout << "Не могу открыть файл.\n";
    return 1;
}
```

9. **ios::binary** указывает, что файл открывается для двоичного, а не текстового ввода-вывода.

10. Справедливо. Когда достигается конец файла, переменная потока становится равной **false**.

11. **while(strm.get(ch)) // ...**

12. Эта задача имеет множество решений. Вот только одно из них:

// Копирование файла.

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

int main(int argc, char *argv[])
{
    char ch;
    if(argc!=3) {
        cout << "Использование: сору <источник> <приемник>\n";
```

```

    return 1;
}

ifstream src(argv[1], ios::in | ios::binary);
if(!src) {
    cout << "Не могу открыть файл-источник.\n";
    return 1;
}

ofstream targ(argv[2], ios::out | ios::binary);
if(!targ) {
    cout << "Не могу открыть файл-приемник.\n";
    return 1;
}

do {
    src.get(ch);
    if(!src.eof()) targ.put(ch);
} while(!src.eof());

src.close();
targ.close();

return 0;
}

```

13. Эта задача имеет множество решений. Вот одно из простых:

// Слияние файлов.

```

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc != 4) {
        cout << "Использование: merge <источник1> <источник2>
                <приемник>\n";
        return 1;
    }

    ifstream src1(argv[1], ios::in | ios::binary);
    if(!src1) {
        cout << "Не могу открыть 1-й файл-источник.\n";
    }
}

```

```
    return 1;
}
ifstream src2(argv[2], ios::in | ios::binary);
if(!src2) {
    cout << "Не могу открыть 2-й файл-источник.\n";
    return 1;
}

ofstream targ(argv[3], ios::out | ios::binary);
if(!targ) {
    cout << "Не могу открыть файл-приемник.\n";
    return 1;
}

// Копируем первый файл-источник.
do {
    src1.get(ch);
    if(!src1.eof()) targ.put(ch);
} while(!src1.eof());

// Копируем второй файл-источник.
do {
    src2.get(ch);
    if(!src2.eof()) targ.put(ch);
} while(!src2.eof());

src1.close();
src2.close();
targ.close();

return 0;
}
```

14. `MyStrm.seekg(300, ios::beg);`

Модуль 12. Исключения, шаблоны и другие дополнительные темы

1. Обработка исключений в C++ основана на трех ключевых словах: **try**, **catch** и **throw**. В самых общих чертах обработка исключений организуется следующим образом. Программные предложения, которые

вы хотите контролировать на предмет исключений, содержатся в блоке **try**. Если внутри блока **try** возникает исключение (т. е. ошибка), выбрасывается исключение (с помощью **throw**). Исключение ловится с помощью **catch** и обрабатывается.

2. Если вы хотите ловить исключения как базового, так и производных классов, в списке **catch** сначала должны идти производные классы, а за ними — базовый.

3. `void func() throw(MyExcept)`

4. Вот один из способов добавления исключения к классу **Queue**. Это только одно из многих возможных решений:

```
/*
    Добавление исключения в Проект 12-1

    Шаблонный класс очереди.
*/

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

// Это исключение, выбрасываемое Queue в случае ошибки.
class QExcept {
public:
    char msg[80];
};

const int maxQsize = 100;

// Далее создается родовой класс очереди.
template <class QType> class Queue {
    QType q[maxQsize]; // массив для хранения очереди
    int size; // максимальное число элементов,
                // которые могут находиться в очереди
    int putloc, getloc; // индексы "положить" и "взять"
    QExcept Qerr; // добавим поле исключения
public:

// Сконструируем очередь конкретной длины.
Queue(int len) {
    // Размер очереди должен быть меньше max и положителен.
    if(len > maxQsize) len = maxQsize;
    else if(len <= 0) len = 1;

    size = len;
    putloc = getloc = 0;
}
```

```
// Поместим данное в очередь.
void put(QType data) {
    if(putloc == size) {
        strcpy(Qerr.msg, "Очередь полна.\n");
        throw Qerr;
    }

    putloc++;
    q[putloc] = data;
}

// Извлечем данное из очереди.
QType get() {
    if(getloc == putloc) {
        strcpy(Qerr.msg, "Очередь пуста.\n");
        throw Qerr;
    }

    getloc++;
    return q[getloc];
}
};

// Демонстрация родового класса Queue.
int main()
{
    // заметьте, что iQa имеет только 2 элемента
    Queue<int> iQa(2), iQb(10);

    try {
        iQa.put(1);
        iQa.put(2);
        iQa.put(3); // здесь будет переполнение!

        iQb.put(10);
        iQb.put(20);
        iQb.put(30);

        cout << "Содержимое целочисленной очереди iQa: ";
        for(int i=0; i < 3; i++) // здесь будет антипереполнение!
            cout << iQa.get() << " ";
        cout << endl;

        cout << " Содержимое целочисленной очереди iQb: ";
        for(int i=0; i < 3; i++)
            cout << iQb.get() << " ";
        cout << endl;
    }
```

```

    Queue<double> dQa(10), dQb(10); // создадим две очереди
                                   // double

    dQa.put(1.01);
    dQa.put(2.02);
    dQa.put(3.03);

    dQb.put(10.01);
    dQb.put(20.02);
    dQb.put(30.03);

    cout << "Содержимое очереди double dQa: ";
    for(int i=0; i < 3; i++)
        cout << dQa.get() << " ";
    cout << endl;

    cout << "Содержимое очереди double dQb: ";
    for(int i=0; i < 3; i++)
        cout << dQb.get() << " ";
    cout << endl;

} catch(QExcpt exc) {
    cout << exc.msg;
}

return 0;
}

```

5. Родовая функция определяет обобщенную форму подпрограммы без указания типов данных, с которыми она работает. Родовая функция создается с помощью ключевого слова **template**.

6. Вот один из способов преобразования **quicksort()** и **qs()** в родовые функции:

```

// Родовая функция Quicksort.

#include <iostream>
#include <cstring>

using namespace std;

// Зададим вызов фактической функции упорядочения.
template <class X> void quicksort(X *items, int len)
{
    qs(items, 0, len-1);
}

// Родовой вариант Quicksort.
template <class X> void qs(X *items, int left, int right)

```

```
{
    int i, j;
    X x, y;

    i = left; j = right;
    x = items[( left+right) / 2 ];

    do {
        while((items[i] < x) && (i < right)) i++;
        while((x < items[j]) && (j > left)) j--;

        if(i <= j) {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}

int main() {

    // упорядочение символов
    char str[] = "jfmckldoelazlkper";
    int i;

    cout << "Исходный массив: " << str << "\n";

    quicksort(str, strlen(str));

    cout << "Упорядоченный массив: " << str << "\n";

    // упорядочение целых
    int nums[] = { 4, 3, 7, 5, 9, 8, 1, 3, 5, 4 };

    cout << "Исходный массив: ";
    for(int i=0; i < 10; i++)
        cout << nums[i] << " ";
    cout << endl;
    quicksort(nums, 10);
    cout << "Упорядоченный массив: ";
    for(int i=0; i < 10; i++)
        cout << nums[i] << " ";
    cout << endl;
```

```
    return 0;
}
```

7. Вот один из способов поместить в **Queue** объекты **Sample**:

```
/*
    Использование Проекта 12-1 для хранения объектов Sample.

    Шаблонный класс очереди.
*/

#include <iostream>
using namespace std;

class Sample {
    int id;
public:
    Sample() { id = 0; }
    Sample(int x) { id = x; }
    void show() { cout << id << endl; }
};

const int maxQsize = 100;

// Далее создается родовой класс очереди.
template <class QType> class Queue {
    QType q[maxQsize]; // массив для хранения очереди
    int size; // максимальное число элементов,
                // которые могут находиться в очереди
    int putloc, getloc; // индексы "положить" и "взять"
public:

    // Сконструируем очередь конкретной длины.
    Queue(int len) {
        // Размер очереди должен быть меньше max и положителен.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Поместим данное в очередь.
    void put(QType data) {
        if(putloc == size) {
            cout << " - Очередь полна.\n";
            return;
        }
    }
};
```



```

    putloc++;
    q[putloc] = data;
}

// Извлечем данное из очереди.
QType get() {
    if(getloc == putloc) {
        cout << " -- Очередь пуста.\n";
        return 0;
    }

    getloc++;
    return q[getloc];
}
};

// Демонстрация родового класса Queue.
int main()
{
    Queue<Sample> sampQ(3);

    Sample o1(1), o2(2), o3(3);

    sampQ.put(o1);
    sampQ.put(o2);
    sampQ.put(o3);

    cout << "Содержимое sampQ:\n";
    for(int i=0; i < 3; i++)
        sampQ.get().show();
    cout << endl;

    return 0;
}

```

8. Динамическое выделение памяти для объектов **Sample**:

```

/*
    Использование Проекта 12-1 для хранения объектов Sample.
    Динамически выделяем память под объекты Sample.
    Шаблонный класс очереди.
*/
#include <iostream>
using namespace std;

class Sample {
    int id;

```

```

public:
    Sample() { id = 0; }
    Sample(int x) { id = x; }
    void show() { cout << id << endl; }
};

const int maxQsize = 100;

// Далее создается родовой класс очереди.
template <class QType> class Queue {
    QType q[maxQsize]; // массив для хранения очереди
    int size; // максимальное число элементов,
                // которые могут находиться в очереди
    int putloc, getloc; // индексы "положить" и "взять"
public:

    // Сконструируем очередь конкретной длины.
    Queue(int len) {
        // Размер очереди должен быть меньше max и положителен.
        if(len > maxQsize) len = maxQsize;
        else if(len <= 0) len = 1;

        size = len;
        putloc = getloc = 0;
    }

    // Поместим данное в очередь.
    void put(QType data) {
        if(putloc == size) {
            cout << " - Очередь полна.\n";
            return;
        }

        putloc++;
        q[putloc] = data;
    }

    // Извлечем данное из очереди.
    QType get() {
        if(getloc == putloc) {
            cout << " -- Очередь пуста.\n";
            return 0;
        }

        getloc++;
        return q[getloc];
    }
}

```

```
};

// Демонстрация родового класса Queue.
int main()
{
    Queue<Sample> sampQ(3);

    Sample *p1, *p2, *p3;

    p1 = new Sample(1);
    p2 = new Sample(2);
    p3 = new Sample(3);

    sampQ.put(*p1);
    sampQ.put(*p2);
    sampQ.put(*p3);

    cout << "Содержимое sampQ:\n";
    for(int i=0; i < 3; i++)
        sampQ.get().show();
    cout << endl;

    delete(p1);
    delete(p2);
    delete(p3);

    return 0;
}
```

9. Для объявления пространства имен под названием **RobotMotion** следует использовать такую конструкцию:

```
namespace RobotMotion {

// ...
}
```

10. Стандартная библиотека C++ содержится в пространстве имен **std**.

11. Нет, статическая функция-член не имеет доступа к нестатическим данным класса.

12. Для получения типа объекта во время выполнения программы следует использовать оператор **typeid**.

13. Для определения допустимости полиморфного приведения во время выполнения следует использовать **dynamic_cast**.

14. **const_cast** отменяет в операции приведения **const** и **volatile**.

Приложение В

Препроцессор

Препроцессор является частью компилятора, которая выполняет различные манипуляции с текстом вашей программы еще до того, как начнется фактическая трансляция вашего исходного кода в объектный код. Вы можете передавать препроцессору команды манипуляции с текстом. Эти команды носят название *директив препроцессора*, и хотя они формально не являются частью языка C++, они расширяют возможности среды программирования.

Препроцессор является в какой-то степени пережитком, оставшимся от C; для C++ он не так важен, как для C. К тому же, некоторые средства препроцессора были замещены более новыми и лучшими элементами языка C++. Однако поскольку многие программисты все еще используют препроцессор, и поскольку он все еще является частью среды языка C++, мы кратко обсудим его в этом приложении.

Препроцессор C++ содержит следующие директивы:

| | | |
|---------|--------|----------|
| #define | #error | #include |
| #if | #else | #elif |
| #endif | #ifdef | #ifndef |
| #undef | #line | #pragma |

Мы видим, что все директивы препроцессора начинаются со знака #. Рассмотрим каждую из них в отдельности.

#define

Директива **#define** используется для определения идентификатора, а также символьной последовательности, которой будет заменяться этот идентификатор каждый раз, когда он будет встречаться в тексте программы. Идентификатор носит название *имени макроса*, а процесс замены называется *макроподстановкой*. Общая форма этой директивы такова:

#define имя-макроса символьная-последовательность

Обратите внимание на отсутствие точки с запятой в конце этого предложения. Между идентификатором и началом символьной последовательности может быть любое число пробелов, но после того, как символьная последовательность началась, она может закончиться только переходом на следующую строку.

Если, например, вы хотите использовать слово “UP” для обозначения числа 1, а слово “DOWN” для обозначения числа 0, вам надо объявить с помощью директивы **#define** два таких макроса:

```
#define UP 1
#define DOWN 0
```

Эти предложения заставят компилятор подставлять 1 или 0 всякий раз, когда в вашем исходном файле встретятся имена UP или DOWN. Например, следующая строка выведет на экран 1 0 2:

```
cout << UP << ' ' << DOWN << ' ' << UP + UP;
```

Важно отдавать себе отчет в том, что макроподстановка представляет собой просто замену идентификатора соответствующей символьной строкой. Так, если вы захотите определить стандартное сообщение, вы можете написать что-нибудь вроде этого:

```
#define GETFILE "Введите имя файла"
// ...
cout << GETFILE;
```

C++, встретив имя **GETFILE**, подставит вместо него строку “Введите имя файла”. Для компилятора предложение **cout** будет фактически выглядеть так:

```
cout << "Введите имя файла";
```

Если имя макроса встречается внутри текстовой строки в кавычках, то макроподстановка не производится. Например,

```
#define GETFILE "Введите имя файла"
// ...
cout << "GETFILE представляет собой имя макроса\n";
```

не выведет

Введите имя файла представляет собой имя макроса

В действительности на экран будет выведено следующее:

GETFILE представляет собой имя макроса

Если символьная строка не умещается на одной строке текста, вы можете перенести продолжение на следующую строку, поместив знак обратной косой черты в конце строки, как это показано ниже:

```
#define LONG_STRING "это очень, очень длинная строка, \
которая используется здесь в качестве примера"
```

Программисты на C++ обычно используют для имен макросов прописные буквы. Это соглашение помогает читающему программу сразу понять, где будут выполняться макроподстановки. Также имеет смысл разместить все директивы **#define** в начале файла или даже выделить их в отдельный “включаемый” файл, а не разбрасывать их по тексту программы.

Последнее замечание: C++ предоставляет лучший способ для определения констант, чем использование директивы **#define**. Однако многие программисты пришли к C++ от C, где для определения констант используется именно **#define**. Так что вы будете часто сталкиваться с этой директивой и в программах на C++.

Макросы, подобные функциям

Директива **#define** имеет еще одно свойство: имя макроса может сопровождаться аргументами. Каждый раз, когда в тексте программы встречается имя макроса, связанные с ним аргументы заменяются фактическими аргументами, указанными в макровывозе. В результате создается макрос, *подобный функции*. Вот пример такого макроса:

```
// Использование макроса, подобного функции.
```

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main()
{
    int x, y;

    x = 10;
    y = 20;
    cout << "Меньшее значение: " << MIN(x, y);

    return 0;
}
```

Когда эта программа компилируется, в нее подставляется выражение, определенное макросом **MIN(a,b)**, но только в качестве операндов этого выражения будут использованы **x** и **y**. Таким образом, предложение **cout** после подстановки будет выглядеть таким образом:

```
cout << " Меньшее значение: " << ((x)<(y)) ? x : y;
```

В сущности, макрос, подобный функции, представляет собой способ определить встроенную (inline) функцию, код которой не вызывается, а расширяется внутри программы.

Излишние, на первый взгляд, круглые скобки, окружающие определение макроса **MIN**, в действительности необходимы, потому что они обеспечивают правильное вычисление подставляемого выражения с учетом относительных приоритетов операторов. Фактически все макроопре-

деления с аргументами должны содержать в дополнительные круглые скобки. В противном случае вы можете получить весьма удивительные результаты. Например, рассмотрим короткую программу, которая с помощью макроса определяет, является ли значение четным или нечетным:

// Эта программа даст неверный ответ.

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

#define EVEN(a) a%2==0 ? 1 : 0

int main()
{
    if(EVEN(9+1)) cout << "четно";
    else cout << "нечетно";

    return 0;
}
```

Эта программа даст неверный ответ, что будет понятно, если рассмотреть процедуру макrorасширения. При компиляции выражение **EVEN(9+1)** будет расширено в

$9+1\%2==0 ? 1 : 0$

Как вы, возможно, помните, оператор деления по модулю % имеет более высокий относительный приоритет, чем оператор плюс. Это значит, что операция % будет выполнена не над $9+1$, а над 1, а ее результат будет прибавлен к 9, что (конечно) не равно 0. Для того чтобы избавиться от неправильной замены, необходимо заключить в круглые скобки параметр **a** в макроопределении **EVEN**, как это показано в исправленном варианте программы:

// Теперь эта программа будет работать правильно.

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

#define EVEN(a) (a)%2==0 ? 1 : 0

int main()
{
    if(EVEN(9+1)) cout << "четно";
    else cout << "нечетно";

    return 0;
}
```


Теперь выражение $9+1$ вычисляется до операции деления по модулю. Вообще полезно всегда заключать параметры макросов в круглые скобки, чтобы избежать неприятностей вроде только что описанной.

Использование макросов вместо настоящих функций имеет одно преимущество: поскольку макрорасширение включается прямо в текст программы, не возникает никаких издержек, связанных с вызовами функций, поэтому скорость программы возрастает. Однако за это возрастание скорости приходится платить увеличением размеров программы, связанным с дублированием программных строк.

Хотя макросы можно часто увидеть в программах на C++, появление описателя **inline**, выполняющего ту же задачу, но лучше и надежнее, сделало макросы, подобные функциям, абсолютно излишним средством. (Вспомним, что **inline** заставляет компилятор включать текст функции прямо в программный код вместо организации процедуры ее вызова.) К тому же встроенные функции не требуют дополнительных скобок, обязательных для большинства макросов. Однако макросы, подобные функциям, несомненно будут еще долго использоваться в C++-программах, потому что многие программисты, начинавшие с языка C, продолжают пользоваться ими по привычке.

#error

Директива **#error**, встретившись в тексте программы, заставляет компилятор прекратить компиляцию. Эта директива используется в основном в целях отладки. Общая ее форма такова:

#error сообщение-об-ошибке

Обратите внимание на отсутствие кавычек вокруг строки с сообщением. Когда компилятор встречает эту директиву, он выводит сообщение об ошибке и другую информацию и прекращает компиляцию программы. Какая именно информация будет выведена, зависит от реализации вашей среды программирования. (Вы можете поэкспериментировать со своим компилятором и посмотреть, что будет выведено.)

#include

Директива препроцессора **#include** требует, чтобы компилятор подключил к тексту программы, содержащей директиву **#include**, либо стандартный заголовок, либо другой файл с исходным текстом. Имя стандартного заголовка заключается в угловые скобки, как это мы делали во всех программах этой книги. Например,

```
#include <fstream>
```

подключает стандартный заголовок для файловых операций ввода-вывода.

При подключении другого исходного файла его имя должно заключаться в двойные кавычки или в угловые скобки. Например, обе приведенных ниже строки

```
#include <sample.h>
#include "sample.h"
```

требуют от компилятора чтения и компиляции файла **sample.h**.

При подключении файла с помощью директивы **#include** окружающие имя файла знаки — кавычки или угловые скобки — определяют способ поиска указанного файла. Если имя файла заключено в угловые скобки, компилятор сначала ищет его в одном или нескольких каталогах, определенных заранее в среде программирования. Если имя файла заключено в кавычки, компилятор ищет его в другом каталоге, тоже определенном в среде программирования. Обычно этим каталогом назначается текущий рабочий каталог. Если файл в этом каталоге не найден, дальнейший поиск производится так же, как если бы имя файла было заключено в угловые скобки. Поскольку пути поиска зависят от реализации среды программирования, вам придется свериться с руководством пользователя для вашего компилятора.

Директивы условной компиляции

Среди директив препроцессора имеются несколько директив, которые позволяют вам выборочно компилировать определенные части вашего программного кода. Этот процесс, называемый *условной компиляцией*, широко используется программистскими фирмами, разрабатывающими и поддерживающими различные специализированные варианты одной программы.

#if, #else, #elif и #endif

Общая идея, воплощенная в директиве **#if**, заключается в том, что если константное выражение, следующее за **#if**, истинно, тогда код между ним и директивой **#endif** компилируется; в противном случае этот код опускается. Директива **#endif** используется для того, чтобы отметить конец блока **#if**.

Общая форма **#if** такова:

```
#if константное-выражение  
    последовательность предложений  
#endif
```

Если константное выражение справедливо, последовательность предложений будет компилироваться; в противном случае эти строки пропустятся. Например:

// 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;  
}
```

Эта программа выведет на экран сообщение о нехватке памяти, потому что, как это определено в программе, **MAX** больше 10. Этот пример иллюстрирует важный момент. Выражение, следующее за **#if**, *вычисляется во время компиляции*. Поэтому оно может содержать только константы или предварительно определенные идентификаторы. Переменные здесь использовать нельзя.

Директива **#else** действует практически так же, как и предложение **else**, являющееся элементом языка C++: она определяет альтернативу, если директива **#if** дает отрицательный результат. Предыдущий пример можно расширить, включив в него директиву **#else**, как это показано ниже:

// Простой 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 << "Наличной памяти достаточно.\n";
  #endif

  // ...
  return 0;
}
```

В этой программе константа **MAX** определена как 6, т. е. меньше 10, поэтому часть кода после **#if** не компилируется, но зато компилируется альтернативный блок, стоящий после **#else**. В результате на экран выводится сообщение “Наличной памяти достаточно”.

Заметьте, что директива **#endif** завершает как блок **#if**, так и (в данном случае) всю конструкцию **#if/#else**. Как в том, так и в другом случае использование **#endif** необходимо, причем с каждой директивой **#if** может быть связана только одна директива **#endif**.

Директива **#elif** обозначает “else if” и используется для образования цепочки if-else-if при наличии нескольких вариантов компиляции. Если выражение истинно, тогда компилируется именно этот программный блок, а остальные выражения **#elif** не анализируются и не компилируются. В противном случае проверяется следующее условие в цепочке. Общая форма все конструкции выглядит таким образом:

```
#if выражение
  последовательность предложений
#elif выражение1
  последовательность предложений
#elif выражение2
  последовательность предложений
#elif выражение3
  последовательность предложений
//...
#elif выражениеN
  последовательность предложений
#endif
```

Приведенный ниже фрагмент использует значение **COMPYLED_BY** для того чтобы определить, кто компилировал данную программу:

```
#define JOHN 0
#define BOB 1
#define TOM 2

#define COMPILED_BY JOHN
```

```
#if COMPILED_BY == JOHN
    char who[] = "Джон";
#elif COMPILED_BY == BOB
    char who[] = "Боб";
#else
    char who[] = "Том";
#endif
```

Директивы **#if** и **#elif** могут быть вложенными. В этом случае **#endif**, **#else** или **#elif** связываются с ближайшим **#if** или **#elif**. Так, приведенный ниже фрагмент вполне правилен:

```
#if COMPILED_BY == BOB
    #if DEBUG == FULL
        int port = 198;
    #elif DEBUG == PARTIAL
        int port = 200;
    #endif
#else
    cout << "Боб должен компилировать с отладочным выводом.\n";
#endif
```

#ifdef и #ifndef

Другой способ условной компиляции использует директивы **#ifdef** и **#ifndef**, которые означают “if defined” (если определено) и “if not defined” (если неопределено) соответственно и относятся к именам макросов.

Общая форма директивы **#ifdef**:

```
#ifdef имя-макроса
    последовательность предложений
#endif
```

Если макрос с указанным именем был предварительно определен в предложении **#define**, последовательность предложений между **#ifdef** и **#endif** компилируется.

Общая форма директивы **#ifndef**:

```
#ifndef имя-макроса
    последовательность предложений
#endif
```

Если макрос с указанным именем не был предварительно определен в предложении **#define**, то блок предложений компилируется. Обе

директивы, и **#ifdef** и **#ifndef**, могут использовать предложения **#else** и **#elif**. Вы также можете вкладывать директивы **#ifdef** и **#ifndef** тем же образом, что и директивы **#if**.

#undef

Директива **#undef** используется для того чтобы отменить указанное ранее определение имени макроса. Общая форма этой директивы такова:

#undef *имя-макроса*

Рассмотрим такой пример:

```
#define TIMEOUT 100
#define WAIT 0
// ...
#undef TIMEOUT
#undef WAIT
```

Здесь **TIMEOUT** и **WAIT** определены до тех пор, пока в тексте программы не встретятся предложения **#undef**. Основное назначение предложений **#undef** — ограничить область действия имени макроса теми частями программы, где оно используется.

Использование defined

Помимо директивы **#ifdef**, имеется еще один способ узнать, определено ли имя макроса. Вы можете использовать директиву **#if** в сочетании с оператором времени компиляции **defined**. Например, для того, чтобы узнать, определен ли макрос **MYFILE**, вы можете использовать любую из этих команд препроцессора:

```
#if defined MYFILE
```

и

```
#ifdef MYFILE
```

Вы также можете предварить ключевое слово **defined** знаком **!**, чтобы обратить условие. Например, следующий фрагмент будет компилироваться, только если **DEBUG** определено:

```
#if !defined DEBUG
cout << "Окончательная версия!\n";
#endif
```

#line

Директива **#line** используется, чтобы изменить содержимое предопределенных макросов `__LINE__` и `__FILE__`. Макрос `__LINE__` содержит номер компилируемой строки, а макрос `__FILE__` содержит имя компилируемого файла. Основная форма директивы **#line** такова:

#line *число* "*имя-файла*"

Здесь *число* представляет собой любое положительное целое, а не обязательный параметр *имя-файла* обозначает любой допустимый идентификатор. Указанное число становится номером текущей строки исходного кода, а указанное имя файла становится именем исходного файла. Директива **#line** используется главным образом в целях отладки и в специальных приложениях.

#pragma

Директива **#pragma** по-разному определяется в конкретных реализациях сред программирования. Она позволяет предавать компилятору различные инструкции, состав которых определил разработчик данного компилятора. Общая форма директивы **#pragma** такова:

#pragma *имя*

Здесь *имя* определяет требуемое действие. Если *имя* не распознается компилятором, тогда директива **#pragma** попросту игнорируется; такая ситуация не является ошибкой.

Чтобы узнать, какие действия можно затребовать с помощью директивы **#pragma**, обратитесь к документации по вашему компилятору. Вы можете найти варианты, которые вам пригодятся при разработке программ. Типичные инструкции, задаваемые директивой **#pragma**, определяют состав выводимых компилятором предупреждений, вид генерируемого кода и имя используемой библиотеки.

Операторы препроцессора # и

C++ поддерживает два оператора препроцессора: `#` и `##`. Эти операторы используются в сочетании с директивой **#define**. Оператор `#`

образует пару кавычек вокруг следующего за ним аргумента. Рассмотрим такую программу:

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

#define mkstr(s) # s

int main()
{
    cout << mkstr(Мне нравится C++);

    return 0;
}
```

Препроцессор C++ преобразует строку

```
cout << mkstr(Мне нравится C++);
```

в такую:

```
cout << "Мне нравится C++";
```

Оператор `##` используется для слияния двух символических обозначений. Вот 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 такую:

```
cout << xy;
```


Если эти операторы показались вам странными, не смущайтесь: в большинстве программ они не нужны и не используются. Операторы `#` и `##` предназначены главным образом для обработки препроцессором некоторых специальных макросов.

Предопределенные макросы

В C++ определены шесть встроенных имен макросов:

```
-- _LINE_ --  
-- _FILE_ --  
-- _DATE_ --  
-- _TIME_ --  
-- _STDC_ --  
-- _cplusplus
```

Макросы `_LINE_` и `_FILE_` уже упоминались в подразделе, описывающем директиву `#line`. Они содержат текущий номер строки и имя файла компилируемой программы.

Макрос `_DATE_` содержит строку в форме *месяц/день/год*, которая представляет собой дату трансляции исходного текста в объектный код.

Макрос `_TIME_` содержит время, когда программа была откомпилирована. Время представлено в виде строки в формате *час:минута:секунда*.

Смысл макроса `_STDC_` зависит от реализации компилятора. Обычно если макрос `_STDC_` определен, компилятор будет воспринимать только стандартный код C/C++, не содержащий никаких нестандартных расширений.

Компилятор, совместимый со стандартным C++ ANSI/ISO, должен определять `_cplusplus` как значение, содержащее по крайней мере 6 цифр. Компилятор, не соответствующий стандарту ANSI/ISO, должен использовать значение с пятью или меньшим числом цифр.

Приложение С

Работа со старым компилятором C++

Если вы используете современный компилятор, у вас не будет проблем с компиляцией программных примеров этой книги. В этом случае вам не нужна информация, приводимая в настоящем приложении. Просто вводите программы в точности в том виде, в каком они приведены в книге. Как уже отмечалось, программы этой книги полностью соответствуют стандарту ANSI/ISO на язык C++ и будут правильно компилироваться почти любым современным компилятором C++, включая Microsoft Visual C++ или Borland C++ Builder.

Если, однако, вы используете компилятор, разработанный несколько лет назад, то при компиляции некоторых программ вы, возможно, будете получать сообщения об ошибках, потому что ваш компилятор не распознает некоторые новейшие средства C++. Не расстраивайтесь — для того, чтобы программные примеры работали с вашим старым компилятором, вам придется внести в них лишь незначительные изменения. Чаще всего различия между старым и новым стилем кода ограничиваются двумя моментами: новым стилем заголовков и предложением **namespace**. Опишем эти различия подробнее.

Как уже объяснялось в Модуле 1, предложение **#include** включает в вашу программу заголовок. В ранних версиях C++ все заголовки представляли собой файлы с расширением **.h**. Например, в программе, написанной в старом стиле, для включения заголовка **iostream** вы используете такое предложение:

```
#include <iostream.h>
```

В результате файл **iostream.h** включался в текст вашей программы. Таким образом, в программе старого стиля для включения заголовка вы указывали имя файла с этим заголовком. Файл этот имел расширение **.h**. Теперь так не делают.

Современный C++ использует другой вид заголовков, появившийся после разработки стандартного C++ ANSI/ISO. Заголовки нового стиля *не* используют имен файлов. Вы просто указываете в программе стандартный идентификатор, который может быть отображен компьютером на файл, но это совсем не обязательно. Заголовки нового стиля представляют собой абстракцию, которая просто гарантирует, что соответствующая информация, требуемая вашей программе, будет в нее включена.

Поскольку заголовки нового стиля не являются именами файлов, они не имеют расширения **.h**. Вы должны указать только имя заголовка в угловых скобках. Вот, например, два заголовка нового стиля, поддерживаемые стандартным C++:

```
<iostream>  
<fstream>
```

Для преобразования этих заголовков в формат старого стиля просто добавьте к их именам расширение **.h**.

Содержимое добавляемого к программе заголовка входит в пространство имен **std**. Как уже говорилось, пространство имен — это просто декларативный район. Его цель заключается в локализации имен идентификаторов, что позволяет избежать конфликтов имен. Старые версии C++ помещали имена библиотечных функций и проч. в глобальное пространство имен, а не пространство имен **std**, как это делают современные компиляторы. Поэтому при работе со старым компилятором нет необходимости в предложении

```
using namespace std;
```

Собственно говоря, старый компилятор даже не будет воспринимать предложение **using**.

Два простых изменения

Если ваш компилятор не поддерживает пространства имен и заголовки нового стиля, то при компиляции первых строк примеров этой книги он будет выдавать одно или несколько сообщений об ошибках. Если это происходит, вам достаточно внести в программы два простых изменения: использовать заголовки старого стиля и удалить предложение **namespace**. Например, строки

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

следует заменить одной строкой

```
#include <iostream.h>
```

Это изменение преобразует современную программу в программу, написанную в старом стиле. Поскольку заголовок старого стиля считывает все свое содержимое в глобальное пространство имен, не возникает необходимости в предложении **namespace**. После внесения указанных изменений программа будет успешно компилироваться старым компилятором.

В некоторых случаях вам придется внести еще одно изменение. C++ наследует некоторые заголовки от C. Язык C не поддерживает заголовки нового стиля. Для того, чтобы обеспечить обратную совместимость, стандартный C++ все еще поддерживает заголовочные файлы C-стиля. Однако C++ определяет также заголовки нового стиля, которые вы можете использовать вместо заголовочных файлов C. В версиях C++ стандартных заголовков C просто прибавляется префикс **c** к имени файла и опускается расширение **.h**. Так, заголовки нового стиля C++ для заголовочного файла **math.h** будут

`<cmath>`. Для файла `string.h` это будет заголовок `<cstring>`. Хотя в настоящее время разрешается включать в программу заголовочные файлы С-стиля, однако стандартный С++ не рекомендует такой подход. Поэтому в этой книге во всех предложениях `#include` используются заголовки С++ нового стиля. Если ваш компилятор не поддерживает форму нового стиля для заголовков С, просто замените их заголовочными файлами старого стиля.

Предметный указатель

+ 78
– 78
* 78
/ 78
% 78
++ 78
-- 78
< 44, 82
<= 44, 82
> 44, 82
>= 44, 82
== 44, 82
!= 44, 82
; 29, 48
: : 318, 397, 542
– 33, 170
– – 170
[] 140, 145, 176
614
614
#define, директива препроцессора 604, 614
#elif, директива препроцессора 611
#else, директива препроцессора 610
#endif, директива препроцессора 609
#error, директива препроцессора 608
#if, директива препроцессора 610
#ifdef, директива препроцессора 612, 613
#ifndef, директива препроцессора 612
#include, директива препроцессора 608, 618
#line, директива препроцессора 614
#pragma, директива препроцессора 614
#undef, директива препроцессора 613
& (оператор указателя) 165, 238, 241, 290
() 92, 190
* 33, 241
* (оператор указателя) 165, 239, 241
, (запятая) 305
. 397
. (оператор-точка) 314, 319
.* 397
.src, расширение имени файла 25

.h, расширение имени файла 618, 619
/ 33
/* */ 28
// 29
? 303, 383, 397
_ _cplusplus, макрос 616
_ _DATE_ _ , макрос 616
_ _FILE_ _ , макрос 614, 616
_ _LINE_ _ , макрос 614, 616
_ _STDC_ _ , макрос 616
_ _TIME_ _ , макрос 616
{ } 29, 30, 47, 48, 189
~ 327
+ 33, 170
++ 170
<<
 оператор вывода 29, 462
<cctype>, заголовок 156
<climits>, заголовок 66
<cmath>, заголовок 68
<cstdio>, заголовок 152
<cstdlib>, заголовок 54, 99, 219
<iomanip>, заголовок 475
<iostream.h>, заголовочный файл 458
<iostream>, заголовок 458, 462
<new>, заголовок 533
<typeinfo>, заголовок 553
= 32, 87
-> 348
>> 35, 36
>>: оператор сдвига вправо 290
<<: оператор сдвига влево 290
 оператор ввода 35, 462, 484

А

abs() 54
adjustfield, флаги формата 470
ANSI/ISO, стандартный C++ 18
argc, параметр 216
argv, параметр 216
ASCII, набор символов 65, 120
asm 560

auto 205, 276

B

B (язык программирования) 15

bad() 501

bad_alloc, исключение 533

bad_typeid, исключение 556

basic_ios, класс 461

basic_ostream, поток 461

basic_istream, класс 461

basic_ostream, поток 461

basic_streambuf, класс 461

BCPL (язык программирования) 15

before() 554

bool, тип данного 68, 82

 преобразование в него и из него 90

break, предложение 105, 128

C

C# 14, 19

C++ 14

 и C# 14, 19

 и Java 14, 19

 и Windows 25, 27

 и объектно-ориентированное про-
граммирование 16, 17, 20

 история имени 81

 история развития 14

 прописные и строчные буквы 56

case 105

catch 504, 516

 группа предложений 510

catch(...) 512

cerr, поток 460

char, тип данного

 и модификаторы типа 65

 как "маленькое" целое 65

cin, поток 35, 36, 151, 460

class, ключевое слово 312, 518

clear() 501

clog, поток 460

close() 482

const, описатель 272, 559

const_cast 559

continue, предложение 130

cout, поток 29, 33, 36, 460

D

dec, флаг формата 469

default, предложение 105

defined, оператор времени компиляции
613

delete, оператор 532

double, тип данного 38, 67

 издержки 40

do-while, цикл 122

 и continue 130

dynamic_cast 557

E

else 98, 101

enum, ключевое слово 285

EOF 490

eof() 491, 501

Esc-последовательности 74

extern, описатель класса памяти 276

F

F, спецификатор 73

fail() 501

false 68, 82

fill() 473

flags() 471

float, тип данного 38, 67

 издержки 40

floatfield, флаги формата 470

flush() 492

fmtflags, перечислимый тип 468

for, цикл 112, 122

 и continue 130

 бесконечный 117

 объявление переменной внутри 118,
127

FORTRAN (язык программирования) 16

free() 534

friend, ключевое слово 367

fstream, класс 480

G

gcount() 489
get() 486, 489
getline() 491
gets() 152
good() 501
goto, предложение 136, 200

H

hex, флаг формата 469

I

if, предложение 98
 объявление переменной внутри 127
if-else-if-цепочка 102, 109
ifstream, класс 480
inline, ключевое слово 335, 336
int
 правило по умолчанию int 230
 тип данного 29, 32, 38
 char как "маленькое" целое 65
 в 16- и 32-разрядных средах 66
 и модификаторы типа 63
internal, флаг формата 469
ios 481, 485, 497, 500
ios, класс 468, 500
 флаги формата 469
ios_base, класс 461
iostate 500
is_open() 482
isalpha() 157
isdigit() 157
islower() 157, 174
ispunct() 157
isspace() 157
isupper() 157, 174

J

Java и C++ 14, 19

L

L, спецификатор 73, 74

left, флаг формата 469
long, модификатор типа 61, 64

M

main() 29, 189, 216
 аргументы командной строки 216
 прототип 190, 216
malloc() 534
Microsoft Foundation Classes (MFC) 447
mutable, ключевое слово 275, 560

N

name() 554
namespace 541, 618
 предложение 619
NET Framework Windows Forms, библиотечка классов 447
new, оператор 532
 выбрасывание исключения 533

O

oct, флаг формата 469
off_type 497
ofstream, класс 480
open() 480
ostream, класс 463
overload, ключевое слово 56

P

Pascal (язык программирования) 15
peek() 492
POD 373
pos_type 499
pow() 92, 93
precision() 473
private, описатель доступа 417, 418, 421
protected, описатель доступа 414, 417, 419, 421
public, описатель доступа 313, 413, 417, 421
put() 486
putback() 492

Q

qsort() 227
Quicksort 227

R

rand() 99
rdstate() 500
read() 485, 487
register, описатель класса памяти 282
reinterpret_cast 560
return, предложение 193
right, флаг формата 469
RTTI 553

S

scientific, флаг формата 469
seekg() 497
seekp() 497
setf() 470
short, модификатор типа 61, 64
showbase, флаг формата 469
showpoint, флаг формата 469
showpos, флаг формата 469
signed, модификатор типа 61, 63, 65
sizeof, оператор времени компиляции 307
skipws, флаг формата 469
sqrt() 67, 113
Standard Template Library (STL) 18
static, описатель класса памяти 278
static_cast 559
std, пространство имен 541, 547, 619
STL (Standard Template Library) 18, 560
strcat() 154
strcmp() 154
strcpy() 153
streamsize 473, 487
string, класс 150
strlen() 154
struct, ключевое слово 372
switch, предложение 104, 130
 вложенное 108
 объявление переменной внутри 127

T

tellg() 499
tellp() 499
template, ключевое слово 518, 522, 524
 синтаксис для специализации 522
 специализация родового класса 526
this, указатель 378
 и дружественные операторные функции 390
 и операторные функции-члены 383, 385
throw 504, 505, 514
 общая форма 505
tolower() 157
toupper() 156, 157
true 68, 82
true и false в C++ 68
try 504
try-catch, общая форма 504
type_info, класс 554
typedef 288
typeid, оператор 553
typename, ключевое слово 518

U

U, спецификатор 73
unitbuf, флаг формата 469
UNIX 15
unsetf() 471
unsigned, модификатор типа 61, 63, 64, 65
uppercase, флаг формата 469
using, предложение 545, 619

V

virtual, ключевое слово 441
void, тип данного 69
volatile, описатель 274, 559

W

wchar_t, тип данного 66
литерал 74

wcin, wcout, wcerr, wlog, потоки 460
while, цикл 120
 и continue 130
width() 473
write() 485, 487

X

XOR

логическая операция 84
битовый оператор 290, 293

A

Аргументы

командной строки 216
передача 234
с инициализацией по умолчанию 261
 и неоднозначность 268
список 53
функции 53
 инициализация по умолчанию 261

B

Базовые классы 410

наследование множественное 436, 439
общая форма наследования 413
ссылки 441
указатели 440, 443, 555
управление доступом 417

Байт-код 20

Библиотека классов 55, 447, 452, 541

Библиотечные функции 55

и заголовки 223

Блок программный 46, 199

Быстрое упорядочение 150

B

Ввод-вывод

C и C++ 485

C++ 458

 библиотеки 458

 потоки 459

 форматированный 468

Виртуальные функции 441

и наследование 444

и полиморфизм 410, 441, 443, 446

иерархический характер 445

чистые 452

Возврат ссылок 243

Восьмеричные литералы 73

Вызов

по значению 234

по ссылке 234

 использование параметра-ссыл-
 ки 238

 использование указателя 236

Выражения 89

преобразования типов 90

условные 100

D

Деструкторы 327

и наследование 437

и параметры 331

Динамическая идентификация типов 553

Динамическое выделение памяти 531

malloc() и free() 534

new и delete 532

инициализация 534

под массивы 535

под массивы объектов 538

под объекты 536

Директива #include 28

Директивы препроцессора 604

#define 604

#elif 611

#else 610

#endif 609

#error 608

#if 609

#ifdef 612

#ifndef 612

#include 608

#line 614

#pragma 614

#undef 613

Дополнение до единицы (~) 290

Дополнительный код 62, 63

Друг класса 367

Дружественные функции 367
и перегрузка операторов 390
перегрузка операторов ввода-вывода 465

З

Завершающий ноль 151
использование 156
Заголовки 28, 222, 618
Заголовок <iostream> 28
Заголовочные файлы 618
Запятая, оператор 114, 304

И

И

логический оператор (&&) 82
побитовый оператор (&) 290
Идентификаторы, правило составления 56
Идентификация типов, динамическая 553
Иерархическая классификация 23
ИЛИ
логический оператор (|) 82
побитовый оператор (|) 290, 292
Инициализация
и объекты 364
переменной 77
Инкапсуляция 21, 200, 313, 321
Исключающее ИЛИ, побитовый оператор (^) 290
Исключения 504
вторичное выбрасывание 515
и коды ошибок 517
Исключительная ситуация 504
Исходный код 25

К

Класс 312
Классы 22
абстрактные 455
вложенные 334
доступ к членам 313
и объединения 375

и структуры 374
объявление 312
объявление, общая форма 420
памяти, описатели 275
полиморфные 441
потокковые 460
члены 312

Ключевые слова, таблица 56
Комментарии 27, 29
Компилятор, C++ 25
старый 618
Компиляция 25
условная 609
Консольный ввод 35
Консольный вывод 29
Константы 72
строковые 177
Конструктор 327
копий 358, 361, 364
по умолчанию 358
копий и возврат из функций 363
копий и инициализация 364
копий и присваивание 364
копий, общая форма 364
Конструкторы
и наследование 422, 437
и передача объектов функциям 357
параметрические 329
перегрузка 352
Косвенность 166
вложенная 184, 185
Куча 531, 532, 533

Л

Литералы 72
с плавающей точкой 72
строки 74
суффиксы для числовых значений 73
целочисленные 72

М

Макроподстановка 604
Макросы 604
подобные функциям 606, 608
предопределенные 616

Манипуляторные функции 477
Манипуляторы ввода-вывода 468, 475
Маскитти (Mascitti, Rick) 81
Массивы
 двумерные 145
 динамическое выделение памяти 535
 индексация 141, 173
 инициализация 157
 использование new и delete 535
 многомерные 146, 147, 149
 неопределенной длины 161
 объектов, динамическое выделение
 памяти 538
 одномерные 140
 определение 140
 передача в функции 208, 210
 проверка границ 144
 строк 162
 указателей 181
 упорядочение 147, 149
Математические функции, стандартная
 библиотека 114
Метка 136
Метод 22
Модификаторы типа 61

Н

Наследование 21, 23, 410
 и конструкторы 422
 и многоуровневая иерархия 433
 конструкторы и деструкторы 438
 многоуровневая иерархия 438
 общая форма класса 413
 управление доступом 417
НЕ
 логический оператор (!) 82
 побитовый оператор (~) 290, 294
Независимые ссылочные переменные
 246
Неоднозначность 266

О

Обработка исключений 504
 улавливание всех исключений 512

Объединения 374
 безымянные 377
 и классы 375
 ограничения 376, 377
Объект 313
 передача по ссылке 360
 создание 314
Объектно-ориентированное програм-
 мирование 16, 17, 20, 21, 23, 312
Объектный код 25
Объекты 22, 312
 возврат из функций 361
 динамическое выделение памяти 536
 инициализация 327, 329, 364
 массивы 344
 передача по ссылке 358
 передача функциям 355
 присваивание 353
 ссылки 350
 указатели 348
Объявление переменных 34, 35
 упреждающее 370
Оператор
 ввода (>>)
 перегрузка 466
 вставки (<<) 462
 вывода (<<) 29, 37, 462
 перегрузка 465
 декремента (– –) 46, 79
 и указатели 170
 перегрузка 385, 395
 деления по модулю (%) 79
 извлечения (>>) 462
 инкремента (++) 46, 79
 перегрузка 385, 395
 и указатели 170
 постфикс и префикс 386
 присваивания (=) 33
 проверки на равенство (==) 44
 разрешения видимости (: :) 318, 397,
 542
 тернарный (?) 303, 383
Оператор-запятая (,) 304
Операторные функции-члены 381
Оператор-стрелка (->) 348
Оператор-точка (.) 314, 319, 368, 443
Операторы

арифметические 33, 78
ввода-вывода 29
 перегрузка 462
как приведения типов 91
логические 83
отношения 43, 172
перегрузка 379
 и дружественные функции 390
 ограничения 390
 ограничения 397
 с использованием функций-членов 381
побитовые 289
приведения типа 557
сдвига 295
указателей 165, 167
циклического сдвига 298
Операции присваивания 87
 и указатели 168
 и функции 243
 составные 88
Отступы, практика использования 49
Очередь 339
Ошибки, обработка 30

П

Параметры 188, 191, 205
 индексация 212
 указатель 209, 210, 236
Параметры-ссылки 238, 239, 241
Перегрузка
 операторов 379
 бинарных 383, 390, 392
 декремента 385
 инкремента 385
 инкремента и декремента, постфиксных и префиксных 395
 отношения 395
 присваивания 384
 унарных 383, 385, 390, 394
 функций 247
 и автоматическое преобразование типов 252
 и неоднозначность 266
Передача

 по значению 234
 по ссылке 234
 и объекты 358
 объекта 360
Переменные 31
 глобальные 199, 205
 и extern 276
 недостатки 208
 статические 280
динамическая инициализация 77
инициализация 77
локальные 199, 207
 объявление 202
 статические 278
объявление 76
объявление и определение 277
присваивание значения 32
регистровые 282
сокрытие имен 204
экземпляра 22, 312, 316
Переменные-члены 339
Переносимость 19, 20
Перечисление 284
Перечислимый тип 284
Побитовые операторы 289
Повышение типа 90
Полиморфизм 21, 234, 247, 249, 250, 349, 410, 441, 553
 времени выполнения 446
 и виртуальные функции 410, 441, 443, 446
 и время выполнения 443
Потоки (ввод-вывод) 459
Правила видимости 199
Предложение
 if 43
 вложенное 101
 и булева переменная 69
 return 30, 193
 using 28, 545
 пустое 117
Предложения
 выбора 98
 и точка с запятой 29, 48
 переходов 98
 управления программой 43, 98
 циклов 98

Предупреждающие сообщения, обработка 31
Преобразование типа
 в выражениях 90
 в операциях присваивания 88
 и неоднозначность 266
 и перегруженные функции 252
Препроцессор 604
 директивы 604
 операторы 614
Приведение типа 90
 и указатели 167
 операторы 557
Приоритеты относительные операторов 308
Присваивание
 и указатель `this` 384
 краткое 307
 множественное 306
 объектов 353
 по умолчанию 353
 составное 306
Пробельные символы 152
Программный блок 46
Производные классы 410
 вызов конструктора базового класса 424
 наследование множественное 436
 общая форма наследования 413
 определение 410
Прописные и строчные буквы в C++ 56
Пространства имен 282, 540
 анонимные 547
 безымянные 547
Пространство имен `std` 28
Прототипы 190, 221
Процессор данных 339
Псевдокод 20

Р

Рекурсия 223
Ритчи (Ritchie, Dennis) 15
Ричардс (Richards, Martin) 15
Родовые классы 523
 с несколькими родовыми данными 523, 525
 явные специализации 526

Родовые функции 518
 и перегрузка 523
 с несколькими родовыми типами 520
 явная перегрузка 521

С

Символ новой строки (`\n`) 37, 40
Символы
 ASCII 65, 120
 литералы 72, 75
 преобразование при вводе-выводе 459
 расширенные 120
Символьные константы с обратным слешем 74
Скобки `()` 81, 92, 93, 95
Соккрытие имен 204, 207
Спецификация компоновки, `extern` 277
Ссылка 234
 возврат из функции 243
 вперед 370
 на базовый класс 556
 упреждающая 370
Ссылки 238, 239, 241
 и объекты 358
Ссылочные переменные
 ограничения 247
Стандартная библиотека C++ 55, 541
Стандартная библиотека шаблонов (STL) 18, 560
Стандартный C++ 18
Статические переменные 278
Статические переменные-члены 548
Статические функции-члены 551
Статические члены классов 548
Стек 339, 531, 532
Степанов (Stepanov, Alexander) 18
Страуструп (Stroustrup, Bjarne) 17, 81
Строки
 библиотечные функции обработки 153
 завершающий ноль 150
 как массивы символов 140, 150
 константы 177
 литералы 74
 массивы 162
 нулевые 151

определение 30
 передача в функцию 213
 преобразование в числа 219
 символьные константы 150
 Структурное программирование 15, 21
 Структуры 372
 и классы 372, 374
 и наследование 417, 419

Т

Таблица строк 177, 182
 Текущая позиция 459
 Тернарный оператор 383
 Типы данных 60
 базовые 60
 диапазоны 61, 66
 классы 312, 313
 модификаторы 61
 числовые 40
 Томпсон (Thompson, Ken) 15

У

Указатели
 арифметика 170, 173, 174, 440
 базового класса 443
 базовый тип 164, 167
 в файле 497
 возврат из функции 214
 вызов по ссылке 236
 и массивы 172, 173, 175, 177, 179, 181, 183
 индексация 175
 массивы указателей 181
 на базовый класс 440, 555, 556
 и виртуальность 441
 на объекты 348
 на указатель 184
 нулевые 183
 ошибки 185
 передача в функции 209
 приведение типа 167
 сравнение 172
 стиль объявлений 242
 Указатель

 ввода 497
 вывода 497
 определение 164
 Упорядочение 147, 149
 Quicksort 227
 быстрое 147, 227
 метод Шелла 147
 пузырьковое 147
 Управляющие последовательности 74
 Управляющие предложения 43

Ф

Файл 25, 480
 определение C++ 459
 Файловый ввод-вывод 480
 неформатированный и двоичный 485
 определение состояния 500
 открытие и закрытие 480
 преобразование символов 484
 произвольный доступ 497
 указатели 497
 чтение и запись текстовых файлов 483
 Флаг знака 63
 Флаги формата (ввод-вывод) 469
 Функции
 main() 189
 void 188, 190
 аргументы 191
 возврат 192
 объектов 361
 указателя 214
 возвращаемые значения 195
 встроенные 334
 и макросы, подобные функциям 608
 выбрасывание исключений
 ограничения 513
 определение 514
 вызов 190
 доступа 337, 415
 дружественные 367
 и указатель this 379
 и операции присваивания 53, 243
 и спецификация компоновки 277
 использование в выражениях 197

как программные блоки 200
манипуляции символами 157
общая форма 188
операторные, общая форма 380
определение 22, 52, 188
перегруженные и родовые 523
перегрузка 247
передача
 массива 210
 объектов 355
 строк 213
 указателя 209
порожденные 520
правила видимости 200
правило по умолчанию int 230
прототип 190, 220
рекурсивные 223
специализация 520
тип возвращаемого значения 188, 195
шаблонные 520
Функции-члены 317
 и указатель this 378
 статические 551
Функция operator, общая форма 380

Ц

Циклическое определение 223
Циклы
 do-while 122
 for 45, 112, 113, 115, 117, 119

while 120, 122
бесконечные 117
вложенные 135
выбор формы 122
и break 129
и continue 130
и goto 137

Ч

Члены класса 22, 312
 доступ 313, 337, 367, 413
 статические 548

Ш

Шаблон 517
Шестнадцатеричные литералы 73

Я

Языки
 B 15
 BCPL 15
 C# 14
 C++ 14
 FORTRAN 16
 Java 14
 Pascal 15
 неполиморфные 553
 полиморфные 553

ОГЛАВЛЕНИЕ

| | |
|---|----|
| От переводчика | 6 |
| Предисловие | 8 |
| МОДУЛЬ 1. Основы C++ | 13 |
| Краткая история C++ | 14 |
| Язык C: Заря современной эры программирования | 14 |
| Потребность в C++ | 15 |
| C++ родился | 17 |
| Эволюция C++ | 17 |
| Как C++ соотносится с языками Java и C# | 18 |
| Объектно-ориентированное программирование | 20 |
| Инкапсуляция | 21 |
| Полиморфизм | 22 |
| Наследование | 23 |
| Первая простая программа | 24 |
| Ввод программы | 25 |
| Компиляция программы | 25 |
| Запуск программы | 26 |
| Первый программный пример строка за строкой | 27 |
| Обработка синтаксических ошибок | 30 |
| Вторая простая программа | 31 |
| Использование операторов | 33 |
| Ввод с клавиатуры | 35 |
| Некоторые дополнительные возможности вывода | 37 |
| Еще один тип данных | 38 |
| Проект 1-1: Преобразование футов в метры | 40 |
| Два управляющих предложения | 43 |
| Предложение if | 43 |
| Цикл for | 45 |
| Использование программных блоков | 46 |
| Знак точки с запятой и позиционирование | 48 |
| Практика использования отступов | 49 |
| Проект 1-2: Создание таблицы преобразования футов в метры | 50 |
| Знакомимся с функциями | 52 |
| Библиотеки C++ | 55 |
| Ключевые слова C++ | 55 |

| | |
|---|-----------|
| Идентификаторы | 56 |
| Вопросы для самопроверки | 57 |
| МОДУЛЬ 2. Знакомимся с данными, типами и операторами .. | 59 |
| Почему так важны типы данных | 60 |
| Типы данных C++ | 60 |
| Целые числа | 63 |
| Символы | 65 |
| Типы данных с плавающей точкой | 67 |
| Булев тип данных | 68 |
| Тип void | 69 |
| Проект 2-1: Разговор с Марсом | 70 |
| Литералы | 72 |
| Шестнадцатеричные и восьмеричные литералы | 73 |
| Строковые литералы | 74 |
| Символьные Esc-последовательности | 74 |
| Подробнее о переменных | 76 |
| Инициализация переменной | 76 |
| Динамическая инициализация | 77 |
| Операторы | 78 |
| Арифметические операторы | 78 |
| Инкремент и декремент | 79 |
| Операторы отношения (сравнения) и логические | 81 |
| Проект 2-2: Конструирование логической операции исключающее ИЛИ | 84 |
| Оператор присваивания | 87 |
| Составные присваивания | 87 |
| Преобразование типов в операциях присваивания | 88 |
| Выражения | 89 |
| Преобразование типа в выражениях | 90 |
| Приведение типа | 90 |
| Пробелы и скобки | 92 |
| Проект 2-3: Вычисление регулярных платежей по ссуде | 92 |
| Вопросы для самопроверки | 96 |
| МОДУЛЬ 3. Предложения управления программой | 97 |
| Предложение if | 98 |
| Условные выражения | 100 |
| Вложенные предложения if | 101 |
| Цепочка if-else-if | 102 |
| Предложение switch | 104 |
| Вложенные предложения switch | 108 |
| Проект 3-1: Начинаем строить справочную систему C++ | 109 |
| Цикл for | 112 |
| Некоторые варианты цикла for | 114 |

| | |
|--|------------|
| Опущенные секции | 115 |
| Бесконечный цикл for | 117 |
| Цикл с отсутствующим телом | 117 |
| Объявление переменных управления циклом внутри цикла for | 118 |
| Цикл while | 120 |
| Цикл do-while | 122 |
| Проект 3-2: Усовершенствование справочной системы C++ | 124 |
| Использование break для выхода из цикла | 128 |
| Использование continue | 130 |
| Проект 3-3: Завершаем разработку справочной системы C++ | 131 |
| Вложенные циклы | 135 |
| Использование предложения goto | 136 |
| Вопросы для самопроверки | 137 |
| МОДУЛЬ 4. Массивы, строки и указатели | 139 |
| Одномерные массивы | 140 |
| Границы не проверяются! | 144 |
| Двумерные массивы | 145 |
| Многомерные массивы | 146 |
| Проект 4-1: Упорядочение массива | 147 |
| Строки | 150 |
| Основа техники строк | 150 |
| Ввод строки с клавиатуры | 151 |
| Некоторые библиотечные функции обработки строк | 153 |
| strcpy() | 153 |
| strcat() | 154 |
| strcmp() | 154 |
| strlen() | 154 |
| Пример обработки строк | 155 |
| Использование завершающего нуля | 156 |
| Инициализация массивов | 157 |
| Инициализация массивов неопределенной длины | 160 |
| Массивы строк | 162 |
| Указатели | 164 |
| Что такое указатели? | 164 |
| Операторы указателей | 165 |
| Базовый тип указателя имеет большое значение | 167 |
| Операции присваивания посредством указателя | 168 |
| Выражения с указателями | 169 |
| Арифметика указателей | 170 |
| Сравнение указателей | 172 |
| Указатели и массивы | 172 |
| Индексация указателя | 175 |
| Строковые константы | 177 |

| | |
|---|------------|
| Проект 4-2: Переворачивание строки | 178 |
| Массивы указателей | 181 |
| Соглашение о нулевом указателе | 183 |
| Указатель на указатель | 184 |
| Вопросы для самопроверки | 186 |
| МОДУЛЬ 5. Введение в функции | 187 |
| Основы функций | 188 |
| Общая форма определения функции | 188 |
| Создание функции | 189 |
| Использование аргументов | 190 |
| Использование предложения return | 192 |
| Возвращаемые значения | 195 |
| Использование функций в выражениях | 197 |
| Правила видимости | 199 |
| Локальная область видимости | 199 |
| Глобальная область видимости | 205 |
| Передача в функции указателей и массивов | 208 |
| Передача указателя | 209 |
| Передача массива | 210 |
| Передача строк | 213 |
| Возврат указателей | 214 |
| Функция main() | 216 |
| argc и argv: аргументы функции main() | 216 |
| Передача числовых аргументов командной строки | 219 |
| Прототипы функций | 220 |
| Заголовки содержат прототипы | 222 |
| Рекурсия | 223 |
| Проект 5-1: Быстрое упорядочение | 227 |
| Вопросы для самопроверки | 231 |
| МОДУЛЬ 6. Подробнее о функциях | 233 |
| Два подхода к передаче аргументов | 234 |
| Как C++ передает аргументы | 234 |
| Использование указателя для создания вызова по ссылке | 236 |
| Параметры-ссылки | 238 |
| Возврат ссылок | 243 |
| Независимые ссылочные переменные | 246 |
| Несколько ограничений при использовании ссылочных переменных | 247 |
| Перегрузка функций | 247 |
| Автоматическое преобразование типов и перегрузка | 252 |
| Проект 6-1: Создание перегруженных функций для вывода на экран | 255 |
| Аргументы функций с инициализацией по умолчанию | 261 |
| Аргументы с инициализацией по умолчанию или перегрузка? | 263 |
| Правильное использование аргументов с инициализацией по умолчанию | 265 |

| | |
|---|------------|
| Перегрузка функций и неоднозначность | 266 |
| Вопросы для самопроверки | 269 |
| МОДУЛЬ 7. Подробнее о типах данных и операторах | 271 |
| Описатели const и volatile | 272 |
| const | 272 |
| volatile | 274 |
| Описатели классов памяти | 275 |
| auto | 276 |
| extern | 276 |
| Статические переменные | 278 |
| Регистровые переменные | 282 |
| Перечислимые типы | 284 |
| typedef | 288 |
| Побитовые операторы | 289 |
| Операторы И, ИЛИ, исключающее ИЛИ и НЕ | 290 |
| Операторы сдвига | 295 |
| Проект 7-1: Создание функций циклического побитового сдвига | 298 |
| Оператор ? | 303 |
| Оператор-запятая | 304 |
| Множественное присваивание | 306 |
| Составное присваивание | 306 |
| Использование оператора sizeof | 307 |
| Обзор относительных приоритетов | 308 |
| Вопросы для самопроверки | 309 |
| МОДУЛЬ 8. Классы и объекты | 311 |
| Основы классов | 312 |
| Общая форма класса | 312 |
| Определение класса и создание объектов | 313 |
| Добавление в класс функций-членов | 317 |
| Проект 8-1: Создание класса справочника | 321 |
| Конструкторы и деструкторы | 326 |
| Параметрические конструкторы | 329 |
| Добавление конструктора в класс Vehicle | 331 |
| Другой способ инициализации | 333 |
| Встроенные функции | 334 |
| Создание встроенных функций внутри класса | 336 |
| Проект 8-2: Создание класса очереди | 339 |
| Массивы объектов | 344 |
| Инициализация массивов объектов | 345 |
| Указатели на объекты | 347 |
| Ссылки на объекты | 350 |
| Вопросы для самопроверки | 350 |

| | |
|---|----------------|
| МОДУЛЬ 9. Подробнее о классах | 351 |
| Перегрузка конструкторов | 352 |
| Присваивание объектов | 353 |
| Передача объектов функциям | 355 |
| Конструкторы, деструкторы и передача объектов | 357 |
| Передача объектов по ссылке | 358 |
| Потенциальные проблемы при передаче объектов | 360 |
| Возврат объектов | 361 |
| Создание и использование конструктора копий | 363 |
| Дружественные функции | 367 |
| Структуры и объединения | 372 |
| Структуры | 372 |
| Объединения | 374 |
| Ключевое слово this | 378 |
| Перегрузка операторов | 379 |
| Перегрузка операторов с использованием функций-членов | 381 |
| Другие вопросы | 384 |
| Использование функций-членов для перегрузки унарных операторов | 385 |
| Операторные функции-не члены | 390 |
| Использование дружественной функции для перегрузки унарного оператора | 394 |
| Советы и ограничения при перегрузке операторов | 396 |
| Проект 9-1: Создание класса, определяющего множество | 397 |
| Вопросы для самопроверки | 407 |
| МОДУЛЬ 10. Наследование, виртуальные функции и полиморфизм | 409 |
| Основы наследования | 410 |
| Доступ к членам и наследование | 413 |
| Управление доступом к базовому классу | 417 |
| Использование защищенных членов | 419 |
| Конструкторы и наследование | 422 |
| Вызов конструктора базового класса | 424 |
| Проект 10-1: Расширение класса Vehicle | 429 |
| Создание многоуровневой иерархии классов | 433 |
| Наследование от нескольких базовых классов | 436 |
| Когда выполняются функции конструктора и деструктора | 437 |
| Указатели на производные классы | 439 |
| Ссылки на производные типы | 441 |
| Виртуальные функции и полиморфизм | 441 |
| Основы виртуальных функций | 441 |
| Виртуальные функции наследуются | 444 |
| Зачем нужны виртуальные функции? | 446 |
| Приложение виртуальных функций | 447 |

| | |
|--|------------|
| Чистые виртуальные функции и абстрактные классы | 451 |
| Вопросы для самопроверки | 455 |
| МОДУЛЬ 11. С++ и система ввода-вывода | 457 |
| Старая и новая системы ввода-вывода | 458 |
| Потоки С++ | 459 |
| Предопределенные потоки С++ | 460 |
| Потоковые классы С++ | 460 |
| Перегрузка операторов ввода-вывода | 462 |
| Создание операторных функций вывода | 462 |
| Использование дружественных функций для перегрузки операторов вывода | 464 |
| Перегрузка операторов ввода | 466 |
| Форматированный ввод-вывод | 468 |
| Форматирование с помощью функций-членов ios | 468 |
| Использование манипуляторов ввода-вывода | 475 |
| Создание собственных манипуляторных функций | 477 |
| Файловый ввод-вывод | 480 |
| Открытие и закрытие файла | 480 |
| Чтение и запись текстовых файлов | 483 |
| Неформатированный и двоичный ввод-вывод | 485 |
| Чтение и запись блоков данных | 487 |
| Больше о функция ввода-вывода | 489 |
| Другие варианты get() | 489 |
| getline() | 491 |
| Обнаружение символа EOF | 491 |
| peek() и putback() | 491 |
| flush() | 492 |
| Проект 11-1: Утилита сравнения файлов | 492 |
| Произвольный доступ | 497 |
| Определение состояния ввода-вывода | 500 |
| Вопросы для самопроверки | 501 |
| МОДУЛЬ 12. Исключения, шаблоны и другие дополнительные темы | 503 |
| Обработка исключений | 504 |
| Основы обработки исключений | 504 |
| Использование группы предложений catch | 510 |
| Улавливание всех исключений | 512 |
| Задание исключений, выбрасываемых функцией | 513 |
| Вторичное выбрасывание исключения | 515 |
| Шаблоны | 517 |
| Родовые функции | 518 |
| Функция с двумя родовыми типами | 520 |
| Явная перегрузка родовых функций | 521 |

| | |
|---|------------|
| Родовые классы | 523 |
| Явные специализации класса | 526 |
| Проект 12-1: Создание родового класса очереди | 527 |
| Динамическое выделение памяти | 531 |
| Инициализация выделенной памяти | 534 |
| Выделение памяти под массивы | 535 |
| Выделение памяти под объекты | 536 |
| Пространства имен | 540 |
| Основы использования пространств имен | 541 |
| Предложение using | 545 |
| Безымянные пространства имен | 547 |
| Пространство имен std | 547 |
| Статические члены классов | 548 |
| Статические переменные-члены | 548 |
| Статические функции-члены | 551 |
| Динамическая идентификация типов (RTTI) | 553 |
| Операторы приведения типа | 557 |
| dynamic_cast | 557 |
| const_cast | 559 |
| static_cast | 559 |
| reinterpret_cast | 560 |
| Что дальше? | 560 |
| Вопросы для самопроверки | 561 |
| Приложение А. Ответы на Вопросы для самопроверки | 563 |
| Приложение В. Препроцессор | 603 |
| #define | 604 |
| Макросы, подобные функциям | 606 |
| #error | 608 |
| #include | 608 |
| Директивы условной компиляции | 609 |
| #if, #else, #elif и #endif | 609 |
| #ifdef и #ifndef | 612 |
| #undef | 613 |
| Использование defined | 613 |
| #line | 614 |
| #pragma | 614 |
| Операторы препроцессора # и ## | 614 |
| Предопределенные макросы | 616 |
| Приложение С. Работа со старым компилятором C++ | 617 |
| Два простых изменения | 619 |
| Предметный указатель | 621 |

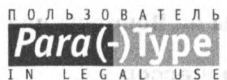
Учебное издание
Серия «Шаг за шагом»

Шилдт Герберт

C++ для начинающих

Перевод *К. Финогенова*
Редактор *К. Финогенов*
Верстка *В. Верхозина*
Дизайн и оформление *О. Будко*

Директор издательства *В. Говорухин*
Главный редактор *Л. Захарова*



Подписано в печать 18.02.2010. Формат 70х100 $\frac{1}{16}$.
Гарнитура Newton. Печать офсетная. Усл. печ. л. 51,85
Тираж 500 экз. Заказ № А-1203.

ЭКОМ Паблишерз

117342, Россия, Москва, ул. Бултерова, д.17а, оф. 105
www.ecom.ru
E-mail: zakaz@ecom.ru



C++

Предлагаемая книга представляет собой самоучитель, который шаг за шагом расскажет о всех основных понятиях языка C++, включая переменные, инструкции управления, функции, типы и массивы данных, классы и объекты. Кроме того вы узнаете о перегрузках, управлении исключениями, наследовании, виртуальных функциях, полиморфизме, вводе/выводе. Рассмотрены и более сложные средства языка, такие как шаблоны и пространства имен. Начинать программировать прямо сейчас – данная книга поможет вам в этой работе.

В книге описаны

- Типы данных и операторы
- Управляющие конструкции
- Классы, объекты и шаблоны
- Интерфейсы, массивы и методы
- Перечисления и структуры
- Перегрузка операторов
- Наследование
- Обработка исключений
- Делегаты, свойства, события
- Индексаторы и атрибуты
- Указатели и полиморфизм
- Многопоточное программирование
- Обобщенные типы
- Обнуляемые типы
- Анонимные методы
- Классы коллекций
- Классы ввода/вывода
- Работа в сети
- Виртуальные функции и многое другое

Исходные коды программ – бесплатно на веб-сайте www.osborne.com

Об авторе

Герберт Шилдт – широко известный в мире эксперт по программированию на языках C++, C# и Java. Автор многочисленных книг в сериях для начинающих (Beginner's) и профессионалов (The Complete Reference), изданных суммарным тиражом более 3 млн. экземпляров.

На русском языке книги этих серий выходят в издательстве ЭКОМ

Информация
о наших новых книгах
на сайте:
www.ecom.ru

The McGraw-Hill Companies

Mc
Graw
Hill

OSBORNE

www.osborne.com

Osborne delivers results!]

ISBN 978-5-9790-0127-2



9 785979 001272 >



OSBORNE

ЭКОМ