ИЗУЧАЕМ ВМЕСТЕ С BHV

Г. Шилдт

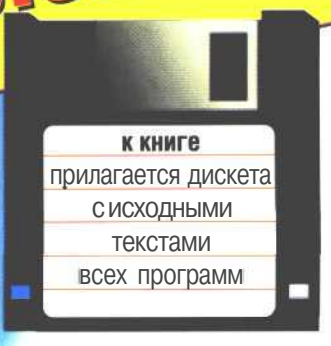
САМОУЧИТЕЛЬ

C++

3-е издание, переработанное и дополненное

- Наследование, полиморфизм, **инкапсуляция**
- Исключительные ситуации
- Шаблоны и контейнеры
- Пространства имен
- **Динамическая** идентификация типа
- Библиотека стандартных шаблонов

БЕСТСЕЛЛЕР!!!



bhv

Herbert Schildt

Teach Yourself C++

Third Edition

Osborne **McGraw-Hill**

Герберт Шилдт

Самоучитель C++

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06

Шилдт Г.

Самоучитель C++: Пер. с англ. — 3-е изд. — СПб.: БХВ-Петербург, 2005. — 688 с.

ISBN 5-7791-0086-1

Необходимость в переработке и дополнении предыдущего издания книги вызвана в первую очередь выходом в свет долгожданного для программистов всего мира единого международного стандарта по C++. Теперь можно быть уверенным, что уже в ближайшем будущем программы на C++ будут выглядеть и функционировать одинаково, независимо от того, в какой среде программирования и для какого компилятора они написаны. В книге сохранен весь материал двух предыдущих изданий, а также добавлено несколько новых глав и множество новых разделов. Эта книга — наиболее удобное руководство для самостоятельного изучения C++ в соответствии с требованиями нового стандарта и рассчитана на читателей, уже владеющих языком программирования C. Методика подачи материала предполагает строго последовательное изучение глав, содержащих множество примеров программ, а также упражнений для проверки и повторения пройденного материала.

Для программистов и опытных пользователей

УДК 681.3.06

Группа подготовки издания:

Главный редактор	*Вадим Сергеев*
Зав. редакцией	*Алексей Жданов*
Перевод с английского	*Алексея Жданова*
Компьютерная верстка:	*Ольги Сергиенко, Натальи Боговой*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Дмитрия Солнцева, Елены Клыковой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.10.04.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 55,5.
Доп. тираж 5000 экз. Заказ №591
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар No 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 0-07-882392-7 (англ.)
ISBN 5-7791-0086-1 (рус.)

Введение

Если вы уже знаете язык программирования C и хотели бы теперь заняться изучением C++ — эта книга для вас.

C++ — это попытка решения разработчиками языка C задач объектно-ориентированного программирования (Object Oriented Programming, OOP). Построенный на твердом фундаменте C, C++ помимо OOP поддерживает множество других полезных инструментов, не жертвуя при этом ни мощью, ни элегантностью, ни гибкостью C. C++ уже стал универсальным языком для программистов всего мира, языком, на котором будет написано следующее поколение высокоэффективного программного обеспечения. Это единственный серьезный язык, который просто обязан знать любой уважающий себя профессиональный программист.

C++ был разработан сотрудником научно-исследовательского центра AT&T Bell Laboratories (Нью-Джерси, США) Бьярном Страуструпом в 1979 году. Первоначальное название "C с классами" было изменено на C++ в 1983 году. Начиная с 1980 года C++ претерпел две существенные модернизации: в 1985 и 1990 годах. Последняя, третья модернизация связана с процессом стандартизации C++. Несколько лет назад началась работа по созданию единого международного стандарта по C++. Для этой цели был сформирован объединенный комитет по стандартизации ANSI (American National Standards Institute, Американский национальный институт стандартов) и ISO (International Standards Organization, Международная организация по стандартам) для языка C++. Первый рабочий проект указанного стандарта был представлен 25 января 1994 года. Комитет ANSI/ISO по C++ (членом которого являлся автор этой книги Герберт Шилдт) фактически сохранил все основные черты языка, заложенные туда еще Страуструпом и добавил несколько новых инструментов. В своей основе этот первый проект лишь отражал положение, в котором в то время находился язык C++.

Вскоре после завершения работы над первым проектом стандарта произошло событие, которое в конечном итоге и привело к его значительному расширению: Александр Степанов создал библиотеку стандартных шаблонов (Standard Template Library, STL). Как вы в дальнейшем узнаете, библиотека стандартных шаблонов устанавливает набор основополагающих процедур, которые можно использовать для обработки данных. Библиотека стандартных шаблонов — это мощный и элегантный инструмент программирования, но одновременно и очень объемный. Сразу после появления первого проекта стандарта комитет ANSI/ISO проголосовал за включение библиотеки стандартных шаблонов в спецификацию C++, что привело к значительному расширению C++ по сравнению с исходным определением этого языка. Несомненно став значительным событием в области программирования, создание библиотеки стан-

дартных шаблонов тем не менее привело к некоторому замедлению процесса стандартизации C++.

Справедливости ради надо сказать, что процесс стандартизации C++ отнял значительно больше времени, чем можно было предположить, когда он только начинался. Тем не менее, он близок к завершению. Комитетом ANSI/ISO разработан и предложен окончательный вариант проекта, который ожидает лишь формального одобрения. С практической точки зрения стандарт для C++ стал наконец реальностью. В появляющихся сейчас компиляторах поддерживаются все новые атрибуты C++.

Предлагаемый в книге материал учит языку программирования C++ в соответствии с новым стандартом этого языка (Standard C++). Именно эта версия предложена комитетом ANSI/ISO и именно она в настоящее время принята на вооружение основными производителями компиляторов. Таким образом, можно быть уверенным, что книга, которую вы начали изучать сегодня, завтра окажется столь же полезной.

Отличия третьего издания

Сейчас вы держите в руках третье издание книги *"Самоучитель C++"*. В ней сохранен весь материал двух предыдущих изданий, а также добавлены две новые главы и множество новых разделов. В первой из этих двух глав описывается динамическая идентификация типа (Run-Time Type Identification, RTTI) и новые, недавно разработанные операторы приведения типов. Во второй главе рассказывается о библиотеке стандартных шаблонов. Обе эти темы посвящены тем главным инструментам, которые были добавлены в C++ уже после выхода в свет предыдущего издания. В новых разделах других глав вы узнаете о пространствах имен, новом стиле оформления заголовков и современной системе ввода/вывода C++. Таким образом, третье издание книги *"Самоучитель C++"* оказалось существенно больше предыдущих.

Если вы работаете под Windows

Если на вашем компьютере установлена операционная система Windows, и вы хотите научиться писать программы для Windows, то C++ — это именно тот язык, который вам нужен. C++ полностью соответствует задачам программирования под Windows. Тем не менее ни одна из программ, предлагаемых в книге, не предназначена для работы в этой операционной системе. Наоборот, все эти программы запускаются из командной строки. Причина очевидна: программы для Windows по самой своей сути большие и сложные. По самым скромным подсчетам, для создания даже простейшей программы для Windows требуется от 50 до 70 строк исходного кода. При написании каждой такой программы для демонстрации возможностей языка C++ потребовалось бы написать тысячи строк исходного кода. Проще говоря, Windows — это не самая подходящая среда для изучения языка программирования. Тем не менее

для компиляции предлагаемых в книге программ вполне подходит компилятор, работающий в среде Windows, поскольку при выполнении программ он автоматически перейдет в консольный режим.

Когда вы в совершенстве овладеете C++, вы несомненно сумеете применить свои знания для программирования под Windows. Действительно, программирование на C++ под Windows позволяет пользоваться библиотеками классов, например, библиотекой классов MFC (Microsoft Foundation Classes), что существенно упрощает разработку приложений. Кроме этого, интерфейс любого приложения под Windows достаточно просто создать с помощью таких средств визуального программирования, как Visual C++ 5 или Borland C++ 5. Сердцевиной же любого профессионального приложения является программная реализация его идеи, а отнюдь не пользовательский интерфейс, пусть даже самый что ни на есть дружественный. Другими словами, эта книга учит не созданию пользовательского интерфейса в стиле Windows, а собственно языку программирования C++.

Как организована эта книга

Эта книга является по-своему уникальной, поскольку учит языку программирования C++, опираясь на передовую методику обучения. Эта методика предполагает знакомство на каждом занятии с единственной темой, дополненной для лучшего ее усвоения примерами и упражнениями. Такой подход гарантирует, что перед тем как перейти к следующей теме, вы полностью освоите предыдущую.

Материал книги представлен в определенном порядке. Поэтому изучайте ее последовательно, от главы к главе. В каждой новой главе предполагается, что вы уже освоили материал всех предыдущих. Во всех главах, за исключением главы 1, имеются упражнения для проверки ваших знаний предыдущей главы, упражнения для проверки усвоения материала текущей главы, а также упражнения для проверки усвоения всего изученного на данном этапе материала. Ответы на упражнения находятся в приложении B в конце книги.

В книге предполагается, что вы уже являетесь состоявшимся программистом на языке C. Проще говоря, гораздо легче учиться программировать на C++, когда уже умеешь это делать на C. Если вы еще не умеете программировать на C, то перед тем как взяться за эту книгу, лучше потратить некоторое время на его изучение.

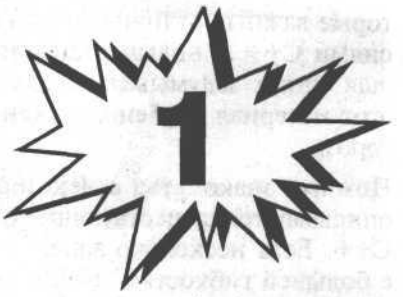
Исходные коды программ

Исходные коды представленных в книге программ можно найти либо на прилагаемой дискете, описание которой находится в приложении D в конце книги, либо в Internet по адресу **http://www.osborne.com.** Использование этих кодов освободит вас от необходимости набирать их вручную.


КЕЙ
Компьютер
Центр
КЕЙ
компьютеры
периферия
Литейный, 59
327-3101
программы
Марата, 8
325-3216
Средний пр.ВО,34
327-4977
комплектующие
расходные
материалы
Кирочная, 19
327-3100
www.key.ru

Глава 1

Краткий обзор C++



C++ — это расширенная версия языка C. C++ содержит в себе все, что имеется в C, но кроме этого он поддерживает объектно-ориентированное программирование (Object Oriented Programming, OOP). В C++ имеется множество дополнительных возможностей, которые независимо от объектно-ориентированного программирования делают его просто "лучше, чем C". За небольшими исключениями C++ — это более совершенный C. В то время как все, что вы знаете о языке C, вполне применимо и к C++, понимание его новых свойств все же потребует от вас значительных затрат времени и сил. Однако преимущества программирования на C++ с лихвой окупят ваши усилия.

Целью этой главы должно стать знакомство с некоторыми наиболее важными свойствами C++. Как вы знаете, элементы языка программирования не существуют в пустоте, изолированно от других. Они работают вместе в виде полной, законченной конструкции. В C++ эта взаимозависимость еще более ярко выражена, чем в C. В результате, трудно обсуждать любой аспект C++ без других его аспектов. Поэтому первая глава посвящена предварительному знакомству с теми свойствами C++, без которых сложно понять приводимые здесь примеры программ. Более подробно эти свойства будут изучаться в следующих главах.

Эта глава также освещает некоторые отличия между стилями программирования на языках C и C++. В C++ имеется несколько возможностей для написания более гибких, чем в C, программ. Хотя некоторые из этих возможностей имеют очень слабую связь с объектно-ориентированным программированием, или вообще ее не имеют, тем не менее, поскольку они содержатся в большинстве программ C++, стоит обсудить их в первую очередь.

Поскольку C++ был задуман для поддержки объектно-ориентированного программирования, эта глава начинается с описания OOP. Как вы увидите, многие свойства C++ тем или иным образом касаются OOP. Однако важно понимать, что C++ может использоваться для написания не только объектно-ориентированных программ. То, как вы используете C++, полностью зависит от вас.

К моменту написания этой книги процесс стандартизации языка программирования C++ был завершен. По этой причине здесь описываются неко-

торые важные отличия между обычными для последних нескольких лет версиями C++ и новым стандартом языка (Standard C++). Поскольку настоящая книга задумывалась как пособие для обучения языку Standard C++, этот материал особенно важен для тех, кто работает с устаревшим компилятором.

Помимо знакомства с некоторыми важными свойствами C++, в этой главе описываются существующие отличия между стилями программирования C и C++. Есть несколько аспектов C++, которые позволяют писать программы с большей гибкостью. Некоторые из этих аспектов C++ имеют очень незначительную связь с объектно-ориентированным программированием или вообще ее не имеют, но поскольку они встречаются в большинстве программ на C++, стоит обсудить их в начале книги.

Перед тем как начать собственно изложение материала, имеет смысл сделать несколько важных замечаний о природе и форме C++. Как правило, программы на C++ внешне напоминают программы на C. Так же, как и на C, программы на C++ начинают выполняться с функции **main()**. Для получения аргументов командной строки C++ использует те же параметры **argc, argv**, что и C. Хотя C++ определяет их в собственной объектно-ориентированной библиотеке, он также поддерживает все функции стандартной библиотеки C. В C++ используются те же управляющие структуры и те же встроенные типы данных, что и в C.

Запомните, в этой книге предполагается, что вы уже знаете язык программирования C. Проще говоря, вы уже должны уметь программировать на C перед тем, как начнете изучать программирование на C++. Если вы еще не знаете C, то желательно потратить некоторое время на его изучение.

В этой книге предполагается, что вы знаете, как компилировать и выполнять программу, используя компилятор C++, Если это не так, вам следует обратиться к соответствующему руководству пользователя. (Из-за отличий в компиляторах дать в этой книге инструкцию для работы с любым из них не представляется возможным.) Поскольку программирование лучше изучать в работе, вам настоятельно рекомендуется вводить, компилировать и запускать приводимые в книге примеры программ в том порядке, в котором они представлены.

1.1. Что такое объектно-ориентированное программирование?

Объектно-ориентированное программирование — это новый подход к созданию программ. По мере развития вычислительной техники возникали разные методики программирования. На каждом этапе создавался новый подход, который помогал программистам справляться с растущим усложне-

нием программ. Первые программы создавались посредством ключевых переключателей на передней панели компьютера. Очевидно, что такой способ подходит только для очень небольших программ. Затем был изобретен язык ассемблера, который позволял писать более длинные программы. Следующий шаг был сделан в 1950 году, когда был создан первый язык высокого уровня Фортран.

Используя язык высокого уровня, программисты могли писать программы до нескольких тысяч строк длиной. Для того времени указанный подход к программированию был наиболее перспективным. Однако язык программирования, легко понимаемый в коротких программах, когда дело касалось больших программ, становился нечитабельным (и неуправляемым). Избавление от таких неструктурированных программ пришло после изобретения в 1960 году *языков структурного программирования* (structured programming language). К ним относятся языки Алгол, Паскаль и C. Структурное программирование подразумевает точно обозначенные управляющие структуры, программные блоки, отсутствие (или, по крайней мере, минимальное использование) инструкций GOTO, автономные подпрограммы, в которых поддерживается рекурсия и локальные переменные. Сутью структурного программирования является возможность разбиения программы на составляющие ее элементы. Используя структурное программирование, средний программист может создавать и поддерживать программы свыше 50 000 строк длиной.

Хотя структурное программирование, при его использовании для написания умеренно сложных программ, принесло выдающиеся результаты, даже оно оказывалось несостоятельным тогда, когда программа достигала определенной длины. Чтобы написать более сложную программу, необходим был новый подход к программированию. В итоге были разработаны принципы объектно-ориентированного программирования. ООР аккумулирует лучшие идеи, воплощенные в структурном программировании, и сочетает их с мощными новыми концепциями, которые позволяют оптимально организовывать ваши программы. Объектно-ориентированное программирование позволяет вам разложить проблему на составные части. Каждая составляющая становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом случае вся процедура в целом упрощается, и программист получает возможность оперировать с гораздо большими по объему программами.

Все языки ООР, включая C++, основаны на трех основополагающих концепциях, называемых инкапсуляцией, полиморфизмом и наследованием. Рассмотрим эти концепции.

Инкапсуляция

Инкапсуляция (encapsulation) — это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-

ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создается так называемый "черный ящик". Когда коды и данные объединяются таким способом, создается *объект (object)*. Другими словами, объект — это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть *закрытыми (private)* для этого объекта или *открытыми (public)*. Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

На самом деле объект является переменной определенного пользователем типа. Может показаться странным, что объект, который объединяет коды и данные, можно рассматривать как переменную. Однако применительно к объектно-ориентированному программированию это именно так. Каждый элемент данных такого типа является составной переменной.

Полиморфизм

Полиморфизм (polymorphism) (от греческого polymorphos) — это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например, для языка С, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трех различных функций: **abs(),** labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа **abs().** (Один из способов, который позволяет это делать, показан далее в этой главе.) Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. Как вы увидите, в C++ можно использовать одно имя функции для множества различных действий. Это называется *перегрузкой функций (function overloading)*.

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания *единого класса действий.* Выбор же *конкретного действия,* в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот вы-

бор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в C, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется *перегрузкой операторов (operator overloading)*.

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

Наследование

Наследование (inheritance) — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию *иерархии классов (hierarchical classification)*. Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом — это часть общего класса, называемого **строением.** С другой стороны, **строение** — это часть более общего класса — **конструкции,** который является частью еще более общего класса объектов, который можно назвать **созданием рук Человека.** В каждом случае порожденный класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путем определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Как вы увидите, наследование играет очень важную роль в ООР.

1. Инкапсуляция не является исключительной прерогативой ООР. Некоторая степень инкапсуляция может быть достигнута и в языке C. Например, при применении библиотечной функции, в конечном итоге, имеет место концепция черного ящика, содержимое которого вы не можете изменить (исключая,

возможно, злой умысел). Рассмотрим функцию **fopen()**. Если она используется для открытия файла, то создаются и инициализируются несколько внутренних переменных. В той мере, в которой это касается вашей программы, эти переменные скрыты и недоступны. Но, конечно, C++ обеспечивает более надежную поддержку инкапсуляции.

2. В реальной жизни примеры полиморфизма вполне обычны. Например, рассмотрим рулевое колесо автомобиля. Оно работает одинаково, независимо от того, используется ли при этом электропривод, механический привод или стандартное ручное управление. Ключевым является то, что интерфейс (рулевое колесо) один и тот же, независимо от того, какой рулевой механизм (метод) применяется на самом деле.
3. Наследование свойств и базовая концепция классов являются основополагающими для организации пути познания. Например, **сельдерей** — член класса **овощей,** которые являются частью класса **растений.** В свою очередь, растения являются живыми организмами и так далее. Без иерархии классов систематизация знаний была бы невозможна.

1. Подумайте о том, какую важную роль в нашей повседневной жизни играют классы и полиморфизм.

1.2. Две версии C++

При написании этой книги C++ находился на перепутье. Как указано в предисловии, в последние годы активно шел процесс стандартизации C++. Целью стандартизации было создание стабильного, ориентированного на будущее языка, которой призван удовлетворить потребности программистов следующего столетия. Результатом стало параллельное существование двух версий C++. Первая, традиционная версия базируется на исходной разработке Бьярна Страуструпа. Это та версия, которая использовалась программистами последние лет десять. Вторая версия, названная Standard C++, создана Бьярном Страуструпом совместно с комитетом по стандартизации (ANSI — American National Standards Institute, Американский национальный институт стандартов; ISO — International Standards Organization, Международная организация по стандартам). Хотя по сути эти две версии очень похожи, Standard C++ содержит несколько усовершенствований, которых нет в традиционном C++. Таким образом Standard C++ по существу является надмножеством традиционного C++.

Настоящая книга учит языку Standard C++. Эта версия C++, определенная комитетом по стандартизации ANSI/ISO, должна быть реализована во всех

современных компиляторах C++. Примеры программ этой книги отражают современный стиль программирования в соответствии с новыми реалиями языка Standard C++. Это означает актуальность содержания книги не только сегодня, но и в будущем. Проще говоря, Standard C++ — это будущее. А поскольку Standard C++ содержит в себе черты всех ранних версий C++, то, что вы узнаете из этой книги, позволит вам работать в любой программной среде C++.

Тем не менее, если вы используете устаревший компилятор, не все приведенные в книге программы будут вам доступны, поскольку в процессе стандартизации комитет ANSI/ISO добавил к языку массу новых черт. По мере определения новых черт, они реализовывались производителями компиляторов. Естественно, что между добавлением новой возможности языка и ее доступностью в коммерческих компиляторах всегда есть определенный промежуток времени, а поскольку такие возможности добавлялись в течение нескольких лет, старые компиляторы могут не поддерживать некоторые из них. Это важно, так как два недавних добавления к языку C++ имеют отношение ко всем, даже простейшим программам. Если у вас устаревший компилятор, в котором указанные новые возможности недоступны, не расстраивайтесь, в следующих разделах описано несколько простых способов обойти проблему.

Отличия между прежним и современным стилями программирования в числе прочих включают две новые черты: изменился стиль оформления заголовков (headers) и появилась инструкция **namespace.** Чтобы продемонстрировать эти отличия, начнем с рассмотрения двух версий простейшей программы на C++. Первая показанная здесь версия написана в прежнем, еще совсем недавно основном стиле программирования.

```
/*
   Программа на C++ в традиционном стиле
*/

#include <iostream.h>

int main()
{
   /* программный код */
   return 0;
}
```

Поскольку C++ строится на C, этот каркас программы должен быть хорошо вам знаком, тем не менее обратите особое внимание на инструкцию **#include**. Эта инструкция подключает к программе заголовочный файл iostream.h, который обеспечивает поддержку системы ввода/вывода C++. (В C++ этот файл имеет то же самое назначение, что и файл stdio.h в C.)

Ниже представлена вторая версия программы, в которой используется современный стиль:

```
/*
   Программа на C++ в современном стиле. Здесь используются
   новое оформление заголовков и ключевое слово namespace
*/

#include <iostream>
using namespace std;
int main()
{
   /* программный код */
   return 0;
}
```

Обратите внимание на две строки в самом начале программы, в которых имеют место изменения. Во-первых, в инструкции **#include** после слова **iostream** отсутствуют символы .h. Во-вторых, в следующей строке задается так называемое *пространство имен* (namespace). Хотя подробно эти нововведения будут рассмотрены позднее, сейчас дадим их краткий обзор.

Новые заголовки в программах на C++

Как вам должно быть известно из опыта программирования на C, при использовании библиотечной функции в программу необходимо включить заголовочный файл. Это делается с помощью инструкции **#include**. Например, при написании программ на языке C заголовочным файлом для функций ввода/вывода является файл **stdio.h,** который включается в программу с помощью следующей инструкции:

```
#include <stdio.h>
```

Здесь **stdio.h** — это имя файла, который используется функциями ввода/вывода, и предыдущая инструкция заставляет компилятор *включить* указанный файл в вашу программу.

В первые несколько лет после появления C++ в нем использовался тот же стиль оформления заголовков, что и в C. Для совместимости с прежними программами в языке Standard C++ этот стиль по-прежнему поддерживается. Тем не менее при работе с библиотекой Standard C++ в соответствии с новым стилем вместо имен заголовочных файлов указываются стандартные идентификаторы, по которым компилятор находит требуемые файлы. Новые заголовки C++ являются абстракциями, гарантирующими объявление соответствующих прототипов и определений библиотеки языка Standard C++.

Поскольку новые заголовки не являются именами файлов, для них не нужно указывать расширение **.h**, а только имя заголовка в угловых скобках. Ниже представлены несколько заголовков, поддерживаемых в языке Standard C++:

```
<iostream>
<fstream>
<vector>
<string>
```

Такие заголовки по-прежнему включаются в программу с помощью инструкции **#include**. Единственным отличием является то, что новые заголовки совершенно не обязательно являются именами файлов.

Поскольку C++ содержит всю библиотеку функций C, по-прежнему поддерживается стандартный стиль оформления заголовочных файлов библиотеки C. Таким образом, такие заголовочные файлы, как stdio.h и ctype.h все еще доступны. Однако Standard C++ также определяет заголовки нового стиля, которые можно указывать вместо этих заголовочных файлов. В соответствии с версией C++ к стандартным заголовкам C просто добавляется префикс c и удаляется расширение **.h**. Например, заголовок **math.h** заменяется новым заголовком C++ **<cmath>**, а заголовок **string.h** — заголовком **<cstring>**. Хотя в настоящее время при работе с функциями библиотеки C допускается включать в программы заголовочные файлы в соответствии со стилем C, такой подход не одобряется стандартом языка Standard C++. (То есть, он не рекомендуется.) По этой причине во всех имеющихся в книге инструкциях #include используется новый стиль написания заголовков программ. Если ваш компилятор для функций библиотеки C не поддерживает заголовки нового стиля, просто замените их заголовками в стиле C.

Поскольку заголовки нового стиля появились в C++ совсем недавно, во многих и многих прежних программах вы их не найдете. В этих программах в соответствии со стилем C в заголовках указаны имена файлов. Ниже представлен традиционный способ включения в программу заголовка для функций ввода/вывода:

```
#include <iostream.h>
```

Эта инструкция заставляет компилятор включить в программу заголовочный файл **iostream.h**. Как правило, в заголовках прежнего стиля вместе с расширением **.h** используется то же имя, что и в соответствующих им новых заголовках.

Как уже отмечалось, все компиляторы C++ поддерживают заголовки старого стиля. Тем не менее такие заголовки объявлены устаревшими и не рекомендуются. Именно поэтому в книге вы их больше не встретите.

Несмотря на повсеместное распространение в программах заголовков старого стиля, они считаются устаревшими.

Пространства имен

Когда вы включаете в программу заголовок нового стиля, содержание этого заголовка оказывается в пространстве имен **std.** *Пространство имен* (namespace) — это просто некая объявляемая область, необходимая для того, чтобы избежать конфликтов имен идентификаторов. Традиционно имена библиотечных функций и других подобных идентификаторов располагались в глобальном пространстве имен (как, например, в C). Однако содержание заголовков нового стиля помещается в пространстве имен **std.** Позднее мы рассмотрим пространства имен более подробно. Сейчас же, чтобы пространство имен **std** стало видимым, просто используйте следующую инструкцию:

```
using namespace std;
```

Эта инструкция помещает std в глобальное пространство имен. После того как компилятор обработает эту инструкцию, вы сможете работать с заголовками как старого, так и нового стиля.

Если вы работаете со старым компилятором

Как уже упоминалось, заголовки нового стиля и пространства имен появились в C++ совсем недавно, поэтому эти черты языка могут не поддерживаться старыми компиляторами. Если у вас один из таких компиляторов, то при попытке компиляции первых двух строк кода, приводимых в книге примеров программ, вы получите одно или несколько сообщений об ошибках. Обойти эту проблему просто — удалите инструкцию **namespace** и используйте заголовки старого стиля. То есть замените, например, инструкции

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

на инструкцию

```
#include <iostream.h>
```

Это простое действие превратит современную программу в такую же, но в традиционном стиле. Поскольку заголовок старого стиля считывает все свое содержание в глобальное пространство имен, необходимость в инструкции **namespace** отпадает.

И еще одно замечание. Еще в течение нескольких лет вы будете встречать программы, в которых заголовки будут оформлены в старом стиле и не будет инструкций **namespace.** Ваш компилятор C++ будет прекрасно справляться с такими программами. Тем не менее, что касается новых программ, вам следует использовать современный стиль, поскольку именно он определен стандартом языка Standard C++. Хотя программы прежнего стиля будут поддерживаться еще многие годы, технически они некорректны.

1. Перед тем как продолжить, попытайтесь откомпилировать представленный выше пример простейшей программы. Хотя эта программа не выполняет никаких действий, попытка ее компиляции поможет определить, поддерживает ли ваш компилятор современный синтаксис C++. Если он не принимает заголовки нового стиля и инструкцию **namespace,** замените их, как только что было описано. Запомните, если ваш компилятор не принимает код нового стиля, вам придется сделать изменения в каждой программе этой книги.

1.3. Консольный ввод и вывод в C++

Поскольку C++ — это улучшенный C, все элементы языка C содержатся также и в C++. Это подразумевает, что все программы, написанные на C, по умолчанию являются также и программами на C++. (На самом деле имеется несколько очень незначительных исключений из этого правила, которые будут рассмотрены позже.) Поэтому можно писать программы на C++, которые будут выглядеть точно так же, как и программы на C. Ошибки не будет, это только будет означать, что вы не смогли использовать все преимущества C++. Чтобы по достоинству оценить C++, необходимо писать программы в стиле C++.

Вероятно, наиболее своеобразной чертой языка C++, используемой программистами, является подход к вводу и выводу. Хотя такие функции, как printf() и **scanf(),** по-прежнему доступны, C++ обеспечивает иной, лучший способ выполнения этих операций. В C++ ввод/вывод выполняется с использованием *операторов,* а не функций ввода/вывода. Оператор вывода — это **<<,** а оператор ввода — **>>**. Как вы знаете, в C эти операторы являются, соответственно, операторами левого и правого сдвига. В C++ они сохраняют свое первоначальное значение (левый и правый сдвиг), выполняя при этом еще ввод и вывод. Рассмотрим следующую инструкцию C++:

```
cout << "Эта строка выводится на экран.\n";
```

Эта инструкция осуществляет вывод строки в заранее определенный поток **cout,** который автоматически связывается с терминалом, когда программа C++ начинает выполняться. Это напоминает действие функции **stdout** в языке C. Как и в C, терминал для ввода/вывода в C++ может быть переопределен, но пока будем считать, что используется экран.

С помощью оператора вывода **<<** можно вывести данные любого базового типа C++. Например, следующая инструкция осуществляет вывод величины 100.99:

```
cout << 100.99;
```

В общем случае, для вывода на экран терминала используется следующая обычная форма оператора <<:

```
cout << выражение;
```

Здесь ***выражение*** может быть любым действительным выражением C++, включая другие выражения вывода.

Для считывания значения с клавиатуры, используйте оператор ввода >>. Например, в этом фрагменте целая величина вводится в тип:

```
int пит;
cin >> пит;
```

Обратите внимание, что переменной **num** *не* предшествует амперсанд &. Как вы знаете, при вводе с использованием функции **scanf()** языка C ей должны передаваться адреса переменных. Только тогда они смогут получить значения, вводимые пользователем. В случае использования оператора ввода C++ все происходит иначе. (Смысл этого станет ясен после того, как вы больше узнаете о C++.)

В общем случае для ввода значения с клавиатуры, используйте следующую форму оператора >>:

```
cin >> переменная;
```

Расширенное толкование символов << и >> является примером перегрузки операторов.

Для правильного использования операторов ввода/вывода в C++ вы должны включить в программу заголовочный файл iostream.h. Он является одним из стандартных заголовочных файлов C++ и поставляется с компилятором.

1. В этой программе выводится строка, два целых числа и одно число с плавающей точкой двойной точности:

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

int main()
{
    int i, j;
    double d;
```

```
    i = 10;
    j = 20;
    d = 99.101;

    cout << "Вот несколько чисел: ";
    cout << i;
    cout << ' ';
    cout << j;
    cout << ' ';
    cout << d;

    return 0;
}
```

Ниже представлен результат работы программы:

```
Вот несколько чисел: 10 20 99.101
```

Если вы работаете с устаревшим компилятором, ему могут оказаться недоступными заголовки нового стиля и инструкции ***namespace,*** *используемые в этой и во всех остальных программах книги. Если это так, замените соответствующие инструкции описанным в предыдущем разделе способом.*

2. В одном выражении ввода/вывода можно выводить более одной величины. Например, версия программы, описанной в примере 1, показывает один из эффективных способов программирования инструкций ввода/вывода.

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

int main()
{
    int i, j;
    double d;

    i = 10;
    j = 20;
    d = 99.101;

    cout << "Вот несколько чисел: ";
    cout << i << ' ' << j << ' ' << d;

    return 0;
}
```

Здесь в строке

```
cout << i << ' ' << j << ' ' << d;
```

выводится несколько элементов данных в одном выражении. В общем случае вы можете использовать единственную инструкцию для вывода любого требуемого количества элементов данных. Если это кажется неудобным, просто запомните, что оператор вывода << ведет себя так же, как и любой другой оператор C++, и может быть частью произвольно длинного выражения.

Обратите внимание, что по мере необходимости следует включать в программу пробелы между элементами данных. Если пробелов не будет, то данные, выводимые на экран, будет неудобно читать.

3. Это программа предлагает пользователю ввести целое число:

```
ttinclude<iostream>
using namespace std;

int main()
{
    int i;

    cout << "Введите число: ";
    cin>>i;
    cout << "Вот ваше число: " << i << "\n";

    return 0;
}
```

Результат работы программы:

```
Введите число: 100
Вот ваше число: 100
```

Как видите, введенное пользователем значение действительно оказывается в i.

4. Следующая программа — это программа ввода целого, числа с плавающей точкой и строки символов. В ней для ввода всего перечисленного используется одна инструкция.

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

int main()
{
    int i;
    float f;
    char s[80];

    cout << "Введите целое, число с плавающей точкой и строку: ";
    cin >> i >> f >> s;
```

```
    cout << "Вот ваши данные: ";
    cout << i << ' ' << f << ' ' << s;

    return 0;
}
```

Как видно из этого примера, можно ввести в одной инструкции ввода столько элементов данных, сколько нужно. Как и в C, различные элементы данных при вводе должны быть отделены друг от друга (пробелами, табуляциями или символами новой строки).

При считывании строки ввод будет остановлен после считывания первого разделительного символа. Например, если вы введете:

```
10 100.12 Это проверка
```

то на экран будет выведено:

```
10 100.12 Это
```

Так происходит потому, что считывание строки прекращается при вводе пробела после слова Это. Остаток строки остается в буфере ввода, в ожидании следующей операции ввода. (Это похоже на ввод строки с использованием функции **scanf()** в формате **%s**.)

5. По умолчанию при использовании оператора >> буферизуется весь ввод строки. Это означает, что до тех пор, пока вы не нажмете клавишу <Enter>, информация не будет передана в вашу программу. (В языке C функция **scanf()** также буферизует ввод строки, поэтому такой стиль ввода не должен быть для вас чем-то новым.) Для исследования построчно-буферизованного ввода рассмотрим следующую программу:

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

int main()
{
    char ch;

    cout << "Вводите символы, для окончания ввода введите x. \n";

    do {
     cout << ":";
     cin >>ch;
    } while (ch !='x');

    return 0;
}
```

Когда вы протестируете эту программу, то убедитесь, что для считывания каждого очередного символа необходимо нажимать клавишу <Enter>.

1. Напишите программу для ввода количества отработанных персоналом часов и размера почасовой оплаты каждого. Затем выведите суммарную зарплату персонала. (Удостоверьтесь в правильности ввода.)
2. Напишите программу для преобразования футов в дюймы. Организуйте ввод числа футов и вывод на экран соответствующего числа дюймов. Повторяйте эту процедуру до тех пор, пока пользователь не введет 0 в качестве числа футов.
3. Ниже приведена программа на языке C. Перепишите ее в соответствии со стилем ввода/вывода C++.

```
/* Преобразуйте эту программу на C в соответствии со стилем
   программирования C++. Эта программа подсчитывает наименьшее общее
   кратное
*/
#include <stdio.h>

int main(void)
{
    int a, b, d, min;

    printf ("Введите два числа:");
    scanf("%d%d", &a, &b);
    min = a > b ? b: a;
    for (d=2; d<min; d++)
            if (((a%d)==0) && ((b%d)==0)) break;
        if (d==min) {
            printf("Нет общего кратного\n");
            return 0;
          }
      printf("Наименьшее общее кратное равно %d\n", d) ;
      return 0;
}
```

1.4. Комментарии в C++

В C++ комментарии в программу можно включать двумя различными способами. Первый способ — это использование стандартного механизма, такого же, как в C, т. е. комментарий начинается с /* и оканчивается */. Как и в C, в C++ этот тип комментария не может быть вложенным.

Вторым способом, которым вы можете писать комментарии в программах C++, является *однострочный комментарий*. Однострочный комментарий начинается с символов // и заканчивается концом строки. Другого символа, помимо физического конца строки (такого, как возврат каретки/перевод строки), в однострочном комментарии не используется.

Обычно программисты C++ используют стиль C для многострочных комментариев, а для коротких замечаний используют однострочные комментарии в соответствии со стилем C++.

1. Программа, в которой есть стили комментариев как C, так и C++:

```
/* Этот комментарий в стиле C. Данная программа определяет
четность целого
*/

#include <iostream>
using namespace std;

int main()
{
    int num; // это однострочный комментарий C++

    // чтение числа
    cout << "Введите проверяемое число:";
    cin>>num;

    // проверка на четность
    if ( (num%2)==0) cout << "Число четное\n";
    else cout << "Число нечетное\n";

    return 0;
}
```

2. Хотя многострочные комментарии не могут быть вложенными, однострочный комментарий в стиле C++ можно вкладывать внутрь многострочного комментария. Например, это совершенно правильный фрагмент:

```
/* Это многострочный комментарий,
внутрь которого // вложен однострочный комментарий.
Это окончание многострочного комментария.
*/
```

Тот факт, что однострочный комментарий может быть вложен в многострочный, дает возможность при отладке "помечать" некоторые строки программы.

1. В качестве эксперимента проверьте, имеет ли комментарий, в котором комментарий стиля C вложен внутрь однострочного комментария C++, право на жизнь:

```
// Это странный /*способ делать комментарии*/
```

2. Добавьте комментарии к ответам на упражнения в разделе 1.3.

1.5. Классы. Первое знакомство

Вероятно, одним из наиболее важных понятий C++ является класс. Класс — это механизм для создания объектов. В этом смысле класс лежит в основе многих свойств C++. Хотя более детально понятие класса раскрывается в следующих главах, оно столь фундаментально для программирования на C++, что краткий обзор здесь необходим.

Класс объявляется с помощью ключевого слова **class.** Синтаксис объявления класса похож на синтаксис объявления структуры. Здесь показана основная форма:

```
class имя_класса {
    закрытые функции и переменные класса
public:
    открытые функции и переменные класса
} список_объектов;
```

В объявлении класса ***список_объектов*** не обязателен. Как и в случае со структурой, вы можете объявлять объекты класса позже, по мере необходимости. Хотя ***имя_класса*** также не обязательно, с точки зрения практики оно необходимо. Доводом в пользу этого является то, что ***имя_класса*** становится именем нового типа данных, которое используется для объявления объектов класса.

Функции и переменные, объявленные внутри объявления класса, становятся, как говорят, *членами (members)* этого класса. По умолчанию все функции и переменные, объявленные в классе, становятся закрытыми для класса. Это означает, что они доступны только для других членов того же класса. Для объявления открытых членов класса используется ключевое слово **public,** за которым следует двоеточие. Все функции и переменные, объявленные после слова **public,** доступны как для других членов класса, так и для любой другой части программы, в которой находится этот класс.

Ниже приводится простое объявление класса:

```
class myclass {
    // закрытый элемент класса
    int a;
public:
    void set_a(int num);
    int get_a();
};
```

Этот класс имеет одну закрытую переменную a, и две открытые функции, **set_a() и get_a().** Обратите внимание, что прототипы функций объявляются

внутри класса. Функции, которые объявляются внутри класса, называются *функциями-членами (memberfunctions).*

Поскольку a является закрытой переменной класса, она недоступна для любой функции вне **myclass.** Однако поскольку **set_a() и get_a()** являются членами **myclass,** они имеют доступ к a. Более того, **set_a() и get_a(),** являясь открытыми членами **myclass,** могут вызываться из любой части программы, использующей **myclass.**

Хотя функции **set_a() и get_a()** и объявлены в **myclass,** они еще не определены. Для определения функции-члена вы должны связать имя класса, частью которого является функция-член, с именем функции. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называются *оператором расширения области видимости (scope resolution operator).* Например, далее показан способ определения функций-членов **set_a() и get_a():**

```
void myclass: : set_a (int num)
{
    a=num;
}

int myclass::get_a()
{
    return a;
)
```

Отметим, что и **set_a() и get_a()** имеют доступ к переменной a, которая для **myclass** является закрытой. Как уже говорилось, поскольку **set_a() и get_a()** являются членами **myclass,** они могут напрямую оперировать с его закрытыми данными.

При определении функции-члена пользуйтесь следующей основной формой:

```
Тип_возвр_значения имя_класса: : имя_функции (список_параметров)
(
    ...// тело функции
}
```

Здесь ***имя_класса*** — это имя того класса, которому принадлежит определяемая функция.

Объявление класса **myclass** не задает ни одного объекта типа **myclass,** оно определяет только тип объекта, который будет создан при его фактическом объявлении. Чтобы создать объект, используйте имя класса, как спецификатор типа данных. Например, в этой строке объявляются два объекта типа **myclass:**

```
myclass ob1, ob2; // это объекты типа myclass
```

Объявление класса является логической абстракцией, которая задает новый тип объекта. Объявление же объекта создает физическую сущность объекта такого типа. То есть, объект занимает память, а задание типа нет.

После того как объект класса создан, можно обращаться к открытым членам класса, используя оператор точка (.), аналогично тому, как осуществляется доступ к членам структуры. Предположим, что ранее объекты были объявлены, тогда следующие инструкции вызывают set_a() для объектов **ob1** и **ob2:**

```
ob1.set_a(10); // установка версии а объекта оЫ равной 10
ob2.set_a(99); // установка версии а объекта оЬ2 равной 99
```

Как видно из комментариев, эти инструкции устанавливают значение переменной а объекта оЫ равной 10 и значение переменной а объекта **ob2** равной 99. Каждый объект содержит собственную копию всех данных, объявленных в классе. Это значит, что а в **ob1** отлично от а в **ob2**.

Каждый объект класса имеет собственную копию всех переменных, объявленных внутри класса.

1. В качестве первого простого примера, рассмотрим программу, в которой используется **myclass,** описанный в тексте, для задания значений а для оЫ и **ob2** и вывода на экран этих значений для каждого объекта:

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

class myclass { -
    // закрытая часть myclass
    int a;
public:
    void set_a(int num);
    int get_a();
};

void myclass: : set_a (int num)
{
    a=num;
}
```

```
int myclass::get_a()
{
    return a;
}

int main()
{
    myclass ob1, ob2;

    ob1.set_a(10);
    ob2.set_a(99);

    cout << ob1.get_a() << "\n";
    cout << ob2.get_a() << "\n";

    return 0;
}
```

Как и следовало ожидать, программа выводит на экран величины 10 и 99.

2. В предыдущем примере переменная **a** в **myclass** является закрытой. Это означает, что она непосредственно доступна только для членов **myclass.** (Это один из доводов в пользу существования открытой функции **get_a().**) Если вы попытаетесь обратиться к закрытому члену класса из той части вашей программы, которая не является членом этого класса, то результатом будет ошибка при компиляции. Например, предположим, что **myclass** задан так, как показано в предыдущем примере, тогда компиляция функции **main()** вызовет ошибку:

```
// Этот фрагмент содержит ошибку
#include <iostream>
using namespace std;

int main()
{
    myclass ob1, ob2;

    ob1.a = 10;   // ОШИБКА! к закрытому члену нет
    ob2.a = 99;   // доступа для функции — не члена

    cout << ob1.get_a() << "\n";
    cout << ob2.get_a() << "\n";

    return 0;
}
```

3. Точно так же, как открытые функции-члены, могут существовать и открытые переменные-члены. Например, если бы a была объявлена в открытой секции **myclass,** тогда к ней, как показано ниже, можно было бы обратиться из любой части программы:

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

class myclass {
public:
    // теперь a открыта
    int a;
    // и здесь не нужны функции set_a() и get_a()
};

int main()
{
    myclass ob1, ob2;

    // здесь есть явный доступ к a
    ob1.a = 10;
    ob2.a = 99;

    cout << ob1.a << "\n";
    cout << ob2.a << "\n";

    return 0;
}
```

В этом примере, поскольку a объявлена открытым членом **myclass,** к ней имеется явный доступ из **main()**. Обратите внимание, как оператор точка (.) используется для доступа к a. Обычно, когда вы вызываете функцию-член, или осуществляете доступ к переменной-члену не из класса, которому они принадлежат, за именем объекта должен следовать оператор точка (.), а за ним имя члена. Это необходимо для исчерпывающего определения того, с членом какого объекта вы имеете дело.

4. Чтобы по достоинству оценить возможности объектов, рассмотрим более практичный пример. В этой программе создается класс **stack,** реализующий стек, который можно использовать для хранения символов:

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

#define SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека

public:
    void init(); // инициализация стека
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};
```

```
// Инициализация стека
void stack::init()
{
    tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон";
        return;
     }
    stck[tos] = ch; '
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    stack s1, s2; // создание двух стеков
    int i;
    // инициализация стеков
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0;i<3;i++) cout << "символ из s1:" << s1.pop() << "\n";
    for(i=0;i<3;i++) cout << "символ из s2:" << s2.pop() << "\n";

    return 0;
}
```

Эта программа выводит на экран следующее:

```
символ из s1: c
символ из s1: b
символ из s1: a
символ из s2: z
символ из s2: y
символ из s2: x
```

Давайте теперь детально проанализируем программу. Класс stack содержит две закрытые переменные: stck и tos. Массив stck содержит символы, фактически помещаемые в стек, а tos содержит индекс вершины стека. Открытыми функциями стека являются init(), push() и pop(), которые, соответственно, инициализируют стек, помещают символ в стек и выталкивают его из стека.

Внутри функции **main**() создаются два стека, **s1** и s2, и по три символа помещаются в каждый из них. Важно понимать, что один объект (стек) не зависит от другого. Поэтому у символов в **s1** нет способа влиять на символы в s2. Каждый объект содержит свою собственную копию stck и tos. Это фундаментальная для понимания объектов концепция. Хотя все объекты класса имеют общие функции-члены, каждый объект создает и поддерживает *свои собственные данные.*

1. Введите и выполните программы, приведенные в этом разделе, если это еще не сделано.
2. Создайте класс card, который поддерживает каталог библиотечных карточек. Этот класс должен хранить заглавие книги, имя автора и выданное на руки число экземпляров книги. Заглавие и имя автора храните в виде строки символов, а количество экземпляров — в виде целого числа. Используйте открытую функцию-член **store()** для запоминания информации о книгах и открытую функцию-член **show()** для вывода информации на экран. В функцию **main()** включите краткую демонстрацию работы созданного класса.
3. Создайте класс с циклической очередью целых. Сделайте очередь длиной 100 целых. В функцию main() включите краткую демонстрацию ее работы.

1.6. Некоторые отличия языков C и C++

У языка C++ есть ряд небольших отличий от C. Хотя каждое из этих отличий само по себе незначительно, вместе они достаточно распространены в программах C++. Поэтому перед тем как двинуться дальше, обсудим эти отличия.

а Во-первых, если в C функция не имеет параметров, ее прототип содержит слово **void** в списке параметров функции. Например, если в C функ-

ция f1() не имеет параметров (и возвращает **char),** ее прототип будет выглядеть следующим образом:

```
char f1(void);
```

В C++ слово **void** не обязательно. Поэтому в C++ прототип обычно пишется так:

```
char f1();
```

C++ отличается от C способом задания пустого списка параметров. Если бы предыдущий прототип имел место в программе C, то это бы означало, что о параметрах функции сказать *ничего нельзя.* А в C++ это означает, что у функции *нет* параметров. Поэтому в предыдущих примерах для исчерпывающего обозначения пустого списка параметров слово **void** не использовалось. (Использование **void** для обозначения пустого списка параметров не ошибочно, скорее, оно излишне. Поскольку большинство программистов C++ гонятся за эффективностью с почти религиозным рвением, вы никогда не увидите **void** в таких случаях.) Запомните, в C++ следующие два объявления эквивалентны:

```
int f1();
int f1(void);
```

- Другим небольшим отличием между C и C++ является то, что в программах C++ все функции должны иметь прототипы. Запомните, в C прототипы функций рекомендуются, но технически они не обязательны, а в C++ прототипы необходимы. Как показывают примеры из предыдущего раздела, содержащийся в классе прототип функции-члена действует так же, как ее обычный прототип, и никакого иного прототипа не требуется.

- Третьим отличием между C и C++ является то, что если в C++ функция имеет отличный от **void** тип возвращаемого значения, то инструкция **return** внутри этой функции должна содержать значение данного типа. В языке C функции с отличным от **void** типом возвращаемого значения фактически не требуется возвращать что-либо. Если значения нет, то функция возвращает неопределенное значение.

  В C, если тип возвращаемого функцией значения явно не задан, функция по умолчанию возвращает значение целого типа. В C++ такого правила нет. Следовательно, необходимо явно объявлять тип возвращаемого значения всех функций.

- Следующим отличием между C и C++ является то, что в программах C++ вы можете выбирать место для объявления локальных переменных. В C локальные переменные могут объявляться только в начале блока, перед любой инструкцией "действия". В C++ локальные переменные могут

объявляться в любом месте программы. Одним из преимуществ такого подхода является то, что локальные переменные для предотвращения нежелательных побочных эффектов можно объявлять рядом с местом их первого использования.

а И последнее. Для хранения значений булева типа (истина или ложь) в C++ определен тип данных **bool**. В C++ также определены ключевые слова **true и false** — единственные значения, которыми могут быть данные типа **bool.** В C++ результатом выполнения операторов отношения и логических операторов являются значения типа **bool,** и направление развития любой условной инструкции должно определяться относительно значения типа **bool.** Хотя такое отличие от C на первый взгляд кажется значительным, на самом деле это не так. Фактически оно совершенно прозрачно и вот почему: как вы знаете, в C любое ненулевое значение является истинным, а нулевое — ложным. В C++ это положение сохраняется, поскольку при использовании таких значений в булевом выражении ненулевое значение автоматически преобразуется в **true,** а нулевое — в **false.** Правильно и обратное: **true** преобразуется в 1, а **false** в 0, если значение типа **bool** оказывается в целом выражении. Добавление в C++ данных типа **bool** усиливает контроль типа и дает возможность различать данные булева и целого типов. Естественно, что использование данных булева типа не обязательно, скорее оно просто удобно.

1. В программах C, при отсутствии в командной строке аргументов, функция **main()** обычно объявляется так:

```
int  main(void)
```

Однако в C++ такое использование слова **void** избыточно и необязательно.

2. Эта короткая программа C++ не будет компилироваться, поскольку у функции **sum()** нет прототипа:

```
// Эта программа не будет компилироваться
ttinclude <iostream>
using namespace std;

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

    cout << "Введите два числа: ";
    cin>>a>>b;
    c=sum(a, b);
    cout << "Сумма равна:" << c;
```

```
    return 0;
}

// Этой функции необходим прототип
sum(int a, int b)
{
    return a+b;
}
```

3. Эта короткая программа иллюстрирует тот факт, что локальные переменные можно объявить в любом месте блока:

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

int main()
{
    int i; // локальная переменная, объявленная в начале блока

    cout << "Введите число: " ;
    cin>>i;

    // расчет факториала
    int j, fact=1; // переменные, объявленные перед инструкциями
                   // действия

    for (j=i; j>=1; j--) fact=fact * j;

    cout << "Факториал равен:" << fact;

    return 0;
}
```

Хотя объявление переменных **j** и **fact** рядом с местом их первого использования в этом коротком примере и не слишком впечатляет, в больших функциях такая возможность может обеспечить программе ясность и предотвратить нежелательные побочные эффекты.

4. В следующей программе создается булева переменная **outcome** и ей присваивается значение **false.** Затем эта переменная используется в инструкции **if.**

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

int main()
{
    bool outcome;

    outcome = false;

    if(outcome) cout << "истина";
    else cout << "ложь";
```

```
    return 0;
}
```

Как и следовало ожидать, в результате выполнения программы на экране появляется слово **ложь.**

1. Следующая программа не будет компилироваться в качестве программы C++. Почему?

```
// В этой программе есть ошибка
#include <iostream>
using namespace std;

int main()
{
    f 0;

    return 0;
}

    void f()
{
    cout << "Программа работать не будет";
}
```

2. Попытайтесь объявлять локальные переменные в различных местах программы C++. Попытайтесь проделать то же с программой C, обращая внимание на то, какие объявления вызовут сообщения об ошибках.

1.7. Введение в перегрузку функций

После классов, вероятно, следующей важной и необычной возможностью C++ является *перегрузка функций (function overloading)*. Перегрузка функций не только обеспечивает механизм, посредством которого в C++ достигается один из типов полиморфизма, она также формирует то ядро, вокруг которого развивается вся среда программирования на C++. Ввиду важности темы в данном разделе предлагается только предварительное знакомство с перегрузкой функций, которой посвящена целая глава этой книги.

В C++ две или более функции могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим. Если две или более функции имеют одинаковое имя, говорят, что они *перегружены.*

Перегруженные функции позволяют упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Перегрузить функцию очень легко: просто объявите и определите все требуемые варианты. Компилятор автоматически выберет правильный вариант вызова на основании числа и/или типа используемых в функции аргументов.

S C++ можно также перегружать и операторы. Однако для того чтобы понять перегрузку операторов, необходимо больше узнать о C++.

1. Одно из основных применений перегрузки функций — это достижение полиморфизма при компиляции программ, который воплощает в себе философию — один интерфейс, множество методов. Как вы знаете, при программировании на C необходимо иметь определенное число близких по назначению функций, отличающихся только типом данных, с которыми они работают. Классический пример этой ситуации дает набор библиотечных функций C. Как ранее упоминалось в этой главе, библиотека содержит функции **abs()**, **labs()** и **fabs()**, которые возвращают абсолютное значение, соответственно, целого, длинного целого и числа с плавающей точкой. Однако из-за того, что для трех типов данных требуется три типа функции, ситуация выглядит более сложной, чем это необходимо. Во всех трех случаях возвращается абсолютная величина числа, отличие только в типе данных. В то же время, программируя на C++, вы можете исправить эту ситуацию путем перегрузки одного имени для трех типов данных так, как показано в следующем примере:

```
ttinclude<iostream>
using namespace std;

// Перегрузка abs() тремя способами
int abs(int n);
long abs(long n);
double abs(double n);

int main()
{
    cout << "Абсолютная величина -10:" << abs (-10) << "\n/n";
    cout << "Абсолютная величина -10L:" << abs(-10L) << "\n/n";
    cout << "Абсолютная величина -10.01:" << abs(-10.01) << "\n/n";

    return 0;
}
```

```
// abs() для целых
int abs(int n)
{
    cout << "В целом abs()\n";
    return n<0 ? -n: n;
}

// abs() для длинных целых
long abs(long n)
{
    cout << "В длинном целом abs() \n";
    return n<0 ? -n: n;
}

// abs() для вещественных двойной точности
double abs(double n)
{
    cout << "В вещественном abs() двойной точности\n";
    return n<0 ? -n: n;
}
```

Как можно заметить, в программе задано три функции abs(), своя для каждого типа данных. Внутри **main()** функция **abs()** вызывается с тремя аргументами разных типов. Компилятор автоматически вызывает правильную версию abs(), основываясь на используемом в аргументе типе данных. В результате работы программы на экран выводится следующее:

```
В целом abs()
Абсолютная величина -10: 10

В длинном целом abs()
Абсолютная величина -10L: 10

В вещественном abs() двойной точности
Абсолютная величина -10.01: 10.01
```

Хотя этот пример достаточно прост, ценность перегрузки функций он все же демонстрирует. Поскольку одно имя используется для описания основного набора действий, искусственная сложность, вызванная тремя слабо различающимися именами, в данном случае **abs(), labs() и fabs(),** устраняется. Теперь вам необходимо помнить только одно имя — то, которое описывает *общее* действие. На компилятор возлагается задача выбора соответствующей *конкретной* версии вызываемой функции (а значит и метода обработки данных). Это имеет лавинообразный эффект в вопросе снижения сложности программ. В данном случае, благодаря использованию полиморфизма, из трех имен получилось одно.

Хотя использование полиморфизма в этом примере довольно тривиально, вы, должно быть, уже поняли, что для очень больших программ подход "один интерфейс, множество методов" может быть очень эффективным.

2. Ниже приведен другой пример перегрузки функций. В этом случае функция date() перегружается для получения даты либо в виде строки, либо в виде трех целых. В обоих этих случаях функция выводит на экран переданные ей данные.

```
ttinclude<iostream>
using namespace std;

void date(char *date) ; // дата в виде строки
void date(int month, int day, int year); // дата в виде чисел

int main()
{
    date("8/23/99");
    date( 8, 23, 99);

    return 0;
}

// Дата в виде строки
void date(char *date)
{
    cout << "Дата:" << date << "\n";
}

// Дата в виде целых
void date(int month, int day, int year)
{
    cout << "Дата:" << month << "/";
    cout << day << "/" << year << "\n";
}
```

Этот пример показывает, как перегрузка функций может обеспечить для функции более понятный интерфейс. Поскольку дату очень естественно представлять либо в виде строки, либо в виде трех целых чисел, содержащих месяц, день и год, нужно просто выбрать наиболее подходящую версию в соответствии с ситуацией.

3. До сих пор мы рассматривали перегруженные функции, отличающиеся типом своих аргументов. Однако перегруженные функции могут также отличаться и числом аргументов, как показано в приведенном ниже примере:

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

void f1(int a) ;
void f1(int a, int b) ;

int main()
{
    f1(10);
    f1(10, 20);
```

```
    return 0;
}
void f1(int a)
{
    cout << "В f1(int a) \n";
}
void f1(int a, int b)
{
    cout << "В f1(int a, int b) \n";
}
```

4. Важно понимать, что тип возвращаемого значения сам по себе еще не является достаточным отличием для перегрузки функции. Если две функции отличаются только типом возвращаемых данных, компилятор не всегда сможет выбрать нужную. Например, следующий фрагмент неправилен, поскольку в нем имеет место избыточность:

```
// Это все неправильно и не будет компилироваться
int f1(int a);
double f1(int a);
.
.
.
f1(10); // какую функцию выбрать компилятору???
```

Как написано в комментарии, у компилятора нет способа выяснить, какую версию **f1()** вызвать.

Упражнения

1. Создайте функцию. **sroot()**, которая возвращает квадратный корень своего аргумента. Перегрузите **sroot()** тремя способами: чтобы получить квадратный корень целого, длинного целого и числа с плавающей точкой двойной точности. (Для непосредственного подсчета квадратного корня вы можете использовать стандартную библиотечную функцию **sqrt()**.)
2. Стандартная библиотека C++ содержит три функции:

```
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
```

Эти функции возвращают численное значение, содержащееся в строке, на которую указывает s. Заметьте, что **atof()** возвращает **double, atoi** возвращает **int** и **atol** возвращает **long.** Почему нельзя перегрузить эти функции?

3. Создайте функцию **min()**, которая возвращает наименьший из двух численных аргументов, используемых при вызове функции. Перегрузите функцию

min() так, чтобы она воспринимала в качестве аргументов символы, целые и действительные двойной точности.

4. Создайте функцию slccp(), приостанавливающую работу компьютера на столько секунд, сколько указано в аргументе функции. Перегрузите **sleep()** так, чтобы она могла вызываться или с целым, или со строкой, задающей целое. Например, оба этих вызова должны заставить компьютер остановиться на 10 секунд:

```
sleep(10);
sleep("10");
```

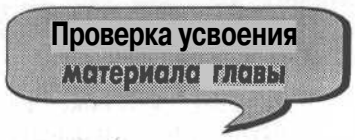
Продемонстрируйте работу ваших функций, включив их в короткую программу. (Почувствуйте удобство их применения для задания паузы в работе компьютера.)

1.8. Ключевые слова C++

В C++ поддерживаются все ключевые слова C и кроме этого еще 30 ключевых слов, которые относятся только к языку C++. Все определенные для C++ ключевые слова представлены в табл. 1.1. Кроме них в ранних версиях C++ было определено ключевое слово **overload,** которое сейчас считается устаревшим.

Таблица 1.1. *Ключевые слова C++*

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



Попытайтесь выполнить следующие упражнения и ответить на вопросы.

1. Дайте краткие определения полиморфизма, инкапсуляции и наследования.
2. Как включить в программу C++ комментарии?

3. Напишите программу, использующую стиль ввода/вывода C++, для ввода двух целых с клавиатуры и затем вывода на экран результата возведения первого в степень второго. (Например, пользователь вводит 2 и 4, тогда результатом будет 24, или 16.)

4. Создайте функцию **rev_str()** для изменения порядка следования символов строки на обратный. Перегрузите **rev_str()** так, чтобы она могла вызываться с одним или двумя символьными строками. Если функция вызывается с одной строкой, то операция должна осуществляться с ней. Если она вызывается с двумя строками, то результирующая строка должна оказаться во втором аргументе. Например:

```
char s1[80], s2[80];
    strcpy(s1, "привет");
    rev_str(s1, s2) ; // измененная строка оказывается в s2,
                      // s1 не меняется
    rev_str(s1);      // измененная строка возвращается в s1
```

5. Дана следующая программа, написанная в соответствии с новым стилем программирования на C++. Покажите, как превратить ее в программу старого стиля.

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

int f(int a);

int main()
{
    cout << f(10);

    return 0;
}

int f(int a)
{
    return a * 3.1416;
}
```

6. Что представляют собой данные типа **bool**?

Глава 2

Введение в классы

В этой главе вводятся понятия классов и объектов. В следующих нескольких важнейших разделах фактически описаны почти все аспекты программирования на C++, поэтому советуем вам читать повнимательнее.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Напишите программу, использующую стиль ввода/вывода C++ для ввода строки и затем вывода на экран ее длины.
2. Создайте класс, который содержит информацию об имени и адресе. Храните всю эту информацию в символьных строках закрытой части класса. Включите в класс открытую функцию для запоминания имени и адреса. Также включите открытую функцию, которая выводит эти имя и адрес на экран. (Назовите эти функции **store()** и **display()**.)
3. Создайте перегружаемую функцию **rotate(),** которая циклически сдвигает влево свой аргумент и возвращает результат. Перегрузите ее так, чтобы она работала с целыми **(int)** и длинными целыми **(long).** (Сдвиг по кольцу аналогичен обычному сдвигу, за исключением того, что выдвигаемый с одного конца слова бит появляется на другом его конце.)
4. Что неправильно в следующем фрагменте?

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

class myclass {
    int i;
public:
.
.
.
};
```

```
int main()
{
    myclass ob;
    ob.i = 10;
.
.
.
}
```

2.1. Конструкторы и деструкторы

Если вы писали очень длинные программы, то знаете, что в некоторых частях программы обычно требуется инициализация. Необходимость в инициализации еще более часто проявляется при работе с объектами. Действительно, если обратиться к реальным проблемам, то, фактически, для каждого создаваемого вами объекта требуется какого-то вида инициализация. Для разрешения этой проблемы в C++ имеется *функция-конструктор (constructor function),* включаемая в описание класса. Конструктор класса вызывается всякий раз при создании объекта этого класса. Таким образом, любая необходимая объекту инициализация при наличии конструктора выполняется автоматически.

Конструктор имеет то же имя, что и класс, частью которого он является, и не имеет возвращаемого значения. Например, ниже представлен небольшой класс с конструктором:

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

class myclass {
    int a;
public:
    myclass(); // конструктор
    void show();
};

myclass::myclass()
{
    cout << "В конструкторе \n";
    a=10;
}
void myclass::show()
{
    cout << a;
}
```

```
int main()
{
    myclass ob;

    ob.show();

    return 0;
}
```

В этом простом примере значение a инициализируется конструктором **myclass().** Конструктор вызывается тогда, когда создается объект ob. Объект, в свою очередь, создается при выполнении инструкции объявления объекта. Важно понимать, что в C++ инструкция объявления переменной является "инструкцией действия". При программировании на C инструкции объявления переменных понимаются просто как создание переменных. Однако в C++, поскольку объект может иметь конструктор, инструкция объявления переменной может вызывать выполнение записанных в конструкторе действий.

Обратите внимание, как определяется конструктор **myclass()**. Как уже говорилось, он не имеет возвращаемого значения. В соответствии с формальными правилами синтаксиса C++ конструктор не должен иметь возвращаемого значения.

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

Функцией, обратной конструктору, является *деструктор (destructor)*. Эта функция вызывается при удалении объекта. Обычно при работе с объектом в момент его удаления должны выполняться некоторые действия. Например, при создании объекта для него выделяется память, которую необходимо освободить при его удалении. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале. Пример класса с деструктором:

```
tfinclude<iostream>
using namespace std;

class myclass {
    int a;
public:
    myclass(); // конструктор
    ~myclass(); // деструктор
    void show();
};

myclass::myclass()
{
    cout << "Содержимое конструктора\n";
    a = 10;
}
```

```
myclass::~myclass()
{
    cout << "Удаление...\n";
}

void myclass::show()
{
    cout << a << "\n";
}
int main()
{
    myclass ob;

    ob.show();

    return 0;
}
```

Деструктор класса вызывается при удалении объекта. Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы.

Адреса конструктора и деструктора получить невозможно.

Фактически как конструктор, так и деструктор могут выполнить любой тип операции. Тем не менее считается, что код внутри этих функций не должен делать ничего, не имеющего отношения к инициализации или возвращению объектов в исходное состояние. Например, конструктор в предшествующем примере мог бы рассчитать число pi с точностью до 100 знаков после запятой. Однако применение конструктора или деструктора для действий, прямо не связанных с инициализацией, является очень плохим стилем программирования и его следует избегать.

1. Вспомните, что в созданном в главе 1 классе **stack** для установки переменной индекса стека требовалась функция **инициализации**. Это именно тот тип **действия,** для выполнения которого и придуман конструктор. Здесь представлена улучшенная версия класса **stack,** где для автоматической инициализации объекта стека при его создании используется конструктор:

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

ttdefine SIZE 10
```

```
// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); // конструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека \n";
    tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон";
        return;
    }
    stck[tos]=ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // образование двух автоматически инициализируемых стеков
    stack s1, s2;
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
```

```
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "символ из s1:" << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "символ из s2:" << s2.pop() << "\n";

    return 0;
}
```

Обратите внимание, что теперь вместо отдельной, специально вызываемой программой функции задача инициализации выполняется конструктором автоматически. Это важное положение. Если инициализация выполняется автоматически при создании объекта, то это исключает любую возможность того, что по ошибке инициализация не будет выполнена. Вам, как программисту, не нужно беспокоиться об инициализации — она автоматически выполнится при появлении объекта.

2. В следующем примере показана необходимость не только конструктора, но и деструктора. В примере создается простой класс для строк, который содержит саму строку и ее длину. Когда создается объект **strtype**, для хранения строки выщеляется память, и начальная длина строки устанавливается равной нулю. Когда объект **strtype** удаляется, эта память освобождается.

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

#define SIZE 255

class strtype {
    char *p;
    int len;
public:
    strtype(); // конструктор
    ~strtype(); // деструктор
    void set(char *ptr);
    voidshow();
};

// Инициализация объекта строка
strtype::strtype()
{
    p=(char *) malloc(SIZE);
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
```

```
    *p='\0';
    len=0;
}

// Освобождение памяти при удалении объекта строка
strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::set(char *ptr)
{
    if(strlen(p) >=SIZE) {
        cout << "Строка слишком велика\n";
        return;
    }
    strcpy(p, ptr);
    len=strlen(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1,s2;

    s1.set("Это проверка");
    s2.set("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

В этой программе для выделения и освобождения памяти используются функции **malloc()** и **free().** Хотя этот пример совершенно правилен, как вы увидите далее в этой книге, в C++ есть и иной путь управления распределением динамической памяти.

В предыдущей программе заголовочные файлы библиотечных функций языка С оформлены в соответствии с новым стилем оформления заголовков. Как уже упоминалось в главе 1, если ваш компилятор не поддерживает такие за-

головки, просто замените их стандартными заголовочными файлами С. То же самое можно сказать о всех программах этой книги, в которых используются библиотечные функции С.

3. В следующем примере приведен интересный способ использования конструктора и деструктора объекта. В программе объект класса **timer** предназначен для измерения временного интервала между его созданием и удалением. При вызове **деструктора** на экран выводится прошедшее с момента создания объекта время. Вы могли бы воспользоваться подобным объектом для измерения времени работы программы или времени работы функции внутри блока. Просто убедитесь, что объект исчезает в момент завершения временного интервала.

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

class timer {
    clock_t start;
public:
    timer(); // конструктор
    ~timer(); // деструктор
};

timer::timer()
{
    start=clock();
}

timer::~timer()
{
    clock_t end;

    end=clock();
  cout << "Затраченное время:" << (end-start)/CLOCKS_PER_SEC <<"\n";
)

int main()
{
    timer ob;
    char c;

    // Пауза . . .
    cout << "Нажмите любую клавишу, затем ENTER: ";
    cin >> c;

    return 0;
}
```

В программе используется стандартная библиотечная функция **clock()**, которая возвращает число временных циклов с момента запуска программы. Если разделить это число на **CLOCKS_PER_SEC**, можно получить значение в секундах.

1. Измените класс **queue** (см. упражнения гл. 1) так, чтобы провести инициализацию с помощью конструктора.
2. Создайте класс **stopwatch** для имитации секундомера. Используйте конструктор для начальной установки секундомера в 0. Образуйте две функции-члена **start()** и **stop()** соответственно для запуска и остановки секундомера. Включите в класс и функцию-член **show()** для вывода на экран величины истекшего промежутка времени. Также используйте деструктор для автоматического вывода на экран времени, прошедшего с момента создания объекта класса **stopwatch,** до его удаления. (Для простоты время приведите в секундах.)
3. Что неправильно в конструкторе, показанном в следующем фрагменте?

```
class sample {
    double a, b, c;
public:
    double sample(); // ошибка, почему?
};
```

2.2. Конструкторы с параметрами

Конструктору можно передавать аргументы. Для этого просто добавьте необходимые параметры в объявление и определение конструктора. Затем при объявлении объекта задайте параметры в качестве аргументов. Чтобы понять, как это делается, начнем с короткого примера:

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

class myclass {
    int a;
public:
    myclass(int x); // конструктор
    void show();
};

myclass::myclass(int x)
{
    cout << "В конструкторе\n";
    a = x;
}

void myclass::show()
{
    cout << a << "\n";
}
```

```
int main()
{
    myclass ob(4);

    ob.show();

    return 0;
}
```

Здесь конструктор класса **myclass** имеет один параметр. Значение, передаваемое в **myclass()**, используется для инициализации переменной a. Обратите особое внимание на то, как в функции **main()** объявляется объект **ob.** Число 4, записанное в круглых скобках, является аргументом, передаваемым параметру x конструктора myclass(), который используется для инициализации переменной a.

Фактически синтаксис передачи аргумента конструктору с параметром является сокращенной формой записи следующего, более длинного выражения:

```
myclass ob = myclass (4);
```

Однако большинство программистов C++ пользуются сокращенной формой записи. На самом деле, с технической точки зрения между этими двумя формами записи имеется небольшое отличие, связанное с конструктором копий (copy constructor), о котором будет рассказано позже. На данном этапе об этом отличии можно не беспокоиться.

6 отличие от конструктора деструктор не может иметь параметров. Смысл этого понять достаточно просто: отсутствует механизм передачи аргументов удаленному объекту.

1. Вполне допустимо передавать конструктору несколько аргументов. В этом примере конструктору **myclass()** передается два аргумента:

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

class myclass {
    int a, b;
public:
    myclass(int x, int y); // конструктор
    void show();
};
```

```
myclass::myclass(int x, int y)
{
    cout << "В конструкторе\n";
    a = x;
    b = y;
}

void myclass::show()
{
    cout << a << ' ' << b << "\n";
}

int main()
{
    myclass ob(4, 7);

    ob.show();

    return 0;
}
```

Здесь 4 передается в x, а 7 передается в y. Такой же общий подход используется для передачи любого необходимого числа аргументов (ограниченного, разумеется, возможностями компилятора).

2. Здесь представлена следующая версия класса **stack,** в котором конструктор с параметром используется для присвоения стеку "имени". Это односимвольное имя необходимо для идентификации стека в случае возникновения ошибки.

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

#define SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
    char who; // идентифицирует стек
public:
    stack(char c); // конструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};

// Инициализация стека
stack::stack(char c)
{
    tos = 0;
    who = c;
```

```
    cout << "Работа конструктора стека " << who << "\n";
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos=SIZE) {
        cout << "Стек " << who << " полон \n";
        return;
    }
    stck[tos]=ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек " << who << " пуст ";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // образование двух автоматически инициализируемых стеков
    stack s1('A'), s2('B');
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    // Это вызовет сообщения об ошибках
    for(i=0; i<5; i++) cout << "символ из стека s1: "
                                    << s1.pop() << "\n";
    for(i=0; i<5; i++) cout << "символ из стека s2: "
                                    << s2.pop() << "\n";

    return 0;
}
```

Присвоение "имени" объекту, как показано в примере, является особенно полезным при отладке, когда важно выяснить, какой из объектов вызывает ошибку.

3. Здесь показан новый вариант разработанного ранее класса **strtype,** в котором используется конструктор с параметром:

```
ttinclude<iostream>
tinclude <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype();
    voidshow();
};

strtype::strtype(char*ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len + 1);
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << "- длина: " << len;
    cout << "\n";
}

intmain()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

В этой версии класса **strtype** строка получает свое начальное значение с помощью конструктора.

4. Конструктору объекта можно передать не только константы, но и любые допустимые выражения с переменными. Например, в следующей программе для создания объекта используется пользовательский ввод:

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

class myclass {
    int i, j;
public:
    myclass(int a, int b);
    void show();
};

myclass::myclass(int a, int b)
{
    i = a;
    j = b;
}

void myclass::show()
{
    cout << i << ' ' << j << "\n";
}

int main()
{
    int x, y;

    cout << "Введите два целых: ";
    cin >> x >> y;

    // использование переменных для создания ob
    myclass ob(x, y);

    ob.show();

    return 0;
}
```

Эта программа иллюстрирует важное свойство объектов. Объекты могут создаваться по мере необходимости, точно в соответствии с возникающей в момент их создания ситуацией. Когда вы побольше узнаете о C++, вы увидите, какой полезной является возможность создания объектов "на ходу".

Упражнения

1. Измените класс **stack** так, чтобы память для стека выделялась динамически. При этом длина стека должна задаваться параметром конструктора. (Не забудьте освободить эту память с помощью деструктора.)

2. Создайте класс **t_and_d**, которому при его создании передается текущее системное время и дата в виде параметров конструктора. Этот класс должен включать в себя функцию-член, выводящую время и дату на экран. (Подсказка: Для нахождения и вывода на экран этих данных воспользуйтесь стандартной библиотечной функцией времени и даты.)
3. Создайте класс **box,** конструктору которого передаются три значения типа **double,** представляющие собой длины сторон параллелепипеда. Класс **box** должен подсчитывать его объем и хранить результат также в виде значения типа **double.** Включите в класс функцию-член **vol(),** которая будет выводить на экран объем любого объекта типа **box.**

2.3. Введение в наследование

Хотя более полно наследование (inheritance) обсуждается в главе 7, предварительно с ним необходимо познакомиться уже сейчас. Применительно к C++ наследование — это механизм, посредством которого один класс может наследовать свойства другого. Наследование позволяет строить иерархию классов, переходя от более общих к более специальным.

Для начала необходимо определить два термина, обычно используемые при обсуждении наследования. Когда один класс наследуется другим, класс, который наследуется, называют *базовым классом (base class).* Наследующий класс называют *производным классом (derived class).* Обычно процесс наследования начинается с задания базового класса. Базовый класс определяет все те качества, которые будут общими для всех производных от него классов. В сущности, базовый класс представляет собой наиболее общее описание ряда характерных черт. Производный класс наследует эти общие черты и добавляет свойства, характерные только для него.

Чтобы понять, как один класс может наследовать другой, давайте начнем с примера, который, несмотря на свою простоту, иллюстрирует несколько ключевых положений наследования.

Для начала — объявление базового класса:

```
// Определение базового класса
class B {
    int i;
public:
    void set_i(int n);
    int get_i();
};
```

Теперь объявим производный класс, наследующий этот базовый:

```
// Определение производного класса
class D: public B {
    int j;
```

```
public:
    void set_j (intn) ;
    intmul();
};
```

Внимательно посмотрите на это объявление. Обратите внимание, что после имени класса D имеется двоеточие, за которым следует ключевое слово **public** и имя класса B. Для компилятора это указание на то, что класс D будет наследовать все компоненты класса B. Само ключевое слово **public** информирует компилятор о том, что, поскольку класс B будет наследоваться, значит, все открытые элементы базового класса будут также открытыми элементами производного класса. Однако все закрытые элементы базового класса останутся закрытыми и к ним не будет прямого доступа из производного класса.

Ниже приводится законченная программа, в которой используются классы **B** и **D**:

```
// Простой пример наследования
#include <iostream>
using namespace std;

// Определение базового класса
class B {
    int i;
public:
    void set_i(int n) ;
    intget_i();
};

// Определение производного класса
class D: public B {
    int j;
public:
    void set_j (intn) ;
    intmul();
};

// Задание значения i в базовом классе
void B::set_i(int n)
{
    i = n;
}

// Возвращение значения i в базовом классе
int B: :get_i()
{
    return i;
}
```

```
// Задание значения j в производном классе
void D::set_j(int n)
{
    j = n;
}

// Возвращение значения i базового класса и j — производного
int D::mul()
{
    // производный класс может
    // вызывать функции-члены базового класса
    return j * get_i();
}

int main()
{
    D ob;

    ob.set_i(10); // загрузка i в базовый класс
    ob.set_j(4); // загрузка j в производный класс

    cout << ob.mul(); // вывод числа 40

    return 0;
}
```

Обратите внимание на определение функции mul(). Отметьте, что функция get_i(), которая является членом базового класса B, а не производного D, вызывается внутри класса D без всякой связи с каким бы то ни было объектом. Это возможно потому, что открытые члены класса B становятся открытыми членами класса D. В функции mul() вместо прямого доступа к **i**, необходимо вызывать функцию get_i(), поскольку закрытые члены базового класса (в данном случае i) остаются закрытыми для нее и недоступными из любого производного класса. Причина, по которой закрытые члены класса становятся недоступными для производных классов — поддержка инкапсуляции. Если бы закрытые члены класса становились открытыми просто посредством наследования этого класса, инкапсуляция была бы совершенно несостоятельна.

Здесь показана основная форма наследования базового класса:

```
class имя_производного_класса: с_д имя_базового_класса {
.
.
.
};
```

Здесь *с_д* (спецификатор доступа) — это одно из следующих трех ключевых слов: **public (открытый), private (закрытый) или protected (защищенный).**

В данном случае для наследования класса используется именно **public**. Полное описание этих спецификаторов доступа будет дано позже.

1. Ниже приведена программа, которая определяет общий базовый класс **fruit,** описывающий некоторые характеристики фруктов. Этот класс наследуется двумя производными классами **Apple и Orange.** Эти классы содержат специальную информацию о конкретном фрукте (яблоке или апельсине).

```
// Пример наследования классов
#include <iostream>
#include <cstring>
using namespace std;

enum yn {no, yes};
enum color {red, yellow, green, orange};

void out(enum yn x);

char *c[ ] = {
    "red", "yellow", "green", "orange"};

// Родовой класс фруктов
class fruit {
// В этом базовом классе все элементы открыты
public:
    enum yn annual;
    enum yn perennial;
    enum yn tree;
    enum yn tropical;
    enum color clr;
    char name[40];
};

// Производный класс яблок
class Apple: public fruit {
    enum yn cooking;
    enum yn crunchy;
    enum yn eating;
public:
    void seta(char *n, enum color c, enum yn ck, enum yn crchy,
enum yn e);
    void show();
};

// Производный класс апельсинов
class Orange: public fruit {
    enum yn juice;
```

```
    enum yn sour;
    enum yn eating;
public:
    void seto(char *n, enum color c, enum yn j, enum yn sr, enum yn e) ;
    void show();
};

void Apple: : seta(char *n, enum color c, enum yn ck, enum yn crchy,
enum yn e)
{
    strcpy(name,n) ;
    annual = no;
    perennial = yes;
    tree = yes;
    tropical = no;
    clr = c;
    cooking = ck;
    crunchy = crchy;
    eating = e;
}

void Orange: : seto ( char *n, enum color c, enum yn j, enum yn sr,
enum yn e)
{
    strcpy(name,n) ;
    annual = no;
    perennial = yes;
    tree = yes;
    tropical = yes;
    clr = c;
    juice = j;
    sour = sr;
    eating = e;
}

voidApple::show()
{
    cout << name << " яблоко — это: " << "\n";
    cout << "Однолетнее растение: "; out(annual);
    cout << "Многолетнее растение: "; out(perennial);
    cout << "Дерево: "; out(tree);
    cout << "Тропическое: "; out(tropical);
    cout << "Цвет: " << c[clr] << "\n";
    cout << "Легко приготавливается: "; out(cooking);
    cout << "Хрустит на зубах: "; out(crunchy);
    cout << "Съедобное: "; out(eating);
    cout << "\n";
}
```

```
void Orange::show()
{
    cout << name << " апельсин - это: " << "\n";
    cout << "Однолетнее растение: "; out(annual);
    cout << "Многолетнее растение: "; out(perennial);
    cout << "Дерево: "; out(tree);
    cout << "Тропическое: "; out(tropical);
    cout << "Цвет: " << c[clr] << "\n";
    cout << "Годится для приготовления сока: "; out(juice);
    cout << "Кислый: "; out(sour);
    cout << "Съедобный: "; out(eating);
    cout << "\n";
}

void out(enum yn x)
{
    if (x==no) cout << "нет\n";
    else cout << "да\n";
}

int main()
{
    Apple a1, a2;
    Orange o1, o2;

    a1.seta("Красная прелесть", red, no, yes, yes);
    a2.seta("Джонатан", red, yes, no, yes);

    o1.seto("Пуп", orange, no, no, yes);
    o2.seto("Валенсия", orange, yes, yes, no);

    a1.show();
    a2.show();

    o1.show();
    o2.show();

    return 0;
}
```

Как можно заметить, базовый класс **fruit** определяет несколько свойств, характерных для фруктов любого типа. (Конечно, чтобы сократить пример и таким образом приспособить его для книги, класс **fruit** отчасти упрощен.) Например, все фрукты растут на однолетних или многолетних растениях. Все фрукты растут на деревьях или на растениях другого типа, таких как лоза или куст. Все фрукты имеют цвет и название. Затем такой базовый класс наследуется классами **Apple и Orange.** Каждый из этих классов обеспечивает объект информацией, характерной для фруктов конкретного типа.

Этот пример иллюстрирует основной смысл наследования. Создаваемый здесь базовый класс определяет основные черты, связанные со *всеми* фруктами.

Производным классам предоставляется возможность обеспечения тех характеристик, которые являются характерными в каждом *конкретном* случае.

Эта программа раскрывает другой важный аспект наследования: производные классы не "владеют" базовым классом безраздельно. Он может наследоваться любым количеством классов.

Упражнения

1. Дан следующий базовый класс:

```
class area_cl {
public:
    double height;
    double width;
};
```

создайте два производных класса **rectangle и isosceles,** которые наследуют базовый класс **area_c1.** Каждый класс должен включать в себя функцию area(), которая возвращает площадь соответственно прямоугольника (rectangle) и равнобедренного треугольника (isosceles). Для инициализации переменных **height и width** (высота и длина основания, соответственно) используйте конструктор с параметрами.

2.4. Указатели на объекты

До сих пор вы осуществляли доступ к членам объекта с помощью оператора точка (.). Если вы работаете с объектом, то это правильно. Однако доступ к члену объекта можно получить также и через указатель на этот объект. В этом случае обычно применяется оператор стрелка (->). (Аналогичным способом оператор стрелка (->) используется при работе с указателем на структуру.)

Вы объявляете указатель на объект точно так же, как и указатель на переменную любого другого типа. Задайте имя класса этого объекта, а затем имя переменной со звездочкой перед ним. Для получения адреса объекта перед ним необходим оператор &, точно так же, как это делается для получения адреса переменной другого типа.

Как и для любого другого указателя, если вы инкрементируете указатель на объект, он будет указывать на следующий объект такого же типа.

Примеры

1. Простой пример использования указателя на объект:

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

```
class myclass {
    int a;
public:
    myclass(int x); // конструктор
    int get();
};

myclass::myclass(int x)
{
    a = x;
}

int myclass::get()
{
    return a;
}

int main()
{
    myclass ob(120); // создание объекта
    myclass *p; // создание указателя на объект

    p = &ob; // передача адреса ob в p

    cout << "Значение, получаемое через объект:" << ob.get();
    cout << "\n";

    cout << "Значение, получаемое через указатель:" << p->get();

    return 0;
}
```

Отметьте, как объявление

```
myclass *p;
```

создает указатель на объект класса **myclass.** Это важно для понимания следующего положения: создание указателя на объект *не* создает объекта, оно создает только указатель на него.

Для передачи адреса объекта ob в переменную p, использовалась инструкция:

```
p = &ob;
```

И последнее, в программе показано, как можно получить доступ к членам объекта с помощью указателя на этот объект.

Мы вернемся к обсуждению указателей на объекты в главе 4, когда у вас уже будет более полное представление о C++.

2.5. Классы, структуры и объединения

Как вы уже заметили, синтаксически класс похож на структуру. Вас, возможно, удивило то, что класс и структура имеют фактически одинаковые свойства. В C++ определение структуры расширили таким образом, что туда, как и в определение класса, удалось включить функции-члены, в том числе конструкторы и деструкторы. Таким образом, единственным отличием между структурой и классом является то, что члены класса, по умолчанию, являются закрытыми, а члены структуры — открытыми. Здесь показан расширенный синтаксис описания структуры:

```
struct имя_типа {
    // открытые функции и данные — члены класса
private:
    // закрытые функции и данные — члена класса
} список_объектов
```

Таким образом, в соответствии с формальным синтаксисом C++ как структура, так и класс создают новые *типы* данных. Обратите внимание на введение нового ключевого слова. Им является слово **private,** которое сообщает компилятору, что следующие за ним члены класса являются закрытыми.

В том, что структуры и классы обладают фактически одинаковыми свойствами, имеется кажущаяся избыточность. Те, кто только знакомится с C++, часто удивляются этому дублированию. При таком взгляде на проблему уже не кажутся необычными рассуждения о том, что ключевое слово **class** совершенно лишнее.

Объяснение этому может быть дано в "строгой" и "мягкой" формах. "Строгий" довод состоит в том, что необходимо поддерживать линию на совместимость с C. Стиль задания структур C совершенно допустим и для программ C++. Поскольку в C все члены структур по умолчанию открыты, это положение также поддерживается и в C++. Кроме этого, поскольку класс синтаксически отличается от структуры, определение класса открыто для развития в направлении, которое в конечном итоге может привести к несовместимости со взятым из C определением структуры. Если эти два пути разойдутся, то направление, связанное с C++, уже не будет избыточным.

"Мягким" доводом в пользу наличия двух сходных конструкций стало отсутствие какого-либо ущерба от расширения определения структуры в C++ таким образом, что в нее стало возможным включение функций-членов.

Хотя структуры имеют схожие с классами возможности, большинство программистов ограничивают использование структур взятыми из C формами и не применяют их для задания функций-членов. Для задания объекта, содержащего данные и код, эти программисты обычно указывают ключевое слово class. Однако все это относится к стилистике и является предметом вашего собственного выбора. (Далее в книге за исключением текущего раздела с

помощью ключевого слова **struct** задаются объекты, которые не имеют функций-членов.)

Если вы нашли интересной связь между классами и структурами, вас заинтересует и следующее необычное свойство C++: объединения и классы в этом языке столь же **близки!** В C++ объединение также представляет собой тип класса, в котором функции и данные могут содержаться в качестве его членов. Объединение похоже на структуру тем, что в нем по умолчанию все члены открыты до тех пор, пока не указан спецификатор **private.** Главное же в том, что в C++ все данные, которые являются членами объединения, находятся в одной и той же области памяти (точно так же, как и в C). Объединения могут содержать конструкторы и деструкторы. Объединения C++ совместимы с объединениями C.

Если в отношениях между структурами и классами существует, на первый взгляд, некоторая избыточность, то об объединениях этого сказать нельзя. В объектно-ориентированном языке важна поддержка инкапсуляции. Поэтому способность объединений связывать воедино программу и данные позволяет создавать такие типы классов, в **которых** все данные находятся в общей области памяти. Это именно то, чего нельзя сделать с помощью классов.

Применительно к C++ имеется несколько ограничений, накладываемых на использование объединений. Во-первых, они не могут наследовать какой бы то ни было класс и не могут быть базовым классом для любого другого класса. Объединения не могут иметь статических членов. Они также не должны содержать объектов с конструктором или деструктором, хотя сами по себе объединения *могут* иметь конструкторы и деструкторы.

В C++ имеется особый тип объединения — это *анонимное объединение (anonymous union).* Анонимное объединение не имеет имени типа и следовательно нельзя объявить переменную такого типа. Вместо этого анонимное объединение просто сообщает компилятору, что все его члены будут находиться в одной и той же области памяти. Во всех остальных отношениях члены объединения действуют и обрабатываются как самые обычные переменные. То есть, доступ к членам анонимного объединения осуществляется непосредственно, без использования оператора точка (.). Например, рассмотрим следующий фрагмент:

```
union { // анонимное объединение
    int i;
    char ch[4];
};
// непосредственный доступ к переменным i и ch
i = 10;
ch[0] = 'X';
```

Обратите внимание на то, что, поскольку переменные i и ch не являются частью какого бы то ни было объекта, доступ к ним осуществляется непосредственно. Тем не менее они находятся в одной и той же области памяти.

Смысл анонимных объединений в том и состоит, что они обеспечивают простой способ сообщить компилятору о необходимости разместить одну или несколько переменных в одной и той же области памяти. Исключая эту особенность, члены анонимного объединения больше ничем не отличаются от других переменных.

Все те ограничения, которые накладываются на использование обычных объединений, применимы и к анонимным объединениям. Кроме этого к ним добавлено еще несколько. Глобальное анонимное объединение должно быть объявлено как статическое. Анонимное объединение не может содержать закрытых членов. Имена членов анонимного объединения не должны конфликтовать с другими идентификаторами той же области видимости.

1. Ниже представлена короткая программа, в которой для создания класса используется ключевое слово **struct:**

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

// использование структуры для определения типа класса
struct st_type {
    st_type(double b, char *n);
    void show();
private:
    double balance;
    char name[40];
};

st_type::st_type(double b, char *n)
{ "
    balance = b;
    strcpy(name,n);
}

void st_type::show()
{
    cout << "Имя:" << name;
    cout << ":$" << balance;
    if(balance < 0.0) cout << "***";
    cout << "\n";
}

int main()
{
    st_type     acc1(100.12, "Johnson");
    st_type     acc2(-12.34, "Hedricks");
```

```
    acc1.show();
    acc2.show();
    return 0;
}
```

Отметьте, что, как уже говорилось, члены структуры по умолчанию являются открытыми. Для объявления закрытых членов необходимо использовать ключевое слово **private.**

Кроме этого отметьте существенное отличие между структурами С и структурами C++. В C++ имя тега становится также и законченным именем типа данных, которое можно использовать для объявления объектов. В С, чтобы имя тега стало законченным именем типа данных, перед ним надо указывать ключевое слово **struct.**

Ниже представлена только что рассмотренная программа, но вместо структуры здесь используется класс:

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

class cl_type {
    double balance;
    char name[40];
public:
    cl_type(double b, char *n);
    void show();
};

cl_type::cl_type(double b, char *n)
{
    balance = b;
    strcpy(name, n);
}

void cl_type::show()
{
    cout << "Имя:" << name;
    cout << ":$" << balance;
    if(balance < 0.0) cout << "***";
    cout << "\n";
}

int main()
{
    cl_type acc1(100.12, "Johnson");
    cl_type acc2(-12.34, "Hedricks");

    acc1.show();
    acc2.show();
```

```
    return 0;
}
```

2. Пример использования объединения для побайтного вывода значения типа **double** в двоичном представлении:

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

union bits {
    bits(doublen);
    void show_bits();
    double d;
    unsigned char c[sizeof (double)];
};

bits::bits(doublen)
{
    d = n;
}

voidbits::show_bits()
{
    int i, j;

    for( j = sizeof (double) - 1; j >= 0; j --) {
        cout << "Двоичное представление байта" << j << ":";
        for( i = 128; i; i >>= 1)
            if(i & c[ j ] ) cout << "1";
            else cout << "0";
        cout << "\n";
    }
}

int main()
{
    bits ob(1991.829);

    ob.show_bits();

    return 0;
}
```

Результат работы программы

```
Двоичное представление байта 7:  01000000
Двоичное представление байта 6:  10011111
Двоичное представление байта 5:  00011111
Двоичное представление байта 4:  01010000
Двоичное представление байта 3:  11100101
```

```
Двоичное представление байта 2:  01100000
Двоичное представление байта 1:  01000001
Двоичное представление байта 0:  10001001
```

3. Структуры и объединения могут иметь конструкторы и деструкторы. В следующем примере класс **strtype** переделан в структуру. В структуре имеются конструктор и деструктор.

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

struct strtype {
    strtype(char *ptr);
    ~strtype();
    void show();
private:
    char *p;
    int len;
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr);
    p = (char *) malloc(len + 1);
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << "- длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");
```

```
    s1.show();
    s2.show();

    return 0;
}
```

4. В следующей программе для побайтного вывода на экран значения типа **double** используется анонимное объединение. (Предполагается, что длина значения типа **double** равна восьми байтам.)

```
// Использование анонимного объединения
#include <iostream>
using namespace std;

int main()
{
    union {
        unsigned char bytes[8];
        double value;
    };
    int i;
    value = 859345.324;

    // побайтный вывод значения типа double
    for(i=0; i<8; i++)
        cout << (int) bytes[i] << " ";

    return 0;
}
```

Обратите внимание, что доступ к переменным **value и bytes** осуществляется так, как если бы они были не частью объединения, а обычными переменными. Несмотря на то, что эти переменные объявлены как часть анонимного объединения, их имена находятся в той же области видимости, что и другие объявленные здесь локальные переменные. Именно по этой причине член анонимного объединения не может иметь то же имя, что и любая переменная в данной области видимости.

Упражнения

1. Перепишите класс **stack,** представленный в разделе 2.1, чтобы вместо класса использовалась структура.
2. Используйте объединение, чтобы поменять местами старший и младший байты целого (предполагается 16-битное целое; если ваш компьютер использует 32-битное целое, то меняйте местами байты типа **short int).**
3. Объясните, что такое анонимное объединение и чем оно отличается от нормального объединения.

2.6. Встраиваемые функции

Перед тем как продолжить исследование классов необходимо краткое отступление. В C++ можно задать функцию, которая на самом деле не вызывается, а ее тело встраивается в программу в месте ее вызова. Она действует почти так же, как макроопределение с параметрами в C. Преимуществом встраиваемых *(in-line)* функций является то, что они не связаны с механизмом вызова функций и возврата ими своего значения. Это значит, что встраиваемые функции могут выполняться гораздо быстрее обычных. (Запомните, что выполнение машинных команд, которые генерируют вызов функции и возвращение функцией своего значения, занимает определенное время. Если функция имеет параметры, то ее вызов занимает еще большее время.)

Недостатком встраиваемых функций является то, что если они слишком большие и вызываются слишком часто, объем ваших программ сильно возрастает. Из-за этого применение встраиваемых функций обычно ограничивается короткими функциями.

Для объявления встраиваемой функции просто впишите спецификатор **inline** перед определением функции. Например, в этой короткой программе показано, как объявить встраиваемую функцию:

```
// Пример встраиваемой функции
#include <iostream>
using namespace std;

inline int even(int x)
{
  return ! (x%2);
}

int main()
{
  if (even(10)) cout << "10 является четным\n";
  if (even(11)) cout << "11 является четным\n";
  return 0;
}
```

В этом примере функция **even(),** которая возвращает истину при четном аргументе, объявлена встраиваемой. Это означает, что строка

```
if (even(10)) cout << "10 является четным\n";
```

функционально идентична строке

```
if (! (10%2)) cout << "10 является четным\n";
```

Этот пример указывает также на другую важную особенность использования встраиваемой функции: она должна быть задана *до* ее первого вызова. Если

это не так, компилятор не будет знать, какой именно код предполагается встроить в программу с помощью встраиваемой функции. Поэтому функция even() была определена перед функцией **main()**.

В пользу использования встраиваемых функций вместо макроопределений с параметрами имеется два довода. Во-первых, они обеспечивают более стройный способ встраивания в программу коротких фрагментов кода. Например, при создании макроса с параметрами легко забыть, что для гарантии правильности встраивания в каждом случае часто требуются круглые внешние скобки. Встраиваемая функция исключает эту проблему.

Во-вторых, компилятор гораздо лучше работает со встраиваемой функцией, чем с макрорасширением. Как правило, программисты C++ для многократных вызовов коротких функций вместо макросов с параметрами практически всегда используют встраиваемые функции.

Здесь важно понимать, что спецификатор inline для компилятора является *запросом,* а не командой. Если, по разным причинам, компилятор не в состоянии выполнить запрос, функция будет компилироваться, как обычная функция, а запрос inline будет проигнорирован.

В зависимости от типа вашего компилятора возможны некоторые ограничения на использование встраиваемых функций. Например, некоторые компиляторы не воспринимают функцию как встраиваемую, если функция является рекурсивной или если она содержит либо статическую (static) переменную, либо любую инструкцию выполнения цикла, либо инструкцию switch, либо инструкцию goto. Вам необходимо просмотреть руководство по вашему компилятору, чтобы точно определить ограничения на использование встраиваемых функций.

Если какое-либо ограничение на использование встраиваемой функции нарушено, компилятор генерирует вместо нее обычную функцию.

1. Любая функция может стать встраиваемой, включая функции — члены классов. Например, функция **divisible()** для ускорения ее выполнения сделана встраиваемой. (Функция возвращает истину, если ее первый аргумент без остатка может делиться на второй.)

```
// Демонстрация встраиваемой функции-члена
#include <iostream>
using namespace std;
```

```
class samp {
    int i, j;
public:
    samp(int a, int b) ;
    int divisible(); // встраивание происходит в этом определении
};

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

/* Возврат 1, если i без остатка делится на j. Тело этой функции-
члена встраивается в программу
*/
inline int samp::divisible()
{
    return !(i%j);
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);

    // это истина
    if(ob1.divisible()) cout << "10 делится на 2\n";

    // это ложь
    if(ob2.divisible()) cout << "10 делится на 3\n";

    return 0;
}
```

2. **Допускается перегружать встраиваемую функцию. Например, эта программа перегружает min() тремя способами. В каждом случае функция также объявляется встраиваемой.**

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

// Перегрузка функции min() тремя способами

// int
inline int min(int a, int b)
{
    return a < b ? a: b;
}

// long
inline long min(long a, long b)
```

```
{
    return a < b ? a: b;
}

// double
inline double min(double a, double b)
{
    return a < b ? a: b;
}

int main()
{
    cout << min(-10, 10) << "\n";
    cout << min(-10.01, 100.002) << "\n";
    cout << min(-10L, 12L) << "\n";

    return 0;
}
```

1. В главе 1 вы перегружали функцию **abs()** так, чтобы она находила абсолютные значения типа **int, long и double.** Модифицируйте программу, чтобы эти функции стали встраиваемыми.
2. Почему следующая функция может не компилироваться как встраиваемая?

```
void f1()
{
    int i;

    for(i = 0; i < 10; i++) cout << i;
}
```

2.7. Встраиваемые функции в объявлении класса

Если определение функции-члена достаточно короткое, его можно включить в объявление класса. Поступив таким образом, мы заставляем, если это возможно, функцию стать встраиваемой. Если функция задается внутри объявления класса, ключевое слово **inline** не требуется. (Однако использование его в такой ситуации не является ошибкой.) Например, как показано ниже, функция **divisible()** из предыдущего раздела может быть по умолчанию сделана встраиваемой:

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

class samp {
    int i, j;
public:
    samp(int a, int b);

/* Функция divisible(), которая здесь определяется, по умолчанию
становится встраиваемой.
*/
    int divisible() { return !(i%j); }
};

samp::samp(int a, int b)
{
    i = a;
    j = b;
}

int main()
{
    samp ob1(10, 2), ob2(10, 3);

    // это истина
    if(ob1.divisible()) cout << "10 делится на 2\n";

    // это ложь
    if(ob2.divisible()) cout << "10 делится на 3\n";

    return 0;
}
```

Как видите, код функции **divisible()** находится внутри объявления класса **samp.** Отметьте, что никакого другого определения функции divisible() не нужно, это даже запрещено. Определение функции **divisible()** внутри класса **samp** автоматически заставляет ее стать встраиваемой функцией.

Если функция, заданная внутри объявления класса, не может стать встраиваемой функцией (поскольку были нарушены ограничения), она, обычно, преобразуется в обычную функцию.

Отметьте, как именно функция divisible() задается внутри класса **samp,** особенно само тело функции. Оно целиком расположено на одной строке. Такой формат для программ C++ является совершенно обычным, если функция объявляется внутри объявления класса. Такое объявление становится более компактным. Однако класс **samp** мог бы быть описан и так:

```
class samp {
    int i, j;
```

```
public:
    samp(int a, int b);

/* Функция divisible(), которая здесь определяется, по умолчанию
становится встраиваемой.
*/
    int divisible()
    {
        return !(i%j);
    }
};
```

Здесь в определении функции **divisible()** используется более или менее стандартный стиль отступов. С точки зрения компилятора, компактный и стандартный стили не отличаются. Однако в программах C++ при задании коротких функций внутри определения класса обычно используется компактный стиль.

На применение таких встраиваемых функций накладываются те же ограничения, что и на применение обычных встраиваемых функций.

Примеры

1. Вероятно наиболее традиционным использованием встраиваемых функций, определяемых внутри класса, является определение конструктора и деструктора. Например, класс **samp** может быть определен более эффективно:

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

class samp {
    int i, j;
public:
    // встраиваемый конструктор
    samp(int a, int b) { i = a; j = b; }
    int divisible() { return !(i%j); }
};
```

   Определения функции **samp()** внутри класса **samp** достаточно, и никакого другого определения не требуется.

2. Иногда короткие функции могут включаться в объявление класса даже тогда, когда преимущества встраивания мало что дают или вовсе не проявляются. Рассмотрим следующее объявление класса:

```
class myclass {
    int i;
```

```
public:
    myclass(int n) { i = n; }
    void show() { cout << i; }
};
```

Здесь функция **show()** по умолчанию становится встраиваемой. Однако, как вы, наверное, знаете, операции ввода/вывода, по сравнению с операциями процессор/память, являются настолько медленными, что какой бы то ни было эффект от устранения вызова функции практически отсутствует. Однако в программах на C++, как правило, можно встретить такие короткие функции внутри класса. Делается это просто для удобства, поскольку никакого вреда не приносит.

1. Переделайте класс **stack** из раздела 2.1, пример 1, так, чтобы в классе, где это возможно, использовались встраиваемые функции.
2. Переделайте класс **strtype** из раздела 2.2, пример 3, так, чтобы в классе использовались встраиваемые функции.

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Что такое конструктор? Что такое деструктор? Когда они вызываются?
2. Создайте класс **line,** который рисует на экране линию. Храните длину линии в виде закрытой целой переменной **len**. Конструктор **line** должен иметь один параметр — длину линии. Он должен сохранять длину линии и собственно рисовать линию. Если ваша система не поддерживает графику, отобразите линию, используя символ *. Необязательное задание: Для удаления линии используйте деструктор **line.**
3. Что выведет на экран следующая программа?

```
ttinclude<iostream>
using namespace std;

int main()
{
    int i = 10;
    long l = 1000000;
    double d = -0.0009;
```

```
    cout << i << ' ' << l << ' ' << d;
    cout << "\n";

    return 0;
}
```

4. Добавьте производный класс, который наследует класс **area_cl** из раздела 2.3, упражнение 1. Назовите этот класс **cylinder** и пусть он вычисляет площадь поверхности цилиндра. Эта площадь задается так: 2 * pi * R2 + pi * D * H.
5. Что такое встраиваемая функция? В чем ее преимущества и недостатки?
6. Измените следующую программу так, чтобы все функции-члены по умолчанию стали встраиваемыми функциями:

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

class myclass {
    int i, j;
public:
    myclass(int x, int y);
    void show();
};

myclass::myclass(int x, int y)
{
    i = x;
    j = y;
}

void myclass::show()
{
    cout << i << "" "" << j << "\n";
}

int main()
{
    myclass count(2, 3);

    count.show();

    return 0;
}
```

7. В чем отличие между классом и структурой?
8. Правилен ли следующий фрагмент?

```
union {
    float f;
    unsigned int bits;
};
```

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Создайте класс **prompt.** Для вывода на экран строки-приглашения включите в класс конструктор. Помимо строки-приглашения конструктор должен выполнить ввод набранного вами целого. Сохраните это значение в закрытой переменной **count.** При удалении объекта типа **prompt** должен быть подан звуковой сигнал, причем столько раз, сколько задано пользователем в переменной **count.**
2. В главе 1 вы создали программу для преобразования футов в дюймы. Теперь для этой цели создайте класс. Класс должен хранить число футов и его эквивалент в дюймах. Передайте конструктору класса число футов и при этом он должен вывести на экран число дюймов.
3. Создайте класс dice, который содержит закрытую целую переменную. Создайте функцию **ro11(),** использующую стандартный генератор случайных чисел **rand(),** для получения чисел от 1 до 6. Функция **roll()** должна вывести это значение на экран.

Глава 3

Подробное изучение классов

В этой главе вы продолжите изучение классов. Вы узнаете о том, как присвоить один объект другому, как объекты передаются функциям в качестве аргументов, как сделать объект возвращаемым значением функций. Вы также узнаете о новом важном типе функций: дружественных (friend) функциях.

Повторение пройденного

Перед тем как продолжить, вы должны правильно ответить на следующие вопросы и сделать упражнения.

1. Пусть дан следующий класс, каковы имена его конструктора и деструктора?

```
class widgit {
    int x, y;
public:
    // ... впишите конструкторы и деструкторы
};
```

2. Когда вызывается конструктор? Когда вызывается деструктор?
3. Пусть дан следующий базовый класс, покажите, как он может наследоваться производным классом **Mars.**

```
class planet {
    int moons;
    double dist_from_sun;
    double diameter;
    double mass;
public:
// ...
};
```

4. Имеются два способа сделать функцию встраиваемой. Что это за способы?
5. Приведите, по крайней мере, два ограничения на использование встраиваемых функций.

6. Пусть дан следующий класс, покажите, каким образом объявить объект **ob,** чтобы значение 100 передать переменной a, а значение X переменной c.

```
class sample {
    int a;
    char c;
public:
    sample(int x, char ch) { a = x; c = ch; }
    // ...
};
```

3.1. Присваивание объектов

Если тип двух объектов одинаков, то один объект можно присвоить другому. По умолчанию, когда один объект присваивается другому, делается поразрядная копия всех данных-членов копируемого объекта. Например, когда объект **o1** присваивается объекту **o2,** то содержимое всех данных объекта **o1** копируется в соответствующие члены объекта **o2.** Это иллюстрируется следующей программой:

```
// Пример присваивания объекта
ttinclude<iostream>
using namespace std;

class myclass {
    int a, b;
public:
    vo.id set(int i, int j) { a= i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    myclass o1, o2;

    o1.set(10, 4);

    // o1 присваивается o2
    o2 = o1;

    o1.show();
    o2.show();

    return 0;
}
```

В этом примере переменным a и b объекта **ol** присваиваются соответственно значения 10 и 4. Далее объект **ol** присваивается объекту **o2.** Это приво-

дит к тому, что текущее значение переменной **o1.a** присваивается переменной o2.a, а текущее значение переменной **o1.b** — o2.b. Таким образом, в процессе выполнения программа выведет на экран следующее:

```
10 4
10 4
```

Запомните, что присваивание двух объектов просто делает одинаковыми данные этих объектов. Два объекта остаются по-прежнему совершенно независимыми. Например, после выполнения присваивания вызов функции-члена **o1.set()** для задания значения **ol.a** не влияет ни на объект **o2**, ни на значение его переменной **a**.

1. Важно понимать, что в инструкции присваивания можно использовать только объекты одинакового типа. Если тип объектов разный, то при компиляции появится сообщение об ошибке. Более того, важно не только чтобы типы объектов были физически одинаковыми, а чтобы одинаковыми были также имена типов. Например, следующая программа неправильна:

```
// Эта программа содержит ошибку
ttinclude<iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

/* Этот класс похож на класс myclass, но из-за другого имени класса
для компилятора он считается другим типом.
*/
class yourclass {
    int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    myclass o1;
    yourclass o2;

    o1.set(10, 4);
```

```
    o2 = o1; // ОШИБКА, присваивание объектов разных типов

    o1.show();
    o2.show();

    return 0;
}
```

Несмотря на то что классы **myclass и yourclass** физически одинаковы, они трактуются компилятором как разные, поскольку имеют разные имена типов.

2. Важно понимать, что все данные-члены одного объекта при выполнении присваивания присваиваются данным-членам другого объекта. Это относится и к сложным данным, таким как массивы. Например, в следующей версии уже знакомого нам класса **stack** символы реально помещаются только в стек **s1**, но после выполнения присваивания массив **stck** объекта s2 также содержит символы a, b и c.

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

#define SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); // конструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает символ из стека
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека\n";
    tos = 0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}
```

```
// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Образование двух, автоматически инициализируемых стеков
    stack s1, s2;
    int i;

    s1.push('a');
    s1.push('b');
    s1.push('c');

    // копирование s1 в s2
    s2 = s1; // теперь s2 и s1 идентичны

    for(i=0; i<3; i++) cout << "символ из s1:" << s1.pop()
                                                << "\n";
    for(i=0; i<3; i++) cout << "символ из s2: " << s2.pop()
                                                  << "\n";

    return 0;
}
```

3. При присваивании одного объекта другому необходимо быть очень внимательным. Например, рассмотрим несколько измененный класс **strtype**, который мы изучали в главе 2. Попытайтесь найти ошибку в этой программе.

```
// Эта программа содержит ошибку
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype();
    void show();
};
```

```
strtype::strtype(char *ptr)
{
    len=strlen(ptr);
    p=(char *) malloc(len+1) ;
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr) ;
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    // s1 присваивается s2 - это ведет к ошибке
    s2 = s1;

    s1.show();
    s2.show();

    return 0;
}
```

Ошибка в этой программе весьма коварна. Когда создаются объекты **s1** и s2, то для хранения в этих объектах строк выделяется память. Указатель на выделенную каждому объекту память хранится в переменной p. Когда объект **strtype** удаляется, эта память освобождается. Однако когда объект **s1** присваивается объекту s2, то указатель p объекта s2 начинает указывать на ту же самую область памяти, что и указатель p объекта **s1**. Таким образом, когда эти объекты удаляются, то память, на которую указывает указатель p объекта **s1**, освобождается *дважды,* а память, на которую до присваивания указывал указатель p объекта **s2**, не освобождается *вообще.*

Хотя в данном случае эта ошибка и не опасна, в реальных программах с динамическим распределением памяти она может вызвать крах программы. Как показано в этом примере, при присваивании одного объекта другому вы

должны быть уверены в том, что не удаляете нужную информацию, которая может понадобиться в дальнейшем.

1. Что неправильно в следующем фрагменте?

```
// В этой программе есть ошибка
#include <iostream>
using namespace std;

class cl1 {
    int i, j;
public:
    cl1(int a, int b) { i = a; j = b; }
    // ...
};

class cl2 {
    int i, j;
public:
    cl2(int a, int b) { i = a; j = b; }
    // ...
};

int main()
{
    cl1 x(10, 20);
    cl2 y(0, 0);
    x = y;

    // ...
}
```

2. Используя класс **queue,** который вы создали в главе 2, раздел 2.1, упражнение 1, покажите, как один объект, который мы назвали очередь, можно присвоить другому.
3. Если класс **queue** из предыдущего вопроса для хранения очереди требует динамического выделения памяти, то почему в такой ситуации одну очередь нельзя присвоить другой?

3.2. Передача объектов функциям

Объекты можно передавать функциям в качестве аргументов точно так же, как передаются данные других типов. Просто объявите параметр функции, как имеющий тип класса, и затем используйте объект этого класса в качест-

ве аргумента при вызове функции. Как и для данных других типов, по умолчанию объекты передаются в функции по значению.

1. В этом коротком примере объект передается функции:

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

class samp {
    int i;
public:
    samp(int n) { i = n; }
    int get_i() { return i; }
};

// Возвращает квадрат o.i.
int sqr_it(samp o)
{
    return o.get_i() * o.get_i();
}

int main()
{
    samp a(10), b(2);

    cout << sqr_it(a) << "\n";
    cout << sqr_it(b) << "\n";

    return 0;
}
```

В этой программе создается класс **samp,** который содержит одну целую переменную i. Функция **sqr_it()** получает аргумент типа **samp,** а возвращаемым значением является квадрат переменной i этого объекта. Результат работы программы — это значения 100 и 4.

2. Как уже установлено методом передачи параметров в C++, включая объекты, по умолчанию является передача объекта по значению. Это означает, что внутри функции создается копия аргумента и эта копия, а не сам объект, используется функцией. Поэтому изменение копии объекта внутри функции не влияет на сам объект. Это иллюстрируется следующим примером:

```
/* Запомните, объекты, как и другие параметры, передаются функции по
значению и при этом в функции создается копия объекта. Таким
образом, изменение параметра внутри функции не влияет на объект,
используемый в вызове.
*/
```

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

class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* Заменяет переменную o.i ее квадратом. Однако это не влияет на
объект, используемый для вызова функции sqr_it ()
*/
void sqr_it(samp o)
{
    o.set_i( o.get_i() * o.get_i());

    cout << "Для копии объекта a значение i равно: " << o.get_i();
    cout << "\n";
}

int main()
{
    samp a(10);

    sqr_it(a); // передача объекта a по значению

    cout << "но переменная a.i в функции main() не изменилась: ";
    cout << a.get_i(); // выводится 10

    return 0;
}
```

В результате работы программы на экран выводится следующее:

```
Для копии объекта a значение i равно: 100
но переменная a.i в функции main() не изменилась: 10
```

3. Как и в случае с переменными других типов, функции может быть передано не значение объекта, а его адрес. В этом случае функция может изменить значение аргумента, используемого в вызове. Например, в рассматриваемом ниже варианте программы из предыдущего примера значение объекта, чей адрес используется при вызове функции **sqr_it(),** действительно меняется.

```
/* Теперь функции sqr_it() передается адрес объекта и функция может
изменить значение аргумента, адрес которого используется при вызове.
*/
#include <iostream>
using namespace std;
```

```
class samp {
    int i;
public:
    samp(int n) { i = n; }
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

/* Заменяет переменную o.i ее квадратом. Это влияет на объект,
используемый при вызове
*/
void sqr_it(samp *o)
{
    o->set_i(o->get_i() * o->get_i());

    cout << "Для объекта a значение i равно: " << o->get_i();
    cout << "\n";
}

int main()
{
    samp a(10);

    sqr_it(&a); // функции sqr_it() передан адрес объекта a

   cout << "Теперь значение объекта a в функции main() изменилось:";
   cout << a.get_i(); // выводится 100

    return 0;
}
```

Теперь результат работы программы следующий:

```
Для объекта a значение i равно: 100
Теперь значение объекта a в функции main() изменилось: 100
```

4. Если при передаче объекта в функцию делается его копия, это означает, что появляется новый объект. Когда работа функции, которой был передан объект, завершается, то копия аргумента удаляется. Возникают два вопроса. Во-первых, вызывается ли конструктор объекта, когда создается его копия? Во-вторых, вызывается ли деструктор объекта, когда эта копия удаляется? Ответ на первый вопрос может показаться неожиданным.

   Когда при вызове функции создается копия объекта, конструктор копии *не* вызывается. Смысл этого понять просто. Поскольку конструктор обычно используется для инициализации некоторых составляющих объекта, он не должен вызываться при создании копии уже существующего объекта. Если бы это было сделано, то изменилось бы содержимое объекта, поскольку при передаче объекта функции необходимо его текущее, а не начальное состояние.

Однако если работа функции завершается и копия удаляется, то деструктор копии вызывается. Это происходит потому, что иначе оказались бы невыполненными некоторые необходимые операции. Например, для копии может быть выделена память, которую, после завершения работы функции, необходимо освободить.

Итак, при создании копии объекта, когда он используется в качестве аргумента функции, конструктор копии не вызывается. Однако, когда копия удаляется (обычно это происходит при возвращении функцией своего значения), вызывается ее деструктор.

Следующая программа иллюстрирует эти положения:

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

class samp {
    int i;
public:
    samp(int n) {
        i = n;
        cout << "Работа конструктора\n";
     }
    ~samp() { cout << "Работа деструктора\n"; }
    int get_i() { return i; }
};

// Возвращает квадрат переменной o.i
int sqr_it(samp o)
{
    return o.get_i()*o.get_i();
}

int main()
{
    samp a(10);

    cout << sqr_it(a) << "\n";

    return 0;
}
```

Эта программа выводит следующее:

```
Работа конструктора
100
Работа деструктора
Работа деструктора
```

Обратите внимание, что конструктор вызывается только один раз. Это происходит при создании объекта a. Однако деструктор вызывается дважды.

Первый раз он вызывается для копии, созданной, когда объект a был передан функции **sqr_it(),** другой — для самого объекта a.

Тот факт, что деструктор объекта, являющегося копией передаваемого функции аргумента, выполняется при завершении работы функции, может стать потенциальным источником проблем. Например, если для объекта, используемого в качестве аргумента, выделена динамическая память, которая освобождается при его удалении, тогда и для копии объекта при вызове деструктора будет освобождаться та же самая память. Это приведет к повреждению исходного объекта. (Для примера см. упражнение 2 данного раздела.) Чтобы избежать такого рода ошибок, важно убедиться в том, что деструктор копии объекта, используемого в качестве аргумента, не вызывает никаких побочных эффектов, которые могли бы повлиять на исходный аргумент.

Как вы, возможно, уже догадались, одним из способов обойти проблему удаления деструктором необходимых данных при вызове функции с объектом в качестве аргумента должна стать передача функции не самого объекта, а его адреса. Если функции передается адрес объекта, то нового объекта не создается и поэтому при возвращении функцией своего значения деструктор не вызывается. (Как вы увидите в следующей главе, в C++ имеется и иное, более элегантное решение этой задачи.) Тем не менее имеется другое, лучшее решение, о котором вы узнаете, изучив особый тип конструктора, а именно *конструктор копий (copy constructor)*. Конструктор копий позволяет точно определить порядок создания копий объекта. (О конструкторах копий рассказывается в главе 5.)

Упражнения

1. Используя класс **stack** из раздела 3.1, пример 2, добавьте в программу функцию **showstack()**, которой в качестве аргумента передается объект типа **stack.** Эта функция должна выводить содержимое стека на экран.
2. Как вы знаете, если объект передается функции, создается копия этого объекта. Далее, когда эта функция возвращает свое значение, вызывается деструктор копии. Вспомнив это, ответьте, что неправильно в следующей программе?

```
// В этой программе есть ошибка
#include <iostream>
#include <cstdlib>
using namespace std;

class dyna {
    int *p;
public:
    dyna(int i);
    ~dyna() { free(p); cout << "освобождение памяти\n"; }
```

```
    int get() { return *p; }
};

dyna::dyna(int i)
{
    p = (int *) malloc(sizeof(int));
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    *p = i;
}

// Возвращает отрицательное значение *ob.p
int neg(dyna ob)
{
    return -ob.get();
}

int main()
{
    dyna o(-10);

    cout << o.get() << "\n";
    cout << neg(o) << "\n";

    dyna o2(20);

    cout << o2.get() << "\n";
    cout << neg(o2) << "\n";

    cout << o.get() << "\n";
    cout << neg(o) << "\n";

    return 0;
}
```

3.3. Объекты в качестве возвращаемого значения функций

Так же как объект может быть передан функции в качестве аргумента, он может быть и возвращаемым значением функций. Для этого, во-первых, объявите функцию так, чтобы ее возвращаемое значение имело тип класса. Во-вторых, объект этого типа возвратите с помощью обычной инструкции **return.**

Имеется одно важное замечание по поводу объектов в качестве возвращаемого значения функций: если функция возвращает объект, то для хранения возвращаемого значения автоматически создается временный объект. После того как значение возвращено, этот объект удаляется. Удаление этого временного объекта может приводить к неожиданным побочным эффектам, что иллюстрируется в примере 2 этого раздела.

1. Пример функции с объектом в качестве возвращаемого значения:

```
// Возвращение объекта из функции
#include <iostream>
#include <cstring>
using namespace std;

class samp {
    char s[80];
public:
    void show() { cout << s << "\n"; }
    void set(char *str) { strcpy(s, str); }
};

// Возвращает объект типа samp
samp input()
{
    char s[80];
    samp str;

    cout << "Введите строку: ";
    cin>>s;

    str.set(s);

    return str;
}

int main()
{
    samp ob;

    // присваивание возвращаемого значения объекту ob
    ob = input();
    ob.show();

    return 0;
}
```

В этом примере функция **input()** создает локальный объект str и считывает строку с клавиатуры. Эта строка копируется в str.s, и затем функция возвращает объект **str**. Внутри функции **main()** при вызове функции **input()** возвращаемый объект присваивается объекту ob.

2. Следует быть внимательными при возвращении объектов из функций, если эти объекты содержат деструктор, поскольку возвращаемый объект выходит из области видимости, как только функция возвращает его значение. Например, если функция возвращает объект, имеющий деструктор, который освобождает динамически выделенную память, то эта память будет освобождена независимо от того, использует ли ее объект, которому присваивается возвращаемое значение, или нет. Например, рассмотрим неправильную версию предыдущей программы:

```
// При возвращении объекта генерируется ошибка
#include <iostream>
ttinclude<cstring>
#include <cstdlib>
using namespace std;

class samp {
    char *s;
public:
    samp() { s = '\0'; }
    ~samp() { if(s) free(s);
                cout << "Освобождение памяти по адресу s\n";}
    void show() { cout << s << "\n"; }
    void set(char *str);
};

// Загружает строку
void samp::set(char *str)
{
    s = (char *} malloc(strlen(str)+1);
    if(!s) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
strcpy(s, str);
}

// Возвращает объект типа samp
samp input()
{
    char s[80];
    samp str;

    cout << "Введите строку: ";
    cin>>s;
```

```
    str.set(s);
    return str;
}

int main()
{
    samp ob;

    // возвращаемое значение присваивается объекту ob
    ob = input(); // Это ведет к ошибке
    ob.show();

    return 0;
}
```

Здесь показан результат работы программы:

```
Введите строку: Привет
Освобождение памяти по адресу s
Освобождение памяти по адресу s
Привет
Освобождение памяти по адресу s
Null pointer assignment
```

Обратите внимание, что деструктор класса **samp** вызывается трижды. Первый раз, когда локальный объект **str** выходит из области видимости при возвращении функцией **input()** своего значения. Второй раз **~samp()** вызывается тогда, когда удаляется временный объект, возвращаемый функцией **input()**.

Запомните, когда объект возвращается функцией, автоматически генерируется невидимый (для вас) временный объект, который и хранит возвращаемое значение. В этом случае временный объект — это просто копия объекта str, являющегося возвращаемым значением функции. Следовательно, после того как функция возвратила свое значение, выполняется деструктор временного объекта. И наконец, при завершении программы вызывается деструктор объекта ob в функции **main()**.

Проблема в этой ситуации в том, что при первом выполнении деструктора память, выделенная для хранения вводимой с помощью функции **input()** строки, освобождается. Таким образом, при двух других вызовах деструктора класса **samp** не только делается попытка освободить уже освобожденный блок динамической памяти, но в процессе работы происходит разрушение самой системы динамического распределения памяти и, как доказательство этому, появляется сообщение "Null pointer assignment". (В зависимости от вашего компилятора, модели используемой памяти и тому подобного при попытке выполнить программу это сообщение может и не появиться.)

Ключевым моментом в понимании проблемы, описанной в этом примере, является то, что при возвращении функцией объекта для временного объекта, который и является возвращаемым значением функции, вызывается дест-

руктор. (Как вы узнаете в главе 5, для решения проблемы в такой ситуации можно воспользоваться конструктором копий.)

1. Для внимательного изучения вопроса, когда при возвращении функцией объекта для него вызываются конструктор и деструктор, создайте класс **who.** Конструктор **who** должен иметь один символьный аргумент, который будет использоваться для идентификации объекта. При создании объекта конструктор должен выводить на экран сообщение:

   ```
   Создание объекта who #x
   ```

   где x — идентифицирующий символ, свой для каждого объекта. При удалении объекта на экран должно выводиться примерно такое сообщение:

   ```
   Удаление объекта who #x
   ```

   где x — снова идентифицирующий символ. Наконец, создайте функцию **make_who(),** которая возвращает объект who. Присвойте каждому объекту уникальное имя. Проанализируйте выводимый на экран результат работы программы.
2. Продумайте ситуацию, в которой, как и при неправильном освобождении динамической памяти, возвращать объект из функции было бы также ошибочно.

3.4. Дружественные функции: обзор

Возможны ситуации, когда для получения доступа к закрытым членам класса вам понадобится функция, не являющаяся членом этого класса. Для достижения этой цели в C++ поддерживаются *дружественные функции (friend functions)*. Дружественные функции не являются членами класса, но тем не менее имеют доступ к его закрытым элементам.

В пользу существования дружественных функций имеются два довода, связанные с перегрузкой операторов и созданием специальных функций ввода/вывода. Об этом использовании дружественных функций вы узнаете несколько позднее. Сейчас познакомимся с третьим доводом в пользу существования таких функций. Вам наверняка когда-нибудь понадобится функция, которая имела бы доступ к закрытым членам *двух или более* разных классов. Рассмотрим, как создать такую функцию.

Дружественная функция задается так же, как обычная, не являющаяся членом класса, функция. Однако в объявление класса, для которого функция будет дружественной, необходимо включить ее прототип, перед которым ставится ключевое слово **Mend.** Чтобы понять, как работает дружественная функция, рассмотрим следующую короткую программу:

```
// Пример использования дружественной функции
#include <iostream>
using namespace std;

class myclass {
    int n, d;
public:
    myclass(int i, int j) { n = i; d = j; }
    // объявление дружественной функции для класса myclass
    friend int isfactor(myclass ob);
};

/* Здесь представлено определение дружественной функции. Она
возвращает истину, если n делится без остатка на d. Отметьте, что
ключевое слово friend в определении функции isfactor() не
используется.
*/
int isfactor(myclass ob)
{
    if(!(ob.n % Ob.d)) return 1;
    else return 0;
}

int main()
{
    myclass ob1(10, 2), ob2(13, 3);

    if(isfactor(ob1) cout << "10 без остатка делится на 2\n";
    else cout << "10 без остатка не делится на 2\n";

    if(isfactor(ob2) cout << "13 без остатка делится на 3\n";
    else cout << "13 без остатка не делится на 3\n";

    return 0;
}
```

В этом примере в объявлении класса **myclass** объявляются конструктор и дружественная функция **isfactor()**. Поскольку функция **isfactor()** дружественна для класса **myclass,** то функция **isfactor()** имеет доступ к его закрытой части. Поэтому внутри функции **isfactor()** можно непосредственно ссылаться на объекты ob.n и ob.d.

Важно понимать, что дружественная функция не является членом класса, для которого она дружественна. Поэтому невозможно вызвать дружественную функцию, используя имя объекта и оператор доступа к члену класса (точку или стрелку). Например, по отношению к предыдущему примеру эта инструкция неправильна:

```
ob1.isfactor(); // неправильно, isfactorO — это не функция-член
```

На самом деле дружественная функция должна вызываться точно так же, как и обычная функция.

Хотя дружественная функция "знает" о закрытых элементах класса, для которого она является дружественной, доступ к ним она может получить только через объект этого класса. Таким образом, в отличие от функции-члена **myclass,** в котором можно непосредственно упоминать переменные n и d, дружественная функция может иметь доступ к этим переменным только через объект, который объявлен внутри функции или передан ей.

В предыдущем разделе был рассмотрен важный момент. Когда функция-член использует закрытый элемент класса, то это делается непосредственно, поскольку функция-член выполняется только с объектами данного класса. Таким образом, если в функции-члене класса упоминается закрытый элемент этого класса, то компилятор знает, к какому объекту этот закрытый элемент относится, поскольку знает, с каким объектом при вызове функции-члена связана функция. Однако дружественная функция не связана с каким бы то ни было объектом. Просто ей предоставлен доступ к закрытым элементам класса. Поэтому дружественная функция не может работать с закрытыми членами класса без упоминания конкретного объекта.

Поскольку дружественные функции — это не члены класса, им обычно передается один или более объектов класса, для которого они являются дружественными. Так было сделано, например, в случае с функцией **isfactor()**. Ей был передан объект **ob** класса **myclass.** Поскольку функция isfactor() дружественна для класса **myclass,** она может иметь доступ к закрытым элементам этого класса через объект **ob.** Случись так, что функция isfactor() не была бы дружественной для класса **myclass,** она не имела бы доступа ни к объекту **ob.n,** ни к объекту **ob.d,** поскольку переменные n и d — это закрытые члены класса **myclass.**

Дружественная функция — это не член класса и она не может быть задана через имя объекта. Она должна вызываться точно так же, как и обычная функция.

Дружественная функция не наследуется. Поэтому если в базовый класс включается дружественная функция, то эта дружественная функция не является таковой для производных классов.

Другим важным моментом, относящимся к дружественным функциям, является то, что функция может быть дружественной более чем к одному классу.

Примеры

1. Обычно дружественная функция бывает полезна тогда, когда у двух разных классов имеется нечто общее, что необходимо сравнить. Например, рассмотрим следующую программу, в которой создаются классы car (легковая машина) и **truck** (грузовик), причем оба содержат в закрытой переменной скорость соответствующего транспортного средства:

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

class truck; // предварительное объявление

class car {
    int passengers;
    int speed;
public:
    car(int p, int s) { passengers = p; speed = s; }
    friend int sp_greater(car c, truck t);
};

class truck {
    int weight;
    int speed;
public:
    truck(int w, int s) { weight = w; speed = s; }
    friend int sp_greater(car c, truck t);
};

/* Возвращает положительное число, если легковая машина быстрее
грузовика. Возвращает 0 при одинаковых скоростях. Возвращает
отрицательное число, если грузовик быстрее легковой машины.
*/
int sp_greater(car c, truck t)
{
    return c.speed - t.speed;
}

int main()
{
    int t;
    car c1(6, 55), c2 (2, 120);
    truck t1(10000, 55), t2(20000, 72);

    cout << "Сравнение значений c1 и t1:\n";
    t = sp_greater(c1, t1);
    if (t<0) cout << "Грузовик быстрее. \n";
    else if (t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";
```

```
    cout << "\nСравнение значений c2 и t2:\n";
    t = sp_greater(c2, t2);
    if(t<0) cout << "Грузовик быстрее. \n";
    else if(t==0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    return 0;
}
```

В этой программе имеется функция **sp_greater()**, которая дружественна для классов **car** и **truck.** (Как уже установлено, функция может быть дружественной двум и более классам.) Эта функция возвращает положительное число, если объект **car** движется быстрее объекта **truck,** нуль, если их скорости одинаковы, и отрицательное число, если скорость объекта **truck** больше, чем скорость объекта car.

Эта программа иллюстрирует один важный элемент синтаксиса C++ — *предварительное объявление (forward declaration),* которое еще называют *ссылкой вперед (forward reference).* Поскольку функция **sp_greater()** получает параметры обоих классов **car** и **truck,** то логически невозможно объявить и тот и другой класс перед включением функции **sp_greater()** в каждый из них. Поэтому необходим иной способ сообщить компилятору имя класса без его фактического объявления. Этот способ и называется предварительным объявлением. В C++, чтобы информировать компилятор о том, что данный идентификатор является именем класса, перед первым использованием имени класса вставляют следующую строку:

class ***имя_класса;***

Например, в предыдущей программе предварительным объявлением является инструкция

```
class truck;
```

после которой класс **truck** можно использовать в объявлении дружественной функции **sp_greater()** без опасения вызвать ошибку компилятора.

2. Функция может быть членом одного класса и дружественной другому. Например, если переписать предыдущий пример так, чтобы функция **sp_greater()** являлась членом класса car и дружественной классу **truck,** то получится следующая программа:

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

class truck; // предварительное объявление

class car {
    int passengers;
    int speed;
```

```
public:
    car(int p, int s) { passengers = p; speed = s; }
    int sp_greater(truck t);
};

class truck {
    int weight;
    int speed;
public:
    truck(int w, int s) { weight = w; speed = s; }

    // отметьте новое использование
    // оператора расширения области видимости
    friend int car::sp_greater(truck t);
};

/* Возвращает положительное число, если легковая машина быстрее
грузовика. Возвращает 0 при одинаковых скоростях. Возвращает
отрицательное число, если грузовик быстрее легковой машины.
*/
int car::sp_greater(truck t)
{
/* Поскольку функция sp_greater() — это член класса car, ей должен
передаваться только объект truck
*/
    return speed - t.speed;
}

int main()
{
    int t;
    car c1(6, 55), c2 (2, 120);
    truck t1(10000, 55), t2 (20000, 72);

    cout << "Сравнение значений c1 и t1:\n";
    t = c1.sp_greater(t1); // вызывается как функция-член класса car
    if(t<0) cout << "Грузовик быстрее. \n";
    else if (t=0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    cout << "\nСравнение c2 и t2:\n";
    t = c2.sp_greater(t2); // вызывается как функция-член класса car
    if(t<0) cout << "Грузовик быстрее. \n";
    else if (t=0) cout << "Скорости машин одинаковы. \n";
    else cout << "Легковая машина быстрее. \n";

    return 0;
}
```

Обратите внимание на новое использование оператора расширения области видимости, который имеется в объявлении дружественной функции внутри объявления класса **truck.** В данном случае он информирует компилятор о том, что функция **sp_greater()** является членом класса **car.**

Существует простой способ запомнить, где в такой ситуации нужно указывать оператор расширения области видимости: сначала идет имя класса, потом — оператор расширения области видимости и последним — имя функции-члена. Таким образом член класса будет полностью задан.

Фактически при упоминании в программе члена класса никогда не помешает полностью (с именем класса и оператором расширения области видимости) задать его имя. Однако при использовании объекта для вызова функции-члена или для доступа к переменной-члену, полное имя обычно излишне и употребляется редко. Например, инструкция

```
t = c1.sp_greater(t1);
```

может быть написана с указанием (избыточным) оператора расширения области видимости и именем класса **car:**

```
t = c1.car::sp_greater(t1);
```

Поскольку объект **c1** является объектом типа car, компилятор уже и так знает, что функция **sp_greater()** — это член класса **car,** что делает необязательным полное задание имени класса.

Упражнения

1. Представьте себе ситуацию, в которой показанные ниже два класса **pr1** и pr2 используют общий принтер, а для оставшейся части программы необходимо знать, когда принтер занят объектом одного из этих классов. Создайте функцию inuse(), которая возвращает **true,** когда принтер занят объектом одного из классов и **false** — в противном случае. Сделайте эту функцию дружественной как классу **pr1,** так и классу **pr2.**

```
class pr1 {
    int printing;
    //...
public:
    pr1() { printing = 0; }
    void set_print(int status) { printing = status; }
    //...
};

class pr2 {
    int printing;
    //...
```

```
public:
    pr2() { printing = 0; }
    void set_print(int status) { printing = status; }
    //...
};
```

Перед тем как продолжить, вам необходимо ответить на следующие вопросы и выполнить упражнения:

1. Какое условие является обязательным для присвоения одного объекта другому?
2. Пусть дан следующий фрагмент:

```
class samp {
    double *p;
public:
    samp(double d) {
        p = (double *} malloc(sizeof(double));
        if(!p) exit(1); //ошибка выделения памяти
        *p = d;
    }
    ~samp() { free(p); }
    //...
};
// ...
samp ob1(123.09), ob2(0.0);
// ...
ob2 = ob1;
```

Какую проблему вызовет присваивание объекта **ob1** объекту **ob2**?

3. Дан следующий класс:

```
class planet {
    int moons;
    double dist_from_sun; // в милях
    double diameter;
    double mass;
public:
    //...
    double get_miles() { return dist_from_sun; }
};
```

создайте функцию **light()**, получающую в качестве аргумента объект типа **planet** и возвращающую число секунд, за которые свет достигает планеты. (Предположим, что скорость света равна 186 000 миль в секунду и что значение **dist_from_sun,** т. е. расстояние от Солнца, задано в милях.)

4. Можно ли адрес объекта передать функции в качестве аргумента?

5. Используя класс **stack,** напишите функцию **loadstack()**, которая бы возвращала стек, заполненный буквами алфавита (a-z). В вызывающей программе присвойте этот стек другому объекту и докажите, что и в этом объекте находится алфавит. *(Замечание.* Удостоверьтесь, что длина стека достаточна для хранения алфавита.)

6. Объясните, почему необходимо быть внимательным при передаче объектов функциям или при возвращении объектов из функций.

7. Что такое дружественная функция?

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Функции можно перегружать благодаря отличиям в числе или типе их параметров. Перегрузите функцию **loadstackQ** из вопроса *5* упражнений по проверке усвоения материала данной главы так, чтобы она получала в качестве аргумента целое число **upper.** В перегруженной версии, если переменная **upper** будет равной 1, загрузите стек символами алфавита в верхнем регистре. В противном случае загрузите его символами алфавита в нижнем регистре.

2. Используя класс **strtype**, представленный в разделе 3.1, пример 3, добавьте дружественную функцию, которая получает в качестве аргумента указатель на объект типа **strtype** и возвращает указатель на строку. (Таким образом, функция должна возвращать указатель **p).** Назовите эту функцию **get_string()**.

3. Если объект производного класса присваивается другому объекту того же производного класса, будут ли также копироваться данные, связанные с базовым классом? Чтобы ответить, воспользуйтесь следующими двумя классами и напишите программу.

```
class base {
    int a;
```

```
public:
    void load_a(int n) { a = n; }
    int get_a() { return a; }
};

class derived: public base {
    int b;
public:
    void load_b (int n) { b = n; }
    int get_b() { return b; }
};
```

Глава 4

Массивы, указатели и ссылки

В этой главе исследуется несколько важных аспектов применения массивов объектов и указателей на объекты. Заканчивается глава обсуждением одного из самых важных нововведений C++ — ссылок. Ссылки определяют многие возможности C++, поэтому при чтении нужно быть особенно внимательным.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Что в действительности происходит при присваивании одного объекта другому?
2. Могут ли возникать ошибки или побочные эффекты при присваивании одного объекта другому? (Приведите пример.)
3. Когда объект передается функции в качестве аргумента, создается копия этого объекта. Вызывается ли конструктор копии? Вызывается ли ее деструктор?
4. По умолчанию объект передается функции по значению, это означает, что появляющаяся внутри функции копия объекта не связана с аргументом, используемом при вызове, т. е. изменения копии не отражаются на оригинале. Возможно ли нарушение этого принципа? Если да, приведите пример.
5. Дан следующий класс, создайте функцию **make_sum()**, возвращаемым значением которой является объект типа **summation**. Пользователь должен ввести число, затем функция должна создать объект, получающий это значение, и возвратить его вызвавшей процедуре. Покажите, что функция работает.

```
class summation {
    int num;
    long sum; // суммирование чисел, составляющих num
```

```
public:
    void set_sum(int n) ;
    void show_sum() {
        cout << " сумма чисел, составляющих " << num
                                << " равна " << sum << "\n";
    }
};

void summation: :set_sum(int n)
{
    int i;
    num = n;
    sum = 0;
    for(i=1; i<=n; i++)
        sum += i;
}
```

6. В предыдущем вопросе функция **set_sum()** не была определена как встраиваемая в объявлении класса **summation.** Объясните, почему это необходимо для некоторых компиляторов?

7. Дан следующий класс, покажите, как добавить дружественную функцию **isneg(),** которая получает один параметр типа **myclass** и возвращает **true,** если значение **num** отрицательно и **false** — в противном случае.

```
class myclass {
    int num;
public:
    myclass (int x) { num = x; }
};
```

8. Может ли дружественная функция быть дружественной более чем одному классу?

4.1. Массивы объектов

Как уже отмечалось ранее, объекты — это переменные, и они имеют те же возможности и признаки, что и переменные любых других типов. Поэтому вполне допустимо упаковывать объекты в массив. Синтаксис объявления массива объектов совершенно аналогичен тому, который используется для объявления массива переменных любого другого типа. Более того, доступ к массивам объектов совершенно аналогичен доступу к массивам переменных любого другого типа.

Примеры

1. Пример массива объектов:

```
#include <iostream>
using namespace std;
class samp {
    int a;
public:
    void set_a(int n) { a = n; }
    int get_a() { return a; }
};
int main()
{
    samp ob[4];
    int i;
    for(i=0; i<4; i++) ob[ i ].set_a(i);
    for(i=0; i<4; i++) cout << ob[ i ].get_a();
    cout << "\n";
    return 0;
}
```

В этой программе создается массив из четырех элементов типа **samp,** которым затем присваиваются значения от 0 до 3. Обратите внимание на то, как вызываются функции-члены для каждого элемента массива. Имя массива, в данном случае **ob,** индексируется; затем применяется оператор доступа к члену, за которым следует имя вызываемой функции-члена.

2. Если класс содержит конструктор, массив объектов может быть инициализирован. Например, здесь объект **ob** является инициализируемым массивом:

```
// Инициализация массива
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4] = { -1, -2, -3, -4 };
    int i;
```

```
    for(i=0; i<4; i++) cout << ob[ i ].get_a() << ' ';
    cout << "\n";
    return 0;
}
```

Эта программа выводит на экран -1 -2 -3 -4. В этом примере значения от −1 до −4 передаются объекту ob конструктором.

Фактически синтаксис списка инициализации — это сокращение следующей конструкции (впервые показанной в главе 2):

```
samp ob[4] = { samp(-1), samp(-2), samp(-3), samp(-4) };
```

Однако при инициализации одномерного массива общепринятой является та форма записи, которая была показана в программе (хотя, как вы дальше увидите, такая форма записи будет работать только с теми массивами, конструктор которых имеет единственный аргумент).

3. Вы также можете работать с многомерными массивами объектов. Например, эта программа создает двумерный массив объектов и инициализирует его:

```
// Создание двумерного массива объектов
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4] [2] = {
        1, 2,
        3, 4,
        5, 6,
        7, 8
     };
    int i;

    for(i=0; i<4; i++) {
        cout << ob[i] [0].get_a() << ' ';
        cout << ob[i] [1].get_a() << "\n";
    }
    cout << "\n";
    return 0;
}
```

Эта программа выводит на экран следующее:

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

4. Как вы знаете, конструктор может иметь более одного аргумента. При инициализации массива объектов с таким конструктором вы должны использовать упоминавшуюся ранее альтернативную форму инициализации. Начнем с примера:

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

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4] [2] = {
        samp(1, 2), samp(3, 4),
        samp(5, 6), samp(7, 8),
        samp(9, 10), samp(11, 12),
        samp(13, 14), samp(15, 16)
    };
    int i;

    for(i=0; i<4; i++) {
        cout << ob[i] [0].get_a() << ' ';
        cout << ob[i] [0].get_b() << "\n";
        cout << ob[i] [1].get_a() << ' ';
        cout << ob[i] [1].get_b() << "\n";
    }

    cout << "\n";

    return 0;
}
```

В этом примере конструктор **samp** имеет два аргумента. Здесь массив **ob** объявляется и инициализируется в функции **main()** с помощью прямых вызовов конструктора **samp.** Это необходимо, поскольку формальный синтаксис C++ позволяет одновременное использование только одного аргумента в разделяе-

мом запятыми списке. При этом невозможно задать, например, два или более аргумента в каждом элементе списка. Поэтому если вы инициализируете массив объектов, имеющих конструктор с более чем одним аргументом, то вам следует пользоваться длинной формой инициализации вместо ее сокращенной формы.

Вы всегда можете использовать длинную форму инициализации, даже при наличии у объекта конструктора с одним аргументом. Хотя для этого случая больше подходит сокращенная форма.

Предыдущая программа выводит на экран следующее:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

1. Используя следующее объявление класса, создайте массив из 10 элементов и инициализируйте переменную **ch** значениями от A до J. Покажите, что массив на самом деле содержит эти значения.

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

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};
```

2. Используя следующее объявление класса, создайте массив из 10 элементов, инициализируйте переменную **num** значениями от 1 до 10, а переменную sqr — квадратом **num.**

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

```
class squares {
    int num, sqr;
public:
    squares(int a, int b) { num = a; sqr = b; }
    void show() (cout << num << ' ' << sqr << "\n"; }
};
```

3. Измените инициализацию переменной **ch** из упражнения 1 так, чтобы использовать ее длинную форму (т. е. чтобы конструктор **letters** явно вызывался в списке инициализации).

4.2. Использование указателей на объекты

Как отмечалось в главе 2, доступ к объекту можно получить через указатель на этот объект. Как вы знаете, при использовании указателя на объект к членам объекта обращаются не с помощью оператора точка (.), а с помощью оператора стрелка (->).

Арифметика указателей на объект аналогична арифметике указателей на данные любого другого типа: она выполняется относительно объекта. Например, если указатель на объект инкрементируется, то он начинает указывать на следующий объект. Если указатель на объект декрементируется, то он начинает указывать на предыдущий объект.

1. Пример арифметики указателей на объекты:

```
// Указатели на объекты
ttinclude <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};

int main()
{
    samp ob[4] = {
        samp(1, 2),
```

```
        samp(3,4),
        samp(5,6),
        samp(7, 8)
    };
int i;

samp *p;

p = ob; // получение адреса начала массива

for(i=0; i<4; i++) {
    cout<<p->get_a() << ' ' ;
    cout << p->get_b() << "\n";
    p++; // переход к следующему объекту
}

cout << "\n";

return 0;
}
```

Эта программа выводит на экран следующее:

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

Как видно из результата, при каждом инкрементировании указателя p он указывает на следующий объект массива.

1. Перепишите пример 1 так, чтобы на экран выводилось содержимое массива ob в обратном порядке.
2. Измените пример 3 раздела 4.1 так, чтобы получить доступ к двумерному массиву через указатель. Подсказка: в C++, как и в C, все массивы хранятся непрерывно, слева направо, от младшего элемента к старшему.

4.3. Указатель *this*

C++ содержит специальный указатель **this.** Это указатель, который автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов. Например, рассмотрим следующую инструкцию:

```
ob.f1(); // предположим, что ob — это объект
```

Функции **f1()** автоматически передается указатель на объект **ob.** Этот указатель и называется **this.**

Важно понимать, что указатель **this** передается только функциям-членам. Дружественным функциям указатель **this** не передается.

1. Как вы уже видели, если функция-член работает с другим членом того же класса, она делает это без уточнения имени класса или объекта. Например, исследуйте эту короткую программу, в которой создается простой класс **inventory:**

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
       strcpy(item, i);
       cost = c;
       on_hand = o;
    }
    void show();
};

void inventory::show()
{
    cout << item;
    cout << ": $" << cost;
    cout << "  On hand:  " << on_hand << "\n";
}

int main()
{
    inventory ob("wrench", 4.95, 4);

    ob.show();
    return 0;
}
```

Обратите внимание, что внутри конструктора **inventory()** и функции-члена **show()** переменные-члены **item, cost и on_hand** упоминаются явно. Так происходит потому, что функция-член может вызываться только в связи с объектом. Следовательно, в данном случае компилятор "знает", данные какого объекта имеются в виду.

Однако имеется еще более тонкое объяснение. Если вызывается функция-член, ей автоматически передается указатель **this** на объект, который является источником вызова. Таким образом, предыдущую программу можно переписать так:

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
    char item[20];
    double cost;
    int on_hand;
public:
    inventory(char *i, double c, int o)
    {
        strcpy(this->item, i) ; // доступ к члену
        this->cost = c;         // через
        this->on_hand = o;      // указатель this
    }
    void,show();
};

voidinventory::show()
{
    cout << this->item; // использование this для доступа к членам
    cout << ": $" << this->cost;
    cout << "  On hand:  " << this->on_hand << "\n";
}'

int main()
{
    inventory ob("wrench", 4.95, 4) ;

    ob.show();

    return 0;
}
```

Здесь к переменным-членам объекта **ob** осуществляется прямой доступ через указатель **this.** Таким образом, внутри функции **show()** следующие две инструкции равнозначны:

```
cost = 123.23;
this->cost = 123.23;
```

На самом деле первая форма — это сокращенная запись второй.

Пока, наверное, еще не родился программист C++, который бы использовал указатель **this** для доступа к членам класса так, как было показано, поскольку сокращенная форма намного проще, но здесь важно понимать, что под этим сокращением подразумевается.

Использовать указатель **this** можно по-разному. Он особенно полезен при перегрузке операторов. Такое его применение более подробно будет изучаться в главе 6. На данный момент важно то, что по умолчанию всем функциям-членам автоматически передается указатель на вызывающий объект.

1. Дана следующая программа, переделайте все соответствующие обращения к членам класса так, чтобы в них явно присутствовал указатель **this.**

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

class myclass {
    int a, b;
public:
    myclass(int n, int m) { a = n; b = m; }
    int add() { return a + b; }
    void show();
};

void myclass::show();
{
    int t;

    t = add(); // вызов функции-члена
    cout << t <<"\n";
}
int main()
{
    myclass ob(10, 14);

    ob.show();

    return 0;
}
```

4.4. Операторы *new* и *delete*

До сих пор при выделении динамической памяти вы использовали функцию **malloc()**, а при освобождении памяти — функцию **free().** Вместо этих стандартных функций в C++ стал применяться более безопасный и удобный способ выделения и освобождения памяти. Выделить память можно с помощью оператора new, а освободить ее с помощью оператора **delete.** Ниже представлена основная форма этих операторов:

```
p-var = new type;
delete p-var;
```

Здесь *type* — это спецификатор типа объекта, для которого вы хотите выделить память, а ***p-var*** — указатель на этот тип. New — это оператор, который возвращает указатель на динамически выделяемую память, достаточную для хранения объекта типа ***type.*** Оператор **delete** освобождает эту память, когда в ней отпадает необходимость. Вызов оператора **delete** с неправильным указателем может привести к разрушению системы динамического выделения памяти и возможному краху программы.

Если свободной памяти недостаточно для выполнения запроса, произойдет одно из двух: либо оператор new возвратит нулевой указатель, либо будет сгенерирована исключительная ситуация. (Исключительные ситуации и обработка исключительных ситуаций описываются далее в этой книге. Коротко об исключительной ситуации можно сказать следующее — это динамическая ошибка, которую можно обработать определенным образом.) В соответствии с требованиями языка Standart C++ по умолчанию оператор new должен генерировать исключительную ситуацию при невозможности удовлетворить запрос на вьщеление памяти. Если ваша программа не обрабатывает эту исключительную ситуацию, выполнение программы прекращается. К сожалению, точные требования к тому, какие действия должны выполняться, если оператор new не в состоянии удовлетворить запрос на вьщеление памяти, за последние годы менялись несколько раз. Поэтому вполне возможно, что в вашем компиляторе реализация оператора new выполнена не так, как это предписано стандартом Standart C++.

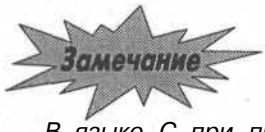
Когда C++ только появился, при невозможности удовлетворить запрос на вьщеление памяти оператор new возвращал нулевой указатель. В дальнейшем эта ситуация изменилась, и при неудачной попытке выделения памяти оператор new стал генерировать исключительную ситуацию. В конце концов было принято решение, что при неудачной попытке выделения памяти оператор new будет генерировать исключительную ситуацию по умолчанию, а возвращение нулевого указателя останется в качестве возможной опции. Таким образом, реализация оператора new оказалась разной у разных производителей компиляторов. К примеру, во время написания этой книги в компиляторе Microsoft Visual C++ при невозможности удовлетворить запрос на вьщеление памяти оператор new возвращал нулевой указатель, а в компиляторе Borland C++ генерировал исключительную ситуацию. Хотя в буду-

щем во всех компиляторах оператор **new** будет реализован в соответствии со стандартом Standart C++, в настоящее время единственным способом узнать, какие именно действия он выполняет при неудачной попытке выделить память, является чтение документации на компилятор.

Поскольку имеется два возможных способа, которыми оператор **new** может сигнализировать об ошибке выделения памяти, и поскольку в разных компиляторах он может быть реализован по-разному, в примерах программ этой книги сделана попытка удовлетворить оба требования. Во всех примерах значение возвращаемого оператором **new** указателя проверяется на равенство нулю. Такое значение указателя обрабатывается теми компиляторами, в которых при неудачной попытке выделить память оператор **new** возвращает нуль, хотя никак не влияет на те, в которых оператор **new** генерирует исключительную ситуацию. Если в вашем компиляторе во время выполнения программы при неудачной попытке выделить память оператор **new** сгенерирует исключительную ситуацию, то такая программа просто завершится. В дальнейшем, когда вы ближе познакомитесь с обработкой исключительных ситуаций, мы вернемся к оператору new, и вы узнаете, как лучше обрабатывать неудачные попытки выделения памяти. Вы также узнаете об альтернативной форме оператора **new,** который при наличии ошибки всегда возвращает нулевой указатель.

И последнее замечание: ни один из примеров программ этой книги не должен вести к ошибке выделения памяти при выполнении оператора **new,** поскольку в каждой конкретной программе выделяется лишь считанное число байтов.

Хотя операторы **new** и **delete** выполняют сходные с функциями **malloc() и free()** задачи, они имеют несколько преимуществ перед ними. Во-первых, оператор **new** автоматически выделяет требуемое для хранения объекта заданного типа количество памяти. Вам теперь не нужно использовать **sizeof**, например, для подсчета требуемого числа байтов. Это уменьшает вероятность ошибки. Во-вторых, оператор **new** автоматически возвращает указатель на заданный тип данных. Вам не нужно выполнять приведение типов, операцию, которую вы делали, когда выделяли память, с помощью функции **malloc()** (см. следующее замечание). В-третьих, как оператор **new,** так и оператор **delete** можно перегружать, что дает возможность простой реализации вашей собственной, привычной модели распределения памяти. В-четвертых, допускается инициализация объекта, для которого динамически выделена память. Наконец, больше нет необходимости включать в ваши программы заголовок **<cstdlib>.**



*В языке C при присваивании указателю возвращаемого функцией **malloc()** значения не требуется приведение типов, поскольку тип **void** *(тип возвращаемого функцией **malloc()** указателя) автоматически преобразуется в тип,*

совместимый с типом указателя, стоящего слева от знака равенства. Однако для C++ этот случай не подходит. При использовании функции ***malloc()*** *требуется полное приведение типов. Смысл этого отличия в том, что оно позволяет C++ усилить контроль типа при возвращении функцией значения.*

Теперь, после введения операторов **new** и **delete,** они будут использоваться в программах вместо функций **malloc() и free().**

1. Для начала рассмотрим программу выделения памяти для хранения целого:

```
// Простой пример операторов new и delete
ttinclude <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int; // выделение памяти для целого
if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    *p = 1000;

    cout << "Это целое, на которое указывает p: " << *p << "\n";

    delete p; // освобождение памяти

    return 0;
}
```

Обратите внимание, что возвращаемое оператором new значение перед использованием проверяется. Как уже упоминалось, эта проверка имеет значение только в том случае, если в вашем компиляторе при неудачной попытке выделения памяти оператор new возвращает нулевой указатель.

2. Пример, в котором память объекту выделяется динамически:

```
// Динамическое выделение памяти объектам
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i = a; j = b; }
```

```
    int get_product() { return i*j; }
};

int main()
{
    samp *p;

    p = new samp; // выделение памяти объекту
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    p->set_ij( 4, 5);

    cout << "Итог равен:" << p->get_product() << "\n";

    return 0;
}
```

1. Напишите программу, в которой оператор new используется для динамического размещения переменных типа **float, long и char.** Задайте этим динамическим переменным значения и выведите эти значения на экран. В завершение с помощью оператора **delete** освободите всю динамически выделенную область памяти.
2. Создайте класс для хранения своего имени и номера телефона. Используя оператор **new,** динамически выделите память для объекта этого класса и введите имя и телефон в соответствующие поля внутри объекта.
3. Какими двумя способами оператор **new** может известить вас о неудачной попытке выделения памяти?

4.5. Дополнительные сведения об операторах new и *delete*

В этом разделе обсуждаются два дополнительных свойства операторов **new** и **delete.** Во-первых, динамически размещаемому объекту может быть присвоено начальное значение. Во-вторых, можно создавать динамически размещаемые массивы объектов.

Вы можете присвоить динамически размещаемому объекту начальное значение, используя следующую форму оператора new:

p-var = **new type (*начальное_значение*);**

Для динамически размещаемого одномерного массива используйте такую форму оператора **new:**

p-var = new *type* **[*size*];**

После выполнения этого оператора указатель ***p-var*** будет указывать на начальный элемент массива из ***size*** элементов заданного типа. Из-за разных чисто технических причин невозможно инициализировать массив, память для которого выделена динамически.

Для удаления динамически размещенного одномерного массива вам следует использовать следующую форму оператора **delete:**

delete **[]** ***p-var;***

При таком синтаксисе компилятор вызывает деструктор для каждого элемента массива. Это не приводит к многократному освобождению памяти по адресу, обозначенному указателем ***p-var,*** она освобождается только один раз.

*Для устаревших компиляторов в операторе **delete** было необходимо указывать в квадратных скобках размер освобождаемого массива. Так требовало исходное определение C++. В современных компиляторах задавать размер массива не нужно.*

1. В следующей программе выделяется и инициализируется память для хранения целого:

```
// Пример инициализации динамической переменной
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int(9); // задание начального значения равного 9

    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    cout << "Это целое, на которое указывает p: " << *p << "\n";
```

```
    delete p; // освобождение памяти

    return 0;
}
```

Как и следовало ожидать, программа выводит на экран число 9, являющееся начальным значением переменной, на которую указывает указатель p.

2. Следующая программа инициализирует динамически размещаемый объект:

```
// Динамическое выделение памяти объектам
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    samp(int a, int b) { i = a; j = b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;

    p = new samp(6, 5); // размещение объекта с инициализацией
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    cout << "Итог равен:" << p->get_product() << "\n";

    delete p;

    return 0;
}
```

При размещении объекта **samp** автоматически вызывается его конструктор, и объекту передаются значения 6 и 5.

3. В следующей программе размещается массив целых:

```
// Простой пример использования операторов new и delete
#include <iostream>
using namespace std;

int main()
{
    int *p;

    p = new int [5]; // выделение памяти для 5 целых
```

```
    // Убедитесь, что память выделена
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }
    int i;

    for(i=0; i<5; i++) p[i] = i;

    for(i=0; i<5; i++) {
        cout << "Это целое, на которое указывает p[" << i << "]: ";
        cout << p[i] << "\n";
  }

    delete [] p; // освобождение памяти

    return 0;
}
```

Эта программа выводит на экран следующее:

```
Это целое, на которое указывает p[0]: 0
Это целое, на которое указывает p[1]: 1
Это целое, на которое указывает p[2]: 2
Это целое, на которое указывает p[3]: 3
Это целое, на которое указывает p[4]: 4
```

4. В следующей программе создается динамический массив объектов:

```
// Динамическое выделение памяти для массива объектов
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i = a; j = b; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;
    int i;

    p = new samp [10] ; // размещение массива объектов
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }
```

```
    for(i=0; i<10; i++)
        p[i].set_ij(i, i);

    for(i=0; i<10; i++) {
        cout << "Содержимое [" << i << "] равно:";
        cout << p[i].get_product() << "\n";
    }
    delete [] p;
    return 0;
}
```

Эта программа выводит на экран следующее:

```
Содержимое [0] равно: 0
Содержимое [1] равно: 1
Содержимое [2] равно: 4
Содержимое [3] равно: 9
Содержимое [4] равно: 16
Содержимое [5] равно: 25
Содержимое [6] равно: 36
Содержимое [7] равно: 49
Содержимое [8] равно: 64
Содержимое [9] равно: 81
```

5. В новой версии предыдущей программы в нее вводится деструктор **samp** и теперь при освобождении памяти, обозначенной указателем p, для каждого элемента массива вызывается деструктор:

```
// Динамическое выделение памяти для массива объектов
#include <iostream>
using namespace std;

class samp {
    int i, j;
public:
    void set_ij(int a, int b) { i = a; j = b; }
    ~samp() { cout << "Удаление объекта...\n"; }
    int get_product() { return i*j; }
};

int main()
{
    samp *p;
    int i;

    p = new samp [10]; // размещение массива объектов
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }
```

```
    for(i=0; i<10; i++)
        p[i].set_ij(i, i);

    for(i=0; i<10; i++) {
        cout << "Содержимое [" << i << "] равно:";
        cout << p[i].get_product() << "\n";
    }
    delete [] p;
    return 0;
}
```

Эта программа выводит на экран следующее:

```
Содержимое [0] равно: 0
Содержимое [1] равно: 1
Содержимое [2] равно: 4
Содержимое [3] равно: 9
Содержимое [4] равно: 16
Содержимое [5] равно: 25
Содержимое [6] равно: 36
Содержимое [7] равно: 49
Содержимое [8] равно: 64
Содержимое [9] равно: 81
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
```

Как видите, деструктор **samp** вызывается десять раз — по разу на каждый элемент массива.

1. Переделайте данную программу так, чтобы в ней использовался оператор **new.**

```
char *p;

p = ( char *) malloc(100);
// ...
strcpy(p, "Это проверка");
```

Подсказка. Строка — это просто массив символов.

2. Используя оператор **new,** покажите, как динамически разместить переменную типа **double и** передать ей начальное значение -123.0987.

4.6. Ссылки

В C++ есть элемент, родственный указателю — это *ссылка (reference)*. Ссылка является скрытым указателем и во всех случаях, и для любых целей ее можно употреблять просто как еще одно имя переменной. Ссылку допустимо использовать тремя способами. Во-первых, ссылку можно передать в функцию. Во-вторых, ссылку можно возвратить из функции. Наконец, можно создать независимую ссылку. В книге рассмотрены все эти применения ссылки, начиная со ссылки в качестве параметра функции.

Несомненно, наиболее важное применение ссылки — это передача ее в качестве параметра функции. Чтобы помочь вам разобраться в том, что такое параметр-ссылка и как он работает, начнем с программы, в которой параметром является указатель (а не ссылка):

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

void f(int *n) ; // использование параметра-указателя

int main()
{
    int i = 0;

    f(&i);

    cout << "Новое значение i:  " << i << 'n\';

    return 0;
}

void f (int *n)
{
    *n = 100;  // занесение числа 100 в аргумент,
               // на который указывает указатель п
}
```

Здесь функция f() загружает целое значение 100 по адресу, который обозначен указателем п. В данной программе функция **f()** вызывается из функции main() с адресом переменной i. Таким образом, после выполнения функции **f()** переменная i будет содержать число 100.

В этой программе показано, как использовать указатель для реализации механизма передачи параметра посредством вызова по ссылке (call by reference). В программах C такой механизм является единственным спосо-

бом добиться вызова функции по ссылке. Однако в C++ с помощью параметра-ссылки можно полностью автоматизировать весь процесс. Чтобы узнать, как это сделать, изменим предыдущую программу. В ее новой версии используется параметр-ссылка:

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

void f(int &n) ; // объявление параметра-ссылки

int main()
{
    int i = 0;

    f(i);

    cout << "Новое значение i: " << i << 'n\';

    return 0;
}

// Теперь в функции f() используется параметр-ссылка
void f(int &n)
{
    // отметьте, что в следующей инструкции знак * не требуется
    n = 100; // занесение числа 100 в аргумент,
             // используемый при вызове функции f()
}
```

Тщательно проанализируйте эту профамму. Во-первых, для объявления параметра-ссылки перед именем переменной ставится знак амперсанда (&). Таким образом, переменная n объявляется параметром функции **f().** Теперь, поскольку переменная n является ссылкой, больше не нужно и даже неверно указывать оператор *. Вместо него всякий раз, когда переменная n упоминается внутри функции **f()**, она автоматически трактуется как указатель на аргумент, используемый при вызове функции **f().** Это значит, что инструкция

```
n = 100;
```

фактически помещает число 100 в переменную, используемую при вызове функции f(), каковой в данном случае является переменная i. Далее, при вызове функции **f()** перед аргументом не нужно ставить знак &. Вместо этого, поскольку функция **f()** объявлена как получающая параметр-ссылку, ей *автоматически* передается адрес аргумента.

Повторим, при использовании параметра-ссылки компилятор автоматически передает функции адрес переменной, указанной в качестве аргумента. Нет необходимости (а на самом деле и не допускается) получать адрес аргумента с помощью знака &. Более того, внутри функции компилятор автома-

тически использует переменную, на которую указывает параметр-ссылка. Нет необходимости (и опять не допускается) ставить знак *. Таким образом, параметр-ссылка полностью автоматизирует механизм передачи параметра посредством вызова функции по ссылке.

Важно понимать следующее: адрес, на который указывает ссылка, вы изменить не можете. Например, если в предыдущей программе инструкция

```
n++;
```

находилась бы внутри функции **f(),** ссылка n по-прежнему указывала бы на переменную i в функции **main()** Вместо инкрементирования адреса, на который указывает ссылка n, эта инструкция инкрементирует значение переменной (в данном случае это переменная i).

Параметры-ссылки имеют несколько преимуществ по сравнению с аналогичными (более или менее) альтернативными параметрами-указателями. Во-первых, с практической точки зрения нет необходимости получать и передавать в функцию адрес аргумента. При использовании параметра-ссылки адрес передается автоматически. Во-вторых, по мнению многих программистов, параметры-ссылки предлагают более понятный и элегантный интерфейс, чем неуклюжий механизм указателей. В-третьих, как вы увидите в следующем разделе, при передаче объекта функции через ссылку копия объекта не создается. Это уменьшает вероятность ошибок, связанных с построением копии аргумента и вызовом ее деструктора.

1. Классическим примером передачи аргументов по ссылке является функция, меняющая местами значения двух своих аргументов. В данном примере в функции **swapargs()** ссылки используются для того, чтобы поменять местами два ее целых аргумента:

```
ttinclude<iostream>
using namespace std;

void swapargs(int &x, int &y);

int main()
{
    int i, j;

    i = 10;
    j - 19;

    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";

    swapargs(i, j);
```

```
    cout << "После перестановки: ";
    cout << "i: " << i << ", ";
    cout << "j: " << j << "\n";

    return 0;
}

void swapargs(int &x, int &y)
{
    int t;

    t = x;
    x = y;
    y = t;
}
```

Если бы при написании функции **swapargs()** вместо ссылок использовались указатели, то функция выглядела бы следующим образом:

```
void swapargs(int *x, int *y)
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}
```

Как видите, благодаря использованию ссылок в функции **swapargs()**, отпадает необходимость указывать знак *.

2. В следующей программе с помощью функции **round()** округляется значение типа **double.** Округляемое значение передается по ссылке.

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

void round(double &num);

int main()
{
    double i = 100.4;

    cout << i << "после округления ";
    round(i);
    cout << i << "\n";

    i = 10.9;
    cout << i << "после округления ";
    round(i);
    cout << i << "\n";
```

```
    return 0;
}

void round(double &num)
{
    double frac;
    double val;

    // разложение num на целую и дробную части
    frac = modf(num, &val);

    if(frac < 0.5) num = val;
    else num = val + 1.0;
}
```

В функции **round()** для разложения числа на целую и дробную части указана редко используемая функция **modf()**. Возвращаемым значением этой функции является дробная часть; целая часть помещается в переменную, на которую указывает второй параметр функции **modf()**.

Упражнения

1. Напишите функцию neg(), которая меняет знак своего целого параметра на противоположный. Напишите функцию двумя способами: первый — используя параметр-указатель и второй — параметр-ссылку. Составьте короткую программу для демонстрации обеих функций.
2. Что неправильно в следующей программе?

```
// В этой программе есть ошибка
#include <iostream>
using namespace std;

void triple(double &num);

int main()
{
    double d = 7.0;

    triple(&d);

    cout << d;

    return 0;
}

// Утроение значения числа
void triple(double &num)
```

```
{
    num = 3 * num;
}
```

3. Перечислите преимущества параметров-ссылок.

4.7. Передача ссылок на объекты

Как вы узнали из главы 3, если объект передается функции по значению, то в функции создается его копия. Хотя конструктор копии не вызывается, при возвращении функцией своего значения вызывается деструктор копии. Повторный вызов деструктора может привести в некоторых случаях к серьезным проблемам (например, при освобождении деструктором динамической памяти).

Решает эту проблему передача объекта по ссылке. (Другое решение, о котором будет рассказано в главе 5, подразумевает использование конструктора копий.) При этом копия объекта не создается, и поэтому при возвращении функцией своего значения деструктор не вызывается. Тем не менее, запомните: изменения объекта внутри функции влияют на исходный объект, указанный в качестве аргумента функции.

Очень важно понимать, что ссылка не указатель, хотя она и указывает на адрес объекта. Поэтому при передаче объекта по ссылке для доступа к его членам используется оператор точка (.), а не стрелка (->).

1. В следующем примере на экран выводится значение, передаваемое объекту по ссылке, но сначала рассмотрим версию программы, в которой объект типа **myclass()** передается в функцию **f()** по значению:

```
ttinclude <iostream>
using namespace std;

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Работа конструктора " << who << "\n";
    }
```

```
    ~myclass() { cout << "Работа деструктора " << who << "\n"; }
    int id() { return who; }
};

// o передается по значению
void f(myclass o)
{
    cout << "Получено" << o.id() << "\n";
}

int main()
{
    myclass x(1);

    f(x);

    return 0;
}
```

Эта функция выводит на экран следующее:

```
Работа конструктора 1
Получено 1
Работа деструктора 1
Работа деструктора 1
```

Как видите, деструктор вызывается дважды: первый раз, когда после выполнения функции **f()** удаляется копия объекта 1, а второй раз — по окончании программы.

С другой стороны, если изменить программу так, чтобы использовать параметр-ссылку, то копия объекта не создается и поэтому после выполнения функции **f()** деструктор не вызывается:

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

class myclass {
    int who;
public:
    myclass(int n) {
        who = n;
        cout << "Работа конструктора " << who << "\n";
    }
    ~myclass() { cout << "Работа деструктора " << who << "\n"; }
    int id (){ return who; }
};

// Теперь o передается по ссылке
void f(myclass &o)
```

```
{
    // отметьте, что по-прежнему используется оператор
    cout << "Получено" << o.id() << "\n";
}
int main()
{
    myclass x(1);
    f(x);
    return 0;
}
```

Эта версия предыдущей программы выводит на экран следующее:

```
Работа конструктора 1
Получено 1
Работа деструктора 1
```

Для доступа к членам объекта по ссылке следует указывать оператор точка (.), а не стрелка (->).

1. **Что неправильно в следующей программе? Покажите, как она может быть исправлена с помощью параметра-ссылки.**

```
// В этой программе есть ошибка
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;
```

```
    l = strlen(s) + 1;

    p = new char [l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Привет"), b("Здесь");

    show(a);
    show(b);

    return 0;
}
```

4.8. Ссылка в качестве возвращаемого значения функции

Функция может возвращать ссылку. Как вы увидите в главе 6, возвращение ссылки может оказаться полезным при перегрузке операторов определенных типов. Кроме этого возвращение ссылки позволяет использовать функцию слева в инструкции присваивания. Это приводит к важному и неожиданному результату.

1. Для начала, простая программа с функцией, которая возвращает ссылку:

```
// Простой пример ссылки в качестве возвращаемого значения функции
#include <iostream>
using namespace std;
```

```
int &f() ;
int x;

int main()
{
    f() = 100; // присваивание 100 ссылке, возвращаемой функцией f()

    cout << x << "\n";

    return 0;
}

// Возвращение ссылки на целое
int &f()
{
    return x; // возвращает ссылку на x
}
```

Здесь функция f() объявляется возвращающей ссылку на целое. Внутри тела функции инструкция

```
return x;
```

не возвращает значение глобальной переменной x, она автоматически возвращает адрес переменной x (в виде ссылки). Таким образом, внутри функции **main()** инструкция

```
f() = 100;
```

заносит значение 100 в переменную x, поскольку функция f() уже возвратила ссылку на нее.

Повторим, функция f() возвращает ссылку. Когда функция f() указана слева в инструкции присваивания, то таким образом слева оказывается ссылка на объект, которую возвращает эта функция. Поскольку функция f() возвращает ссылку на переменную x (в данном примере), то эта переменная x и получает значение 100.

2. Вам следует быть внимательными при возвращении ссылок, чтобы объект, на который вы ссылаетесь, не вышел из области видимости. Например, рассмотрим эту, слегка переделанную функцию f():

```
// Возвращение ссылки на целое
int &f()
{
    int x; // x – локальная переменная
    return x; // возвращение ссылки на x
}
```

В этом случае x становится локальной переменной функции f() и выходит из области видимости после выполнения функции. Это означает, что ссылку, возвращаемую функцией f(), уже нельзя использовать.

Некоторые компиляторы C++ не позволяют возвращать ссылки на локальные переменные. Тем не менее, эта проблема может проявиться по-другому, например, когда память для объектов выделяется динамически.

3. В качестве возвращаемого значения функции ссылка может оказаться полезной при создании массива определенного типа — так называемого защищенного массива (bounded array). Как вы знаете, в C и C++ контроль границ массива не производится. Следовательно, имеется вероятность при заполнении массива выйти за его границы. Однако в C++ можно создать класс массива с автоматическим контролем границ (automatic bounds checking). Любой класс массива содержит две основные функции — одну для запоминания информации в массиве и другую для извлечения информации. Именно эти функции в процессе работы могут проверять, не нарушены ли границы массива.

   Следующая программа реализует контроль границ символьного массива:

```
// Пример защищенного массива
#include <iostream>
ttinclude<cstdlib>
using namespace std;

class array {
    int size;
    char *p;
public:
    array(int num);
    ~ array() { delete [] p; }
    char &put(int i);
    char get(int i);
};

array::array(int num)
{
    p = new char [num];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit (1);
    }
    size = num;
}

// Заполнение массива
char &array::put(int i)
```

```
{
    if(i<0 | | i>=size) {
        cout << "Ошибка, нарушены границы массива! ! !\n";
        exit(1);
    }
    return p[i]; // возврат ссылки на p[i]
}

// Получение чего-нибудь из массива
char array::get(int i)
{
    if(i<0 I I i>=size) {
        cout << "Ошибка, нарушены границы массива! !!\n";
        exit(1);
    }
    return p[i]; // символ возврата
}

int main()
{
    array a(10);

    a.put(3) = 'X';
    a.put(2) = 'R';

    cout << a.get(3) << a.get(2);
    cout << "\n";

    /* теперь генерируем динамическую ошибку, связанную с нарушением
границ массива */
    a.put(11) = '!';

    return 0;
}
```

Это был пример практического применения ссылок в качестве возвращаемого значения функций, и вам следует тщательно его изучить. Отметьте, что функция **put()** возвращает ссылку на элемент массива, заданный параметром i. Если индекс, заданный параметром i, не выходит за границы массива, то чтобы поместить значение в массив, эту ссылку можно использовать слева в инструкции присваивания. Обратной функцией является функция get(), которая возвращает заполненное по заданному индексу значение, если этот индекс находится внутри диапазона. При таком подходе к работе с массивом он иногда упоминается как *безопасный массив (safe array)*.

Имеется еще одна вещь, которую следует отметить в предыдущей программе, — это использование оператора **new** для динамического выделения памяти. Этот оператор дает возможность объявлять массивы различной длины.

Как уже упоминалось, способ контроля границ массива, реализованный в программе, является примером практического применения C++. Если вам

необходимо во время работы программы проверять границы массива, такой способ позволяет вам легко этого добиться. Тем не менее, запомните: контроль границ замедляет доступ к массиву. Поэтому контроль границ лучше включать в программу только в том случае, если имеется высокая степень вероятности нарушения границ массива.

1. Напишите программу, которая создает безопасный двумерный (2x3) массив целых. Покажите, как она работает.
2. Правилен ли следующий фрагмент? Если нет, то почему?

```
int &f();
.
.
.
int *x;
x = f();
```

4.9. Независимые ссылки и ограничения на применение ссылок

Хотя они обычно и не используются, вы можете создавать *независимые ссылки (independent reference)*. Независимая ссылка — это ссылка, которая во всех случаях является просто другим именем переменной. Поскольку ссылкам нельзя присваивать новые значения, независимая ссылка должна быть инициализирована при ее объявлении.

Поскольку независимые ссылки все-таки иногда используются, важно, чтобы вы имели о них представление. Однако большинство программистов чувствует их ненужность, они просто добавляют неразберихи в программу. Более того, независимые ссылки существуют в C++ только потому, что пока не было достаточного повода избавиться от них. Как правило, их следует избегать.

Имеется несколько ограничений, которые относятся к ссылкам всех типов. Нельзя ссылаться на другую ссылку. Нельзя получить адрес ссылки. Нельзя создавать массивы ссылок и ссылаться на битовое поле. Ссылка должна быть инициализирована до того, как стать членом класса, возвратить значение функции или стать параметром функции.

Ссылки похожи на указатели, но это не указатели.

1. Пример программы с независимой ссылкой:

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

int main()
{
    int x;
    int &ref = x;  // создание независимой ссылки

    x = 10;        // эти две инструкции
    ref = 10;      // функционально идентичны

    ref = 100;
    // здесь дважды печатается число 100
    cout << x << ' ' << ref <<"\n";

    return 0;
}
```

   В этой программе независимая ссылка **ref** служит другим именем переменной x. С практической точки зрения **ref** и x идентичны.

2. Независимая ссылка может ссылаться на константу. Например, следующая инструкция вполне допустима:

```
const int &ref = 10;
```

   В ссылках такого типа выгода невелика, но иногда их можно встретить в программах.

1. Попытайтесь найти полезное применение для независимой ссылки.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы:

1. Ниже представлен класс **a_type**. Создайте двумерный, два на пять, массив и дайте каждому элементу массива начальное значение по своему выбору. Затем выведите содержимое массива на экран.

```
class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << ' ' << b << "\n"; }
};
```

2. Модифицируйте решение предыдущей задачи так, чтобы доступ к массиву осуществлялся через указатель.
3. Что такое указатель **this?**
4. Покажите основные формы операторов **new** и **delete.** Какие преимущества они дают в сравнении с функциями **malloc() и free()?**
5. Что такое ссылка? Какое можно получить преимущество от использования ссылки в качестве параметра?
6. Создайте функцию **recip(),** которая получает один параметр-ссылку на значение типа **double.** Эта функция должна изменить значение своего параметра на обратное. Напишите программу вывода на экран результатов работы функции.

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Пусть дан указатель на объект. Какой оператор использовать для доступа к члену объекта?
2. В главе 2 была создана программа с классом **strtype,** в которой память для строки выделялась динамически. Переделайте программу (показанную здесь для удобства) так, чтобы в ней использовались операторы **new** и **delete.**

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

class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype();
    void show();
};

strtype::strtype(char*ptr)
{
    len = strlen(ptr)
    p=(char *) malloc(len+1);
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

3. Переделайте любую программу из предыщущей главы так, чтобы в ней использовались ссылки.

Глава 5

Перегрузка функций

В этой главе вы более подробно изучите перегрузку функций. Хотя с этой темой вы уже встречались, имеется несколько дополнительных аспектов, с которыми необходимо познакомиться. Здесь вы найдете ответы на следующие вопросы: как перефузить конструктор, как создать конструктор копий, как функции передать аргументы по умолчанию, как можно избежать неоднозначности при перегрузке функций.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Что такое ссылка? Приведите две важных области применения ссылок?
2. Покажите, как с помощью оператора **new** выделить память для значений типа **float** и **int.** Покажите также, как освободить память с помощью оператора **delete.**
3. Какова основная форма оператора new, используемая для инициализации динамических переменных? Приведите конкретный пример.
4. Дан следующий класс. Как инициализировать массив из десяти элементов так, чтобы переменная x получила значения от 1 до 10 включительно.

```
class samp {
    int x;
public:
    sump(int i) { x = n; }
    int getx() { return x; }
};
```

5. Перечислите достоинства и недостатки параметров-ссылок.
6. Может ли быть инициализирован массив, память для которого выделяется динамически?

7. На основе следующего прототипа создайте функцию **mag()**, повышающую порядок значения переменной **num** до того уровня, который задан переменной **order:**

```
void mag(long &num, long order);
```

Например, если переменная **num** равна 4, а переменная **order** равна 2, то после выполнения функции **mag()** переменная **num** должна стать равной 400. Напишите демонстрационную программу, показывающую, что функция работает.

5.1. Перегрузка конструкторов

В программах на C++ перегрузка конструктора класса вполне обычна. (Деструктор, однако, перегружать нельзя.) Имеется три основных причины перегрузки конструктора, которая, как правило, выполняется либо для обеспечения гибкости, либо для поддержки массивов, либо для создания конструкторов копий. В этом разделе рассказывается об обеспечении гибкости и поддержке массивов, а о конструкторах копий — в следующем.

Перед изучением примеров необходимо запомнить одну вещь: каждому способу объявления объекта класса должна соответствовать своя версия конструктора класса. Если эта задача не решена, то при компиляции программы обнаружится ошибка. Именно поэтому перегрузка конструктора столь обычна для программ C++.

1. Вероятно, наиболее частое использование перегрузки конструктора — это обеспечение возможности выбора способа инициализации объекта. Например, в следующей программе объекту **o1** дается начальное значение, а объекту o2 — нет. Если вы удалите конструктор с пустым списком аргументов, программа не будет компилироваться, поскольку у неинициализируемого объекта типа **samp** не будет конструктора. И наоборот, если вы удалите конструктор с параметром, программа не будет компилироваться, поскольку не будет конструктора у инициализируемого объекта типа **samp.** Для правильной компиляции программы необходимы оба конструктора.

```
ttinclude <iostream>
using namespace std;

class myclass {
    int x;
public:
    // перегрузка конструктора двумя способами
    myclass() { x = 0; } // нет инициализации
```

```
    myclass(int n) { x = n; } // инициализация
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // объявление с начальным значением
    myclass o2; // объявление без начального значения

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}
```

2. Другой традиционный довод в пользу перегрузки конструктора состоит в том, что такая перегрузка позволяет сосуществовать в программе как отдельным объектам, так и массивам объектов. Как вы, наверное, знаете по своему опыту программирования, вполне обычно инициализировать отдельную переменную, тогда как инициализация массива встречается достаточно редко. (Гораздо чаще элементам массива присваиваются их значения в зависимости от информации, получаемой уже при выполнении программы.) Таким образом, для сосуществования в программе неинициализированных массивов объектов наряду с инициализированными объектами вы должны использовать конструктор, который поддерживает инициализацию, и конструктор, который ее не поддерживает.

   Например, для класса **myclass** из примера 1 оба этих объявления правильны:

```
myclass ob(10);
myclass ob[5];
```

   Обеспечив наличие обоих конструкторов (с параметрами и без параметров), вы в своей программе получаете возможность создавать объекты, которые при необходимости можно либо инициализировать, либо нет.

   Естественно, что после определения конструктора с параметрами и конструктора без параметров, их можно использовать для создания инициализированных или неинициализированных массивов. Например, в следующей программе объявляются два массива типа **myclass;** при этом один из них инициализируется, а другой нет:

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

class myclass {
    int x;
public:
    // перегрузка конструктора двумя способами
    myclass() { x = 0; } // нет инициализации
```

```
    myclass(int n) { x = n; } // инициализация
    int getx() { return x; }
};

int main()
{
    myclass o1[10]; // объявление массива без инициализации

    // объявление с инициализацией
    myclass o2[10] = {1,2,3,4,5,6,7,8,9,10};

    int i;

    for(i=0; i<10; i++) {
        cout << "o1[ " << i << "]:" << o1[i].getx();
        cout << '\n';
        cout << "o2[ " << i << "]:" << o2[i].getx();
        cout << '\n';
    }

    return 0;
}
```

В этом примере все элементы массива **ol** конструктор устанавливает в нуль. Элементы массива о2 инициализируются так, как показано в программе.

3. Другой довод в пользу перегрузки конструкторов состоит в том, что такая перегрузка позволяет программисту выбрать наиболее подходящий метод инициализации объекта. Чтобы понять, как это делается, рассмотрим следующий пример, в котором создается класс для хранения календарной даты. Конструктор **date()** перегружается двумя способами. В первом случае данные задаются в виде строки символов, в другом — в виде трех целых.

```
#include <iostream>
#include <cstdio> // заголовок для функции sscanf()
using namespace std;

class date {
    int day, month, year;
public:
    date(char *str);
    date (int m, int d, int y) {
        day = d;
        month = m;
        year = y;
    }
    void show() {
        cout << month << '/' << day << '/';
```

```
        cout << year << '\n';
    }
};

date::date(char *str)
{
    sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

int main()
{
    // использование конструктора для даты в виде строки
    date sdate("11/1/92");

    // использование конструктора для даты в виде трех целых
    date idate(11, 1, 92);

    sdate.show();
    idate.show();

    return 0;
}
```

Преимущество перегрузки конструктора **date()**, как показано в программе, в том, что вы можете выбрать ту версию инициализации, которая лучше всего подходит к текущей ситуации. Например, если объект типа **date** создается в результате пользовательского ввода, то проще использовать строковую версию. Однако если объект типа **date** строится путем каких-то простых внутренних расчетов, то версия с тремя целыми параметрами становится, вероятно, более привлекательной.

Хотя конструктор можно перегружать любое количество раз, лучше этим не злоупотреблять. С точки зрения стилистики, конструктор имеет смысл перегружать только тогда, когда такая перегрузка позволяет адаптировать программу к часто встречающимся ситуациям. Например, еще одна перегрузка конструктора **date()** для ввода трех восьмеричных целых вряд ли будет иметь какой-то смысл. Однако перегрузка конструктора **date()** для доступа к объекту типа **time_t** (тип данных для хранения системных даты и времени) могла бы оказаться весьма полезной. (См. упражнения для проверки усвоения материала данной главы, где приведен именно такой пример.)

4. Другая ситуация, в которой вам потребуется перегрузить конструктор класса, возникает при выделении динамической памяти массиву объектов этого класса. Как вы должны были узнать из предыдущей главы, динамический массив не может быть инициализирован. Поэтому, если в классе есть инициализирующий конструктор, вам необходимо включить туда и его перегруженную версию без инициализации. Например, ниже приведена программа, в которой массиву объектов динамически выделяется память:

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

```
class myclass {
    int x;
public:
    // перегрузка конструктора двумя способами
    myclass () { x = 0; } // нет инициализации
    myclass (int n) { x = n; } // инициализация
    int getx() { return x; }
    void setx(int n) { x = n; }
};

int main ()
{
    myclass *p;
    myclass ob(10); // инициализация отдельной переменной

    p = new myclass [10]; // здесь инициализировать нельзя
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    int i;

    // инициализация всех элементов значением ob
    for(i=0; i<10; i++) p[i] = ob;

    for(i=0; i<10; i++) {
        cout << "p[ " << i << "]:" << p[i].getx();
        cout << '\n';
    }

    return 0;
}
```

Без перегруженной версии конструктора **myclass()**, в которой отсутствует инициализация, оператор **new** при компиляции вызвал бы ошибку.

Упражнения

1. Дано неполное определение класса:

```
class strtype {
    char *p;
    int len;
public:
    char *getstring() { return p; }
    int getlength() { return len; }
};
```

Добавьте в это определение два конструктора. В первом не должно быть параметров. Он должен выделять 255 байтов памяти (с помощью оператора **new),** инициализировать эту память нулевой строкой и устанавливать переменную **len** равной 255. Во втором конструкторе должно быть два параметра. Первый — это строка, используемая при инициализации, второй — число выделяемых байтов. Во второй версии конструктора должно выделяться заданное количество памяти, в которую должна помещаться копия строки. Необходимо реализовать полный контроль границ массива и, разработав короткую программу вывода, показать, что оба конструктора работают так, как это было задумано.

2. В главе 2, раздел 2.1, упражнение 2 вы создали эмулятор секундомера. Модифицируйте ваше решение так, чтобы в классе **stopwatch** был и конструктор без параметров (как это уже сделано) и его перегруженная версия для доступа к системному времени через стандартную функцию **clock()** Покажите, что внесенные изменения работают.
3. Подумайте о том, каким образом перегруженный конструктор может быть полезен для ваших собственных программных задач.

5.2. Создание и использование конструкторов копий

Одной из важнейших форм перегруженного конструктора является *конструктор копий (copy constructor)*. Как показано в многочисленных примерах из предыдущих глав, передача объектов функциям и их возвращение из функций могут привести к разного рода проблемам. В этом разделе вы узнаете, что одним из способов обойти эти проблемы является определение конструктора копий.

Для начала давайте обозначим проблемы, для решения которых предназначен конструктор копий. Когда объект передается в функцию, делается поразрядная (т. е. точная) копия этого объекта и передается тому параметру функции, который получает объект. Однако бывают ситуации, в которых такая точная копия объекта нежелательна. Например, если объект содержит указатель на выделенную область памяти, то в копии указатель будет ссылаться на *ту же самую* область памяти, на которую ссылается исходный указатель. Следовательно, если копия меняет содержимое области памяти, то эти изменения коснутся также и исходного объекта! Кроме того, когда выполнение функции завершается, копия удаляется, что приводит к вызову деструктора этой копии. Вызов деструктора может привести к нежелательным побочным эффектам, которые в дальнейшем повлияют на исходный объект.

Сходная ситуация имеет место, когда объект является возвращаемым значением функции. Как правило, компилятор генерирует временный объект для хранения возвращаемого функцией значения. (Это происходит автоматиче-

ски и незаметно для вас.) Как только значение возвращается в вызывающую процедуру, временный объект выходит из области видимости, что приводит к вызову деструктора временного объекта. Однако если деструктор удаляет что-то необходимое в вызывающей процедуре (например, если он освобождает динамически выделенную область памяти), то это также приводит к проблемам.

В основе этих проблем лежит факт создания поразрядной копии объекта. Для решения задачи вам, как программисту, необходимо предварительно определить все то, что будет происходить при образовании копии объекта, и таким образом избежать неожиданных побочных эффектов. Способом добиться этого является создание конструктора копий. Путем определения такого конструктора вы можете полностью контролировать весь процесс образования копии объекта.

Важно понимать, что в C++ точно разделяются два типа ситуаций, в которых значение одного объекта передается другому. Первая ситуация — это присваивание. Вторая — инициализация, которая может иметь место В трех случаях:

- Когда в инструкции объявления объекта один объект используется для инициализации другого
- Когда объект передается в функцию в качестве параметра
- Когда в качестве возвращаемого значения функции создается временный объект

Конструктор копий употребляется только для инициализации, но не для присваивания.

По умолчанию при инициализации компилятор автоматически генерирует код, осуществляющий поразрядное копирование. (То есть C++ автоматически создает конструктор копий по умолчанию, который просто дублирует инициализируемый объект.) Однако путем определения конструктора копий вполне возможно предварительно задать то, как один объект будет инициализировать другой. После того как конструктор копий определен, он вызывается всегда при инициализации одного объекта другим.

Конструкторы копий никак не влияют на операции присваивания.

Ниже показана основная форма конструктора копий:

```
имя_класса (const имя_класса fiobj) {
    // тело конструктора
}
```

Здесь ***obj*** — это ссылка на объект, предназначенный для инициализации другого объекта. Например, пусть имеется класс myclass, а **y** — это объект типа myclass, тогда следующие инструкции могли бы вызвать конструктор копий **myclass**:

```
myclass x=y;  // y явно инициализирует x
func1(y);     // y передается в качестве параметра
y=func2();    // y получает возвращаемый объект
```

В двух первых случаях конструктору копий можно бьшо бы передать ссылку на объект y. В последнем случае конструктору копий передается ссылка на объект, возвращаемый функцией **func2()**.

Примеры

1. В данном примере показано, почему необходимо явное определение конструктора копий. В этой программе создается простейший "безопасный" массив целых, в котором предотвращена возможность нарушения границ массива. Память для массива выделяется с помощью оператора new, а указатель на эту память поддерживается внутри каждого объекта-массива.

```
/* В этой программе создается класс "безопасный" массив. Поскольку
память для массива выделяется динамически, то, когда один массив
используется для инициализации другого, для выделения памяти
создается конструктор копий
*/
ttinclude<iostream>
#include <cstdlib>
using namespace std;

class array {
    int *p;
    int size;
public:
    array (int sz) {        // конструктор
    p=new int[sz];
    if(!p) exit(1);
    size=sz;
    cout << "Использование обычного конструктора\n";
    }
    ~ array() {delete [] p; }

    // конструктор копий
    array(const array &a);

    void put(int i, int j) {
        if(i>=0 && i<size) p[i]=j;
    }
```

```
    int get(int i) {
        return p[i];
    }
};

/* Конструктор копий
Память выделяется специально для копии, и адрес этой памяти
передается в указатель p. Следовательно, указатель p больше не
ссылается на ту же самую, где находится исходный объект, динамически
выделенную область памяти:
*/
array::array(const array &a) {
    int i;
    size = a.size;

    p=new int[a.size]; // выделение памяти для копии
    if(!p) exit(1);
    for(i=0; i<a.size; i++) p[i]=a.p[i]; // копирование содержимого
    cout << "Использование конструктора копий\n";
}

int main()
{
  ' array num(10); // вызов обычного конструктора
    int i;
    // помещение в массив нескольких значений
    for(i=0; i<10; i++) num.put(i, i);

    // вывод на экран массива пшп
    for(i=9; i>=0; i--) cout << num.get(i);
    cout << "\n";

    // создание другого массива и инициализация его массивом num
    array x=num; // вызов конструктора копий

    // вывод на экран массива x
    for(i=0; i<10; i++) cout << x.get(i);

    return 0;
}
```

Когда массив **пшп** используется для инициализации массива x, вызывается конструктор копий и для нового массива по адресу **x.p** выделяется память, а содержимое массива **пшп** копируется в массив x. В этом случае в массивах x и пшп находятся одинаковые значения, но при этом — это совершенно различные массивы. (Другими словами, указатели **x.p** и **num.p** теперь не ссылаются на одну и ту же область памяти.) Если бы не был создан конструктор копий, то поразрядная инициализация при выполнении инструкции **array x= пшп** привела бы к тому, что массивы x и пшп оказались бы в одной и той же области памяти! (То есть, указатели **x.p** и **num.p** ссылались бы на одну и ту же область памяти.)

Конструктор копий вызывается только для инициализации. Например, следующая последовательность инструкций не ведет к вызову определенного в предыдущей программе конструктора копий:

```
array a(10);
array b(10);

b = a; // конструктор копий не вызывается
```

В данном случае инструкция b = a представляет собой операцию присваивания.

2. Чтобы понять, как конструктор копий помогает предотвратить некоторые проблемы, связанные с передачей функциям объектов определенных типов, рассмотрим следующую, неправильную программу:

```
// В этой программе имеется ошибка
#include <iostream>
#include <cstring>
ftinclude <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() {delete [] p; }
    char *get() {return p; }
};

strtype:: strtype(char *s)
{
    int l;

    l=strlen(s)+1;

    p=new char[l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s=x.get();
    cout << s << "\n";
}
```

```
int main()
{
strtype a("Hello"), b("There");

show(a);
show(b);

return 0;
}
```

В этой программе, когда объект типа **strtype** передается в функцию **show()**, создается поразрядная копия объекта (поскольку не был определен конструктор копий) и передается параметру x. Таким образом, когда функция возвращает свое значение, x выходит из области видимости и удаляется. Это, естественно, приводит к вызову деструктора объекта x, который освобождает область памяти по адресу x.p. Однако освобожденная память — это та самая память, которую продолжает занимать объект, используемый при вызове функции. Это приводит к ошибке.

Решение предыдущей проблемы лежит в определении конструктора копий для класса **strtype,** который при создании копии объекта типа **strtype** выделяет для нее память. Такой подход используется в следующей, исправленной версии программы:

```
/* В этой программе используется конструктор копирования, что
позволяет передавать функции объекты типа strtype
*/
ttinclude<iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);        // конструктор
    strtype(const strtype &o);    // конструктор копий
    ~strtype() {delete [] p;}       // деструктор
    char *get() {return p;}
};

// Обычный конструктор
strtype::strtype(char *s)
{
    int l;

    l=strlen(s)+1;

    p=new char[l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
```

```
        exit(1);
    }

    strcpy(p, s);
}

// Конструктор копий
strtype::strtype(const strtype &o)
{
    int l;

    l=strlen(o.p)+1;

    p=new char[l]; // выделение памяти для новой копии
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    strcpy(p, o.p); // копирование строки в копию
}

void show(strtype x)
{
    char *s;

    s=x.get();
    cout << s << "\n";
}

int main()
{
strtype a("Hello"), b("There");

show(a);
show(b);

return 0;
}
```

Теперь, когда функция show() завершается и объект x выходит из области видимости, память, на которую ссылается указатель **x.p** (освобождаемая память), — это уже не та память, которая используется переданным в функцию объектом.

1. Конструктор копий вызывается и в тех случаях, когда функция генерирует временный объект, используемый в качестве ее возвращаемого значения (для тех функций, которые возвращают объекты). Зная это, рассмотрим следующий результат работы программы:

```
Работа обычного конструктора
Работа обычного конструктора
Работа конструктора копий
```

Эти строки появились в результате работы следующей программы. Объясните, что именно там происходит и почему.

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

class myclass {
public:
    myclass();
    myclass(const myclass &o);
    myclass f();
};

// Обычный конструктор
myclass::myclass()
{
    cout << "Работа обычного конструктора\n";
}

// Конструктор копий
myclass::myclass(const myclass &o)
{
    cout << "Работа конструктора копий\n";
}

// Возвращение объекта
myclass myclass::f()
{
    myclass temp;

    return temp;
}

int main()
{
    myclass obj;

    obj=obj.f();

    return 0;
}
```

2. Объясните, что в следующей программе неправильно, и исправьте ее.

```
// В этой программе имеется ошибка
#include <iostream>
```

```
ttinclude <cstdlib>
using namespace std;

class myclass {
    int *p;
public:
    myclass(int i);
    ~myclass() {delete p;}
    friend int getval(myclass o);
};

myclass::myclass(int i)
{
    p=new int;
    if(!p) {
    cout << "Ошибка выделения памяти\n";
    exit(1);
    }
    *p=i;
}

int getval(myclass o)
{
    return *o.p; // получение значения
}

int main()
{
    myclass a(1), b(2);

    cout << getval(a) << " " << getval(b);
    cout << "\n";
    cout << getval(a) << " " << getval(b);

    return 0;
}
```

3. Объясните своими словами, зачем нужен конструктор копий и чем он отличается от обычного конструктора.

5.3. Устаревшее ключевое слово *overload*

В ранних версиях C++ для создания перегружаемых функций требовалось ключевое слово **overload.** Хотя ключевое слово **overload** в современных компиляторах больше не поддерживается, вы все еще можете встретить его в существующих программах, поэтому полезно знать, как оно использовалось.

Ниже показана основная форма ключевого слова **overload:**

overload *имя_функции;*

Здесь ***имя_функции*** — это имя перегружаемой функции. Перед этой инструкцией должно находиться объявление перегружаемой функции. Например, следующая инструкция сообщает компилятору, что вы будете перегружать функцию **timer()**:

```
overload timer();
```

*Ключевое слово **overload** является устаревшим и в современных компиляторах C++ не поддерживается.*

5.4. Аргументы по умолчанию

В синтаксисе C++ имеется элемент, который имеет непосредственное отношение к перегрузке функций и называется *аргументом по умолчанию (default argument)*. Аргумент по умолчанию позволяет вам, если при вызове функции соответствующий аргумент не задан, присвоить параметру значение по умолчанию. Как вы увидите далее, применение аргумента по умолчанию является скрытой формой перегрузки функций.

Чтобы передать параметру аргумент по умолчанию, нужно в инструкции определения функции приравнять параметр тому значению, которое вы хотите передать, когда при вызове функции соответствующий аргумент не будет указан. Например, в представленной ниже функции двум параметрам по умолчанию присваивается значение 0:

```
void f(int a = 0, int b = 0);
```

Обратите внимание, что данный синтаксис напоминает инициализацию переменных. Теперь эту функцию можно вызвать тремя различными способами. Во-первых, она может вызываться с двумя заданными аргументами. Во-вторых, она может вызываться только с первым заданным аргументом. В этом случае параметр b по умолчанию станет равным нулю. Наконец, функция **f()** может вызываться вообще без аргументов, при этом параметры a и b по умолчанию станут равными нулю. Таким образом, все следующие вызовы функции **f()** правильны:

```
f(); // a и b по умолчанию равны 0
f(10); // a равно 10, b по умолчанию равно 0
f(10, 99); // a равно 10, b равно 99
```

Из этого примера должно быть ясно, что невозможно передать по умолчанию значение а и при этом задать Ь.

Когда вы создаете функцию, имеющую один или более передаваемых по умолчанию аргументов, эти аргументы должны задаваться только один раз: либо в прототипе функции, либо в ее определении, если определение предшествует первому использованию функции. Аргументы по умолчанию нельзя задавать одновременно в определении и в прототипе функции. Это правило остается в силе, даже если вы просто дублируете одни и те же аргументы по умолчанию.

Как вы, вероятно, догадываетесь, все параметры, задаваемые по умолчанию, должны указываться правее параметров, передаваемых обычным путем. Больше того, после того как вы начали определять параметры по умолчанию, параметры, которые по умолчанию не передаются, уже определять нельзя.

Еще несколько слов об аргументах по умолчанию: они должны быть константами или глобальными переменными. Они не могут быть локальными переменными или другими параметрами.

1. Программа для иллюстрации вышеописанной функции:

```
// Первый простой пример аргументов по умолчанию
tinclude <iostream>
using namespace std;

void f(int a = 0, int b = 0)
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}

int main()
{
    f();
    f(10);
    f(10, 99);

    return 0;
}
```

Как и следовало ожидать, на экран выводится следующее:

```
a: O, b: O
a: 10, b: O
a: 10, b: 99
```

Запомните, если первый аргумент задан по умолчанию, все последующие параметры должны также задаваться по умолчанию. Например, такое небольшое изменение функции **f()** приведет к ошибке при компиляции программы:

```
void f(int a = 0, int Ь) // Неправильно! Параметр b тоже должен
                         // задаваться по умолчанию
{
    cout << "a: " << a << ", Ь: " << b;
    cout << '\n';
}
```

2. Чтобы понять, какое отношение аргументы по умолчанию имеют к перегрузке функций, рассмотрим следующую программу, в которой перегружается функция **rect_area()**. Эта функция возвращает площадь прямоугольника.

```
/* Расчет площади прямоугольника с использованием перегрузки функций
*/
ttinclude<iostream>
using namespace std;

// Возвращает площадь неравностороннего прямоугольника
double rect_area(double length, double width)
{
    return length * width;
}

// Возвращает площадь квадрата
double rect_area(double length)
{
    return length * length;
}

int main()
{
    cout << "площадь прямоугольника 10 x 5.8 равна: ";
    cout << rect_area(10.0, 5.8) << '\n';

    cout << "площадь квадрата 10 x 10 равна: ";
    cout << rect_area(10.0) << '\n';

    return 0;
}
```

В этой программе функция **rect_area()** перегружается двумя способами. В первом — функции передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая версия функции **rect_area()**.

Если исследовать этот пример, то становится ясно, что на самом деле в такой ситуации нет необходимости в двух функциях. Вместо этого второму параметру можно по умолчанию передать некоторое значение, действующее как флаг для функции **rect_area()**. Когда функция встретит это значение, она дважды использует параметр **length**. Пример такого подхода:

```
/* Расчет площади прямоугольника с передачей аргументов по умолчанию
*/
#include <iostream>
using namespace std;

// Возвращает площадь прямоугольника
double rect_area(double length, double width = 0)
{
    if(!width) width = length;
    return length * width;
}

int main()
{
    cout << "площадь прямоугольника 10 x 5.8 равна: ";
    cout << rect_area(10.0, 5.8) << '\n';

    cout << "площадь квадрата 10 x 10 равна: ";
    cout << rect_area(10.0) << '\n';

    return 0;
}
```

Теперь параметру **width** по умолчанию присваивается нуль. Такое значение выбрано потому, что не бывает прямоугольника с нулевой стороной. (Фактически, Прямоугольник с нулевой стороной — это линия.) Таким образом, когда в **rect_area()** встречается такое, переданное по умолчанию значение, для ширины прямоугольника автоматически используется параметр **length.**

Как показано в этом примере, аргументы по умолчанию часто обеспечивают простую альтернативу перегрузке функций. (Конечно, имеется масса ситуаций, в которых перегрузка функций необходима по-прежнему.)

3. Передавать конструкторам аргументы по умолчанию не только правильно, но и вполне обычно. Как отмечалось ранее в этой главе, часто конструктор перегружается просто для того, чтобы могли создаваться как инициализируемые, так и неинициализируемые объекты. Во многих случаях можно избежать перегрузки конструктора путем передачи ему одного или более аргументов по умолчанию. Например, рассмотрим следующую программу:

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

```
class myclass {
    int x;
public:
/* Использование аргумента по умолчанию вместо перегрузки
конструктора
*/
    myclass(int n = 0) { x = n; }
    int getx() { return x; }
};

int main()
{
    myclass o1(10); // объявление с начальным значением
    myclass o2; // объявление без начального значения

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}
```

Как показано в этом примере, путем передачи по умолчанию параметру n нулевого значения, можно создавать не только объекты, имеющие явно заданные начальные значения, но и такие, для которых достаточно значений, задаваемых по умолчанию.

4. Другим хорошим применением аргумента по умолчанию является случай, когда с помощью такого параметра происходит выбор нужного варианта развития событий. Можно передать параметру значение по умолчанию так, чтобы использовать его в качестве флага, сообщающего функции о необходимости продолжить работу в обычном режиме. Например, в следующей программе функция **print()** выводит строку на экран. Если параметр **how** равен значению **ignore,** текст выводится в том виде, в каком он задан. Если параметр **how** равен значению **upper,** текст выводится в верхнем регистре. Если параметр how равен значению **lower,** текст выводится в нижнем регистре. Если параметр **how** не задан, его значение по умолчанию равно —1, что говорит функции о необходимости повторно использовать его предыдущее значение.

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

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

int main()
{
    print("Привет\n", ignore);
```

```
    print("Привет \n", upper);
    print("Привет \n"); // продолжение вывода в верхнем регистре
    print("Привет \n", lower);
    print("Это конец \n"); // продолжение вывода
                          // в нижнем регистре

    return 0;
}

/* Печать строки в заданном регистре. Использование заданного
последним регистра, если он не задан.
*/
void print(char *s, int how)
{
    static int oldcase = ignore;

    // повторять работу с прежним регистром, если новый не задан
    if(how<0) how = oldcase;
    while (*s) {
        switch(how) {
            case upper: cout << (char) toupper(*s);
                break;
            case lower: cout << (char) tolower(*s);
                break;
            default: cout << *s;
        }
        s++;
    }
    oldcase = how;
}
```

Эта программа выводит следующее:

```
Привет
ПРИВЕТ
ПРИВЕТ
привет
это конец
```

5. Ранее в этой главе мы рассматривали общую форму конструктора копий. В этой общей форме имелся только один параметр. Однако вполне возможно создавать конструкторы копий, получающие дополнительные аргументы, если только это аргументы по умолчанию. Например, вполне приемлема следующая форма конструктора копий:

```
myclass(const myclass &obj, int x=0) {
    // тело конструктора
}
```

Поскольку первый аргумент является ссылкой на копируемый объект, а все остальные — это аргументы по умолчанию, эту функцию можно квалифицировать как конструктор копий. Такая гибкость позволяет создавать самые разнообразные конструкторы копий.

6. Хотя аргументы по умолчанию являются мощным и удобным инструментом, ими нельзя злоупотреблять. Несомненно, что при правильном применении аргументы по умолчанию позволяют функции выполнять свою работу эффективным и простым по реализации образом. Однако так происходит лишь тогда, когда переданное по умолчанию значение имеет смысл. Например, если аргумент, используемый в девять или десять раз чаще других, передать функции по умолчанию, то, очевидно, это неплохо. Однако в случае, если нет значения, используемого чаще других, или нет выгоды от аргумента по умолчанию в качестве флага, то нет большого смысла передавать что-либо по умолчанию. Фактически, обеспечение передачи аргумента по умолчанию, когда это не вызвано необходимостью, ограничивает возможности вашей программы и вводит в заблуждение всех пользователей такой функции.

Как и при перегрузке функций, хороший программист в каждом конкретном случае всегда сумеет определить, стоит или нет пользоваться аргументом по умолчанию.

Упражнения

1. В стандартной библиотеке C++ существует функция **strtol()**, имеющая следующий прототип:

```
long strtol(const char *start, const **end, int base);
```

Функция преобразует обозначающую число строку, на которую ссылается указатель ***start,*** в длинное целое. Число ***base*** задает основание системы счисления этого числа. При возвращении функцией своего значения указатель ***end*** ссылается на символ в строке, следующий сразу за последней цифрой строки. Возвращаемое длинное целое эквивалентно тому числу, которое записано в строке. Диапазон значений ***base*** от 2 до 38. Однако наиболее часто основание системы счисления равно 10.

Создайте функцию **mystrtol()**, работающую точно так же, как и функция **strtol(),** но аргумент 10 должен передаваться параметру ***base*** по умолчанию. (Свободно пользуйтесь функцией **strtol()** для фактического преобразования. Для этого в программу требуется включить заголовок **<cstdlib>.)** Покажите, что ваша версия работает правильно.

2. Что неправильно в следующем прототипе функции?

```
char *f(char *p, int x = 0, char *q);
```

3. В большинстве компиляторов C++ применяются нестандартные функции, управляющие позиционированием курсора и другими аналогичными деист-

виями. Если в вашем компиляторе применяются такие функции, создайте функцию **myclreol()**, которая стирает строку, начиная от текущей позиции курсора до конца строки. Передайте этой функции параметр, задающий число стираемых позиций. Если параметр не задавать, то по умолчанию должна стираться вся строка. В противном случае должно стираться число символьных позиций, заданное параметром.

4. Что неправильно в следующем прототипе функции с аргументом по умолчанию?

```
int f(int count, int max = count);
```

5.5. Перегрузка и неоднозначность

При перегрузке возможно внесение неоднозначности в программу. *Неоднозначность (ambiguity),* вызванная перегрузкой функций, может быть введена в программу при преобразовании типа, а также при использовании параметров-ссылок и аргументов по умолчанию. Некоторые виды неоднозначности вызываются самой перегрузкой функций. Другие виды связаны со способом вызова перегруженных функций. Чтобы программа компилировалась без ошибок, от неоднозначности необходимо избавиться.

Примеры

1. Один из наиболее частых видов неоднозначности вызывается правилами преобразования типа в C++. Как вы знаете, при вызове функции с аргументом, тип которого совместим (но не аналогичен) с типом параметра, которому он передается, тип аргумента по умолчанию преобразуется в тип параметра. Об этой операции иногда говорят как о *приведении типа (type promotion).* Приведение типа — это такой вид преобразования типа, который позволяет некоторым функциям, например **putchar()**, вызываться с символьным параметром, даже тогда, когда аргумент функции имеет тип **int.** Однако в некоторых случаях это преобразование типа при перегрузке функций вызовет ситуацию неоднозначности. Чтобы понять, как это происходит, исследуем следующую программу:

```
// Эта программа содержит ошибку неоднозначности
#include <iostream>
using namespace std;

float f(float i)
{
    return i / 2.0;
}
```

```
double f(double i)
{
    return i / 3.0;
}
int main()
{
    float x = 10.09;
    double y = 10.09;
    cout << f(x); // нет неоднозначности
                  // используется функция f(float)
    cout << f(y); // нет неоднозначности
                  // используется функция f(double)
    cout << f(10); // неоднозначность
                   // куда преобразовать 10?
                   // в значение типа double или float?
    return 0;
}
```

Как указано в комментариях к функции **main(),** компилятор в состоянии выбрать правильную версию функции **f(),** если она вызывается либо с переменными типа **double,** либо с переменными типа **float.** Однако что случается, если она вызывается с целым? Какую функцию вызовет компилятор **f(float)** или **f(double)**? (Оба преобразования правильны!) И в том, и в другом случае правильно "привести" тип **int** либо к типу **float,** либо к типу **double.** Таким образом, возникает ситуация неоднозначности.

Этот пример выявляет также то, как неоднозначность может проявляться при вызове перегруженных функций. Очевидно, что сама по себе неоднозначность не присуща перегруженным версиям функции **f(),** пока каждая вызывается с аргументом соответствующего типа.

2. Другой пример перегрузки функции, которая сама по себе не должна приводить к неоднозначности. Тем не менее, при вызове с аргументом неправильного типа, правила преобразования типа C++ создают ситуацию неоднозначности.

```
// Эта программа неоднозначна
ttinclude <iostream>
using namespace std;
void f(unsigned char c)
{
    cout << c;
}
void f(char c)
{
    cout << c;
}
```

```
int main()
{
    f('c');
    f(86); // какая версия функции f() вызывается?

    return 0;
}
```

Когда функция f() вызывается с числовой константой 86, компилятор не может понять, какую версию функции вызвать: f(unsigned char) или **f(char)**. Оба преобразования одинаково правильны, что и ведет к неоднозначности.

3. Один из видов неоднозначности проявляется, если вы пытаетесь перегрузить функции, единственным отличием которых является то, что одна использует параметр-ссылку, а другая параметр-значение по умолчанию. В рамках формального синтаксиса C++ у компилятора нет способа узнать, какую функцию вызвать. Запомните, что нет синтаксических отличий между вызовом функции по значению и вызовом функции по ссылке. Например:

```
// Эта программа неоднозначна
#include <iostream>
using namespace std;

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

// здесь внутренняя неоднозначность
int f(int a, int &b)
{
    return a - b;
}

int main()
{
    int x = 1, y = 2;

    cout << f(x, y); // какую версию f() вызвать?

    return 0;
}
```

Здесь вызов функции f(x, y) неоднозначен, поскольку вызвана может быть любая версия функции. При этом компилятор выставит флаг ошибки даже раньше того, как встретится такая инструкция, поскольку сама перегрузка этих двух функций внутренне неоднозначна, и компилятор не будет знать, какую из них предпочесть.

4. Другим видом неоднозначности при перегрузке функций является случай, когда одна или более перегруженных функций используют аргумент по умолчанию. Рассмотрим программу:

```
// Неоднозначность, основанная на аргументах по умолчанию
// и перегрузке функций
#include <iostream>
using namespace std;

int f(int a)
{
    return a * a;
}

int f(int a, int b = 0)
{
    return a * b;
}

int main()
{
    cout << f(10, 2); // вызывается f(int, int)
    cout << f(10); // неоднозначность,
                   // что вызвать f(int, int) или f(int)???

    return 0;
}
```

Здесь вызов функции **f(10,** 2) совершенно правилен и не ведет к неоднозначности. Однако у компилятора нет способа выяснить, какую версию функции **f()** вызывает версия **f(10)** — первую или вторую, в которой параметр b передается по умолчанию.

1. Попытайтесь провести компиляцию всех предыдущих программ, в которых имеет место неоднозначность. Запомните сообщения об ошибках. Это поможет вам сразу распознать ошибки неоднозначности, если они появятся в ваших программах.

5.6. Определение адреса перегруженной функции

В заключение этой главы вы узнаете, как найти адрес перегруженной функции. Так же, как и в C, вы можете присвоить адрес функции указателю и получить доступ к функции через этот указатель. Адрес функции можно найти, если поместить имя функции в правой части инструкции присваивания без всяких скобок или аргументов. Например, если **zap()** — это функ-

ция, причем правильно объявленная, то корректным способом присвоить переменной p адрес функции zap() является инструкция:

```
p = zap;
```

В языке C любой тип указателя может использоваться как указатель на функцию, поскольку имеется только одна функция, на которую он может ссылаться. Однако в C++ ситуация несколько более сложная, поскольку функция может быть перегружена. Таким образом, должен быть некий механизм, который позволял бы определять адреса перегруженных версий функции.

Решение оказывается не только элегантным, но и эффектным. *Способ объявления указателя* и определяет то, адрес какой из перегруженных версий функции будет получен. Уточним, объявления указателей соответствуют объявлениям перегруженных функций. Функция, объявлению которой соответствует объявление указателя, и является искомой функцией.

Примеры

1. Здесь представлена программа, которая содержит две версии функции **space()**. Первая версия выводит на экран некоторое число пробелов, заданное в переменной **count.** Вторая версия выводит на экран некоторое число каких-то иных символов, вид которых задан в переменной ch. В функции **main()** объявляются оба указателя на эти функции. Первый задается как указатель на функцию, имеющую только один целый параметр. Второй объявляется как указатель на функцию, имеющую два параметра.

```
/* Иллюстрация присваивания и получения указателей на перегруженные
функции
*/
#include <iostream>
using namespace std;

// вывод заданного в переменной count числа пробелов
void space(int count)
{
    for( ; count; count--) cout << ' ';
}

// вывод заданного в переменной count числа символов,
// вид которых задан в переменной ch
void space(int count, char ch)
{
    for(; count; count --) cout << ch;
}
```

```
int main()
{
    /* Создание указателя на функцию с одним целым параметром. */
    void (*fp1)(int);

    /* Создание указателя на функцию с одним целым и одним
символьным параметром. */
    void (*fp2)(int, char);

    fp1 = space; // получение адреса функции space(int)

    fp2 = space; // получение адреса функции space(int, char)

    fp1(22); // выводит 22 пробела
    cout <<" | \n";

    fp2(30, 'x'); // выводит 30 символов x
    cout <<" | \n";

    return 0;
}
```

Как показано в комментариях, на основе того, каким образом объявляются указатели **flpl** и **fp2,** компилятор способен определить, какой из них на какую из перегруженных функций будет ссылаться.

Повторим, если вы присваиваете адрес перегруженной функции указателю на функцию, то объявление указателя определяет, адрес какой именно функции ему присваивается. Более того, объявление указателя на функцию должно точно соответствовать одной и только одной перегруженной функции. Если это не так, будет внесена неоднозначность, что приведет к ошибке при компиляции программы.

Упражнения

1. Ниже приведены две перегруженные функции. Покажите, как получить адрес каждой из них.

```
int dif(int a, int b)
{
    return a — b;
}

float dif(float a, float b)
{
    return a — b;
}
```

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы:

1. Перегрузите конструктор **date()** из раздела 5.1, пример 3 так, чтобы он имел параметр типа **time_t.** (Вспомните, что **time_t** — это тип данных, определенный стандартными библиотечными функциями времени и даты компилятора C++.)
2. Что неправильно в следующем фрагменте?

```
class samp {
    int a;
public:
    samp(int i) { a = i; }
    // ...
};
// ...
int main()
{
    samp x, y(10);
    // ...
}
```

3. Приведите два довода в пользу того, почему вам могло бы потребоваться перегрузить конструктор класса.
4. Какова основная форма конструктора копий?
5. Какой тип операций ведет к вызову конструктора копий?
6. Кратко объясните, зачем нужно ключевое слово **overload,** и почему оно больше не употребляется.
7. Объясните, что такое аргумент по умолчанию?
8. Создайте функцию **reverse()** с двумя параметрами. Первый параметр str — это указатель на строку, порядок следования символов в которой, после возвращения функцией своего значения, должен быть заменен на обратный. Второй параметр **count** задает количество переставляемых в строке **str** символов. Значение **count** по умолчанию должно быть таким, чтобы в случае его задания функция **reverse()** меняла порядок следования символов в целой строке.
9. Что неправильно в следующем прототипе функции?

```
char *wordwrap(char *str, int size = 0, char ch);
```

10. Приведите несколько причин появления неоднозначности при перегрузке функций.

11. Что неправильно в следующем фрагменте?

```
void compute(double *num, int divisor = 1);
void compute(double *num);
// ...
compute(&x);
```

12. При присваивании указателю адреса перегруженной функции, что определяет конкретную версию используемой функции?

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Создайте функцию **order(),** которая получает два параметра-ссылки на целые. Если первый аргумент больше второго, поменяйте их значения. В противном случае ничего делать не надо. Таким образом, порядок следования двух аргументов, используемых при вызове функции **order(),** должен быть таким, чтобы всегда после возвращения функцией своего значения первый аргумент был меньше второго. Например, если дано

```
int x = 1, y = 0;
order(x, y);
```

   то после вызова функции x будет равен 0, а y будет равен 1.

2. Почему следующие две перегруженные функции внутренне неоднозначны?

```
int f(int a);
int f(int &a);
```

3. Объясните, почему использование аргумента по умолчанию связано с перегрузкой функций.

4. Пусть дано следующее неполное описание класса, добавьте конструкторы так, чтобы оба объявления в функции **main()** были правильны. (Подсказка: вам необходимо дважды перегрузить конструктор **samp()**.)

```
class samp {
    int a;
```

```
public:
    // добавьте конструкторы
    int get_a() { return a; }
};

int main()
{
    samp ob(88); // инициализация объекта a значением 88
    samp obarray[10]; // неинициализируемый 10-элементный массив

    // ...
}
```

5. Кратко объясните, зачем нужны конструкторы копий.

Глава 6

Введение в перегрузку операторов

В этой главе рассматривается очередное важное свойство C++: перегрузка операторов. Это свойство позволяет определять значение операторов C++ относительно задаваемых вами классов. Путем перегрузки связанных с классами операторов можно легко добавлять в программу новые типы данных.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Покажите, как перегрузить конструктор для следующего класса так, чтобы можно было создавать не только инициализируемые, но и неинициализируемые объекты. (При создании таких объектов присвойте переменным x и y значение 0.)

```
class myclass {
    int x,y;
public:
    myclass(int i, int j) { x = i; y = j; }
    // ...
};
```

2. Используя класс из вопроса 1, покажите, как с помощью аргументов по умолчанию можно избежать перегрузки конструктора **myclass()**.
3. Что неправильно в следующем объявлении?

```
int f(int a = 0, double balance);
```

4. Что неправильно в следующих двух перегруженных функциях?

```
void f(int a);
void f(int &a);
```

5. Когда удобнее использовать аргументы по умолчанию? Когда этого лучше не делать?

6. Дано следующее определение класса. Возможно ли динамически выделить память для массива объектов такого типа?

```
class test {                                          1
    char *p;
    int *q;
    int count;
public:
    test(char *x, int *y, int c) {
        p = x;
        q = y;
        count = c;
    }
    // ...
};
```

7. Что такое конструктор копий и при каких условиях он вызывается?

6.1. Основы перегрузки операторов

Перегрузка операторов напоминает перегрузку функций. Более того, перегрузка операторов является фактически одним из видов перегрузки функций. Однако при этом вводятся некоторые дополнительные правила. Например, оператор всегда перегружается относительно определенного пользователем типа данных, такого, как класс. Другие отличия будут обсуждаться ниже по мере необходимости.

Когда оператор перегружается, то ничего из его исходного значения не теряется. Наоборот, он приобретает дополнительное значение, связанное с классом, для которого оператор был определен.

Для перегрузки оператора создается *оператор-функция (operator function)*. Чаще всего, оператор-функция является членом класса или дружественной классу, для которого она определена. Однако есть небольшая разница между оператор-функцией — членом класса и дружественной оператор-функцией. В первой части этой главы обсуждается создание оператор-функций — членов класса. О дружественных оператор-функциях будет рассказано далее в этой главе.

Здесь представлена основная форма оператор-функции — члена класса:

```
возвращаемый_тип имя_класса::operator#(список_аргументов)
{
    // выполняемая операция
}
```

Часто типом возвращаемого значения оператор-функции является класс, для которого она определена. (Хотя оператор-функция может возвращать данные любого типа.) В представленной общей форме оператор-функции вме-

сто знака # нужно подставить перегружаемый оператор. Например, если перегружается оператор +, то у функции должно быть имя **operator+**. Содержание списка ***список-аргументов*** зависит от реализации оператор-функции и от типа перегружаемого оператора.

Следует запомнить два важных ограничения на перегрузку операторов. Во-первых, нельзя менять приоритет операторов. Во-вторых, нельзя менять число операндов оператора. Например, нельзя перегрузить оператор / так, чтобы в нем использовался только один операнд.

Большинство операторов C++ можно перегружать. Ниже представлены те несколько операторов, которые перегружать нельзя:

```
.  ::  .*  ?
```

Кроме того, нельзя перегружать операторы препроцессора. (Оператор .* является сугубо специальным и в книге не рассматривается.)

Запомните, что в C++ понятие оператора трактуется очень широко: в это понятие входят оператор индексирования **[],** оператор вызова функции (), операторы **new** и **delete,** операторы . (точка) и -> (стрелка). Однако в данной главе мы коснемся более обычных операторов.

Оператор-функции, за исключением оператора =, наследуются производным классом. Тем не менее для производного класса тоже можно перегрузить любой выбранный оператор (включая операторы, уже перегруженные в базовом классе).

Вы уже пользовались двумя перегруженными операторами: << и >>, которые перегружались для реализации ввода/вывода. Как уже упоминалось, перегрузка этих операторов для реализации ввода/вывода не мешает им выполнять свои традиционные функции левого и правого сдвига.

Хотя допустимо иметь оператор-функцию для реализации *любого* действия — связанного или нет с традиционным употреблением оператора — лучше, если действия перегружаемых операторов остаются в сфере их традиционного использования. При создании перегружаемых операторов, для которых этот принцип не поддерживается, имеется риск существенного снижения читабельности программ. Например, перегрузка оператора / так, чтобы 300 раз записать в дисковый файл фразу "Мне нравится C++", является явным злоупотреблением перегрузкой операторов.

Несмотря на вышесказанное, иногда может потребоваться использовать какой-либо оператор нетрадиционным образом. Типичным примером этого как раз и являются перегруженные для ввода/вывода операторы << и >>. Однако даже в этом случае, левые и правые стрелки обеспечивают визуально понятный смысл их значения. Поэтому, даже если вам очень хочется перегрузить какой-нибудь оператор нестандартным способом, лучше приложите дополнительные усилия и постарайтесь воспользоваться каким-нибудь более подходящим оператором.

И последнее, оператор-функции не могут иметь параметров по умолчанию.

6.2. Перегрузка бинарных операторов

Когда оператор-функция — член класса перегружает бинарный оператор, у функции будет только один параметр. Этот параметр получит тот объект, который расположен справа от оператора. Объект слева генерирует вызов оператор-функции и передается неявно, с помощью указателя **this.**

Важно понимать, что для написания оператор-функций имеется множество вариантов. Примеры, показанные здесь и в других местах главы, не являются исчерпывающими, хотя они иллюстрируют несколько наиболее общих технических приемов.

1. В следующей программе перегружается оператор + относительно класса **coord.** Этот класс используется для поддержания координат X,Y.

```
// Перегрузка оператора + относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x,y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int Si, int Sj) { i = x; j = y; }
    coord operator+(coordob2) ;
};

// Перегрузка оператора + относительно класса coord
coord coord: :operator+(coord ob2)
{
    coord temp;
    temp.x = x +ob2 .x;
    temp.y = y +ob2 .y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // сложение двух объектов
                  // вызов функции operator+()
```

```
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
(o1 + o2) X: 15, Y: 13
```

Давайте внимательно рассмотрим программу. Функция **operator+()** возвращает объект типа **coord,** в котором сумма координат по оси X находится в переменной x, а сумма координат по оси Y — в переменной y. Отметьте, что временный объект **temp** используется внутри функции **operator+()** для хранения результата и является возвращаемым объектом. Отметьте также, что ни один из операндов не меняется. Назначение переменной **temp** легко понять. В данной ситуации (как и в большинстве ситуаций) оператор + был перегружен способом, аналогичным своему традиционному арифметическому использованию. Поэтому и было важно, чтобы ни один из операндов не менялся. Например, когда вы складываете 10+4, результат равен 14, но ни 10, ни 4 не меняются. Таким образом, временный объект необходим для хранения результата.

Смысл того, что функция **operator+()** возвращает объект типа **coord,** состоит в том, что это позволяет использовать результат сложения объектов типа **coord** в сложном выражении. Например, инструкция

```
o3 = o1 + o2;
```

правильна только потому, что результат выполнения операции o1 + o2 является объектом, который можно присвоить объекту o3. Если бы возвращаемым значением был объект другого типа, то эта инструкция была бы неправильна. Более того, возвращая объект типа **coord,** оператор сложения допускает возможность существования строки, состоящей из нескольких сложений. Например, следующая инструкция вполне корректна:

```
o3 = o1 + o2 + o1 + o3;
```

Хотя у вас будут ситуации, в которых понадобится оператор-функция, возвращающая нечто иное, чем объект класса, для которого она определена, большинство создаваемых вами оператор-функций будут возвращать именно такие объекты. (Основное исключение из этого правила связано с перегрузкой операторов отношения и логических операторов. Эта ситуация исследуется в разделе 6.3 "Перегрузка операторов отношения и логических операторов" далее в этой главе.)

Последнее замечание по этому примеру. Поскольку объект типа **coord** является возвращаемым значением оператор-функции, то следующая инструкция также совершенно правильна:

```
(o1 + o2).get_xy(x, y);
```

Здесь временный объект, возвращаемый функцией **operator+(),** используется непосредственно. Естественно, что после выполнения этой инструкции временный объект удаляется.

2. В следующей версии предыдущей программы относительно класса **coord** перегружаются операторы — и =.

```
// Перегрузка операторов +, — и = относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j ) { i = x; j = y; }
    coord operator+ (coordob2) ;
    coord operator- (coordob2) ;
    coord operator=(coord ob2);
};

// Перегрузка оператора + относительно класса coord
coord coord: :operator+(coord ob2)
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// Перегрузка оператора — относительно класса coord
coord coord: : operator- (coordob2)
{
    coord temp;

    temp.x = x — ob2.x;
    temp.y = y — ob2.y;

    return temp;
}

// Перегрузка оператора = относительно класса coord
coord coord: :operator=(coord ob2)
{
    x = ob2.x;
    y = ob2.y;
```

```
    return *this; // возвращение объекта,
                  // которому присвоено значение
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // сложение двух объектов
                  // вызов функции operator+()
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // вычитание двух объектов
                  // вызов функции operator-()
    o3.get_xy(x, y);
    cout << " (o1 - o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // присваивание объекта- вызов функции operator=()
    o3.get_xy(x, y);
    cout << "(o3 = o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Реализация функции **operator–()** похожа на реализацию функции **operator+()**. Однако она иллюстрирует ту особенность перегрузки операторов, где важен порядок следования операндов. При создании функции **operator+()** порядок следования операндов значения не имел. (То есть A+B тождественно B+A.) Однако результат операции вычитания зависит от порядка следования операндов. Поэтому, чтобы правильно перегрузить оператор вычитания, необходимо вычесть правый операнд из левого. Поскольку левый операнд генерирует вызов функции **operator–()**, порядок вычитания должен быть следующим:

```
x — ob2.x;
```

При перегрузке бинарного оператора левый операнд передается функции неявно, а правый оператор передается функции в качестве аргумента.

Теперь рассмотрим оператор-функцию присваивания. В первую очередь необходимо отметить, что левый операнд (т. е. объект, которому присваивается значение) после выполнения операции меняется. Здесь сохраняется обычный смысл присваивания. Во-вторых, функция возвращает указатель ***this.** Это

происходит потому, что функция **operator=()** возвращает тот объект, которому присваивается значение. Таким образом удается выстраивать операторы присваивания в цепочки. Как вы уже должны знать, в C++ следующая инструкция синтаксически правильна (и на практике вполне обычна):

```
a = b = c = d = 0;
```

Возвращая указатель ***this,*** перегруженный оператор присваивания дает возможность подобным образом выстраивать объекты типа **coord.** Например, представленная ниже инструкция вполне корректна:

```
o3 = o2 = o1;
```

Запомните, нет правила, требующего, чтобы перегруженная оператор-функция присваивания возвращала объект, полученный в результате присваивания. Однако если вы хотите перегрузить оператор = относительно класса, то, как и в случае присваивания встроенных типов данных, он должен возвращать указатель ***this**.

3. Имеется возможность перегрузить оператор относительно класса так, что правый операнд будет объектом встроенного типа, например, целого, а не объектом того класса, членом которого является оператор-функция. Например, в приведенном ниже примере оператор + перегружается так, что прибавляет целое значение к объекту типа **coord:**

```
// Перегрузка оператора + как для операции ob+ob,
// так и для операции ob+int
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator+(coord ob2); // ob + ob
    coord operator+(int i); // ob + int
};

// Перегрузка оператора + относительно класса coord
coord coord::operator+(coord ob2)
{
    coord temp;

    temp.x = x +ob2.x;
    temp.y = y +ob2.y;

    return temp;
}
```

```
// Перегрузка оператора + для операции ob+int
coord coord::operator+(int i)
{
    coord temp;

    temp.x = x + i;
    temp.y = y + i;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // сложение двух объектов
                  // вызов функции operator+(coord)
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 + 100; // сложение объекта и целого
                   // вызов функции operator+(int)
    o3.get_xy(x, y);
    cout << "(o1 + 100) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Здесь важно запомнить следующее: когда оператор-функция — член класса перегружается так, чтобы объект этого класса можно было использовать в одной операции с переменной встроенного типа данных, встроенный тип данных должен находиться справа от оператора. Смысл этого легко понять: он в том, что объект, который находится слева, генерирует вызов оператор-функции. Однако что произойдет, если компилятор встретит следующую инструкцию?

```
o3 = 19 + o1; // int + ob
```

Для обработки сложения целого с объектом встроенной операции не существует. Перегруженная функция **operator+(int i)** работает только в том случае, если объект находится слева от оператора. Поэтому эта инструкция приведет к ошибке при компиляции. (Позже вы узнаете способ обойти это ограничение.)

4. В оператор-функции можно использовать параметр-ссылку. Например, допустимым способом перегрузки оператора + относительно класса **coord** является следующий:

```
// Перегрузка + относительно класса coord, с использованием ссылки
coord coord::operator+(coord&ob2)
```

```
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}
```

Одним из доводов в пользу использования ссылки в качестве параметра оператор-функции является ее эффективность. Передача объекта функции в качестве параметра часто требует больших затрат процессорного времени. Передача же адреса объекта всегда быстрее и эффективней. Если оператор многократно используется, параметр-ссылка обычно позволяет значительно повысить производительность.

Другой довод в пользу использования параметра-ссылки состоит в том, что ссылка позволяет избежать неприятностей, связанных с удалением копии операнда. Как вы знаете по предыдущим главам, при передаче аргумента по значению создается его копия. Если у такого объекта есть деструктор, то после завершения выполнения функции вызывается деструктор копии. Иногда возможны ситуации, когда деструктор удаляет нечто такое, что необходимо вызывающему объекту. Использование в этом случае в качестве параметра не самого объекта, а ссылки на этот объект — это простой (и эффективный) способ избежать проблем. Тем не менее, запомните, что в общем случае решить эту проблему могло бы определение конструктора копий.

Упражнения

1. Для класса **coord** перегрузите операторы * и /. Продемонстрируйте их работу.
2. В приведенном ниже примере некорректно перегружен оператор %. Почему?

```
coord coord::operator%(coordob)
{
    double i;

    cout << "Введите число: ";
    cin>>i;
    cout << "корень " << i << " равен ";
    cout << sqr(i);
}
```

3 Поэкспериментируйте, меняя тип возвращаемого значения оператор-функций на что-нибудь отличное от **coord.** Обратите внимание на генерируемые компилятором сообщения об ошибках.

6.3. Перегрузка операторов отношения и логических операторов

Существует возможность перегрузки операторов отношения и логических операторов. При перегрузке операторов отношения и логических операторов так, чтобы они вели себя обычным образом, не нужны оператор-функции, возвращающие объект класса, для которого эти оператор-функции определены. Вместо этого они должны возвращать целое, интерпретируемое как значение **true** или **false.** Помимо того, что возвращаемым значением таких оператор-функций должно быть значение **true** или **false,** должна быть возможность встраивания операторов отношения и логических операторов в большие выражения, включающие также данные других типов.

*В современных компиляторах C++ оператор-функции для перегрузки операторов отношения и логических операторов могут возвращать значения булева типа, хотя особого преимущества в этом нет. Как было описано в главе 1, в булевом типе данных определены только два значения — **true и false.** Эти значения автоматически конвертируются в ненулевое и нулевое значения. И наоборот, целые ненулевое и нулевое значения автоматически конвертируются в значения **true и false.***

1. В следующей программе перегружаются операторы **==** и **&&**:

```
// Перегрузка операторов == и && относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord () { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int si, int &j ) { i = x; j = y; }
    int operator==(coord ob2) ;
    int operator&&(coordob2) ;
};

// Перегрузка оператора == для класса coord
int coord: :operator==(coord ob2)
{
    return x==ob2.x && y==ob2.y;
}
```

```
// Перегрузка оператора && для класса coord
int coord::operator&&(coord ob2)
{
    return (x && ob2.x) && (y &S ob2.y);
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3(10, 10), o4 (0, 0);

    if(o1==o2) cout << "o1 равно o2\n";
    else cout << "o1 не равно o2\n";

    if(o1==o3) cout << "o1 равно o3\n";
    else cout << "o1 не равно o3\n";

    if(o1&&o2) cout << "o1 && o2 равно истина\n";
    else cout << "o1 && o2 равно ложь\n";

    if(o1&&o4) cout << "o1 && o4 равно истина\n";
    else cout << "o1 && o4 равно ложь\n";

    return 0;
}
```

Упражнения

1. Относительно класса coord перегрузите операторы отношения < и >.

6.4. Перегрузка унарных операторов

Перегрузка унарных операторов аналогична перефузке бинарных, за исключением того, что мы имеем дело не с двумя, а с одним операндом. При перефузке унарного оператора с использованием функции-члена у функции нет параметров. Поскольку имеется только один операнд, он и генерирует вызов оператор-функции. Другие параметры не нужны.

Примеры

1. В следующей программе относительно класса **coord** перегружается оператор инкремента (++):

```
// Перегрузка оператора ++ относительно класса coord
#include <iostream>
using namespace std;
```

```
class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coordoperator++();
};

// Перегрузка оператора ++ для класса coord
coord coord::operator++()
{
    x++;
    y++;

    return *this;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    ++o1; // инкремент объекта
    o1.get_xy(x, y);
    cout << "(++o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Поскольку оператор инкремента увеличивает свой операнд на единицу, перегрузка этого оператора меняет объект, с которым он работает. Это позволяет использовать оператор инкремента в качестве части более сложной инструкции, например, такой:

```
o2 = ++o1;
```

Как и в случае бинарных операторов, нет правила, которое заставляло бы перегружать унарный оператор с сохранением его обычного смысла. Однако в большинстве случаев лучше поступать именно так.

2. В ранних версиях C++ при перегрузке оператора инкремента или декремента положения операторов ++ и — относительно операнда не различались. Поэтому по отношению к предыдущей программе следующие две инструкции эквивалентны:

```
o1++;
++o1;
```

Однако в современной спецификации C++ определен способ, по которому компилятор может различить эти две инструкции. В соответствии с этим

способом задаются две версии функции **operator++().** Первая определяется так, как было показано в предыдущем примере. Вторая определяется следующим образом:

```
coord coord::operator++(int notused);
```

Если оператор ++ указан перед операндом, вызывается функция **operator++()**. Если оператор '++ указан после операнда, вызывается функция **operator++(int notused).** В этом случае переменной **notused** передается значение 0. Таким образом, если префиксный и постфиксный инкремент или декремент важны для объектов вашего класса, то понадобится реализовать обе оператор-функции.

3. Как вы знаете, знак минус в C++ является как бинарным, так и унарным оператором. Вы, наверное, хотели бы знать, как его можно перегрузить относительно создаваемого вами класса так, чтобы оператор сохранил оба эти качества. Реальное решение достаточно элементарно: просто перегрузите его дважды, один раз как бинарный оператор, а второй — как унарный. Программа, реализующая этот прием, показана ниже:

```
// Перегрузка оператора — относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int sj) { i = x; j = y; }
    coord operator-(coord ob2); // бинарный минус
    coord operator-(); // унарный минус
};

// Перегрузка оператора — относительно класса coord
coord coord::operator-(coord ob2)
{
    coord temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
}

// Перегрузка унарного оператора — для класса coord
coord coord::operator-()
{
    x = -x;
    y = -y;
```

```
    return *this;
}

int main ()
{
    coord o1(10, 10), o2(5, 7);
    int x, y;

    o1 = o1 - o2; // вычитание
    o1.get_xy(x, y);
    cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

    o1 = -o1; // отрицание
    o1.get_xy(x, y);
    cout << "(-o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Как видите, если минус перегружать как бинарный оператор, то у функции будет один параметр. Если его перегружать как унарный оператор, то параметров не будет. Это отличие в числе параметров и делает возможным перегрузку минуса для обоих операторов. Как показано в программе, при использовании минуса в качестве бинарного оператора вызывается функция **operator-(coord ob2),** а в качестве унарного — функция **operator-().**

1. Перегрузите оператор — относительно класса **coord.** Создайте его префиксную и постфиксную формы.
2. Перегрузите оператор + относительно класса **coord** так, чтобы он был как бинарным (как было показано ранее), так и унарным оператором. При использовании в качестве унарного оператор + должен делать положительным значение любой отрицательной координаты.

6.5. Дружественные оператор-функции

Как отмечалось в начале этой главы, имеется возможность перегружать оператор относительно класса, используя не только функцию-член, но и дружественную функцию. Как вы знаете, дружественной функции указатель **this** не передается. В случае бинарного оператора это означает, что дружественной оператор-функции явно передаются оба операнда, а в случае унарного — один. Все остальное в обоих случаях одинаково, и нет особого смысла вместо оператор-функции — члена класса использовать дружественную

оператор-функцию, за одним важным исключением, которое будет рассмотрено в примерах.

Нельзя использовать дружественную функцию для перегрузки оператора присваивания. Оператор присваивания можно перегружать только как оператор-функцию — член класса.

1. Здесь функция **operator+()** перегружается для класса **coord** с использованием дружественной функции:

```
// Перегрузка оператора + относительно класса coord
// с использованием дружественной функции
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    friend coord operator+(coord ob1, coord ob2);
};

// Перегрузка оператора + с использованием дружественной функции
coord operator+(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x + ob2.x;
    temp.y = ob1.y + ob2.y;

    return temp;
}

int main()
{
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = ol + o2; // сложение двух объектов
                  // вызов функции operator+()
```

```
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Обратите внимание, что левый операнд передается первому параметру, а правый — второму.

2. Перегрузка оператора посредством дружественной функции дает одну очень важную возможность, которой нет у функции — члена класса. Используя дружественную оператор-функцию, в операциях с объектами можно использовать встроенные типы данных, и при этом встроенный тип может располагаться слева от оператора. Как отмечалось ранее в этой главе, можно перегрузить оператор-функцию, являющуюся членом класса, так, что левый операнд становится объектом, а правый — значением встроенного типа. Но нельзя для функции — члена класса располагать значение встроенного типа слева от оператора. Например, пусть перегружается оператор-функция — член класса, тогда первая показанная здесь инструкция правильна, а вторая нет:

```
ob1 = ob2 + 10; // правильно
ob1 = 10 + ob2; // неправильно
```

Несмотря на то, что допустимо строить выражения так, как показано в первом примере, необходимость постоянно думать о том, чтобы объект находился слева от оператора, а значение встроенного типа — справа, может быть обременительной. Решение проблемы состоит в том, чтобы сделать перегруженную оператор-функцию дружественной и задать обе возможные ситуации.

Как вы знаете, дружественной оператор-функции передаются явно *оба* операнда. Таким образом, можно задать перегружаемую дружественную функцию так, чтобы левый операнд был объектом, а правый — операндом другого типа. Затем можно снова перегрузить оператор, чтобы левый операнд был значением встроенного типа, а правый — объектом. Следующая программа иллюстрирует такой подход:

```
// Дружественные оператор-функции придают гибкость программе
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    friend coord operator+(coord ob1, int i);
    friend coord operator+(int i, coord ob1);
};
```

```
// Перегрузка оператора + для операции ob + int
coord operator+ (coord ob1, int i)
{
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;

    return temp;
}

// Перегрузка оператора + для операции int + ob
coord operator+ (int i, coord ob1)
{
    coord temp;

    temp.x = ob1.x + i;
    temp.y = ob1.y + i;

    return temp;
}

int main()
{
    coord o1 (10, 10);
    int x, y;

    o1 = o1 + 10; // объект + целое
    o1.get_xy(x,y);
    cout << "(o1 + 10) X: " << x << ", Y: " << y << "\n";

    o1 = 99 + o1; // целое + объект
    o1.get_xy(x,y);
    cout << "(99 + o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

В результате перегрузки дружественных оператор-функций становятся правильными обе инструкции:

```
o1 = o1 + 10;
o1 = 99 + o1;
```

3. При использовании дружественной оператор-функции для перегрузки унарного оператора ++ или — необходимо передавать операнд в функцию в качестве параметра-ссылки, поскольку дружественной функции не передается указатель **this.** Запомните, что в операторах инкремента и декремента подразумевается, что операнд будет изменен. Однако при перегрузке этих операторов посредством дружественных функций операнд передается по значению. Таким образом, любое изменение параметра внутри дружественной оператор-функции не влияет на объект, являющийся источником вызова. Поскольку при ис-

пользовании дружественной функции отсутствует явно передаваемый указатель на объект (т. е. указатель **this),** инкремент и декремент не влияют на операнд.

Однако при передаче операнда дружественной функции в качестве параметра-ссылки, изменения, которые имеют место внутри дружественной функции, влияют на объект, являющийся источником вызова. Например, в следующей программе посредством дружественной функции перегружается оператор ++.

```
// Перегрузка оператора ++ с использованием дружественной функции
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j ) { i = x; j = y; }
    friend coord operator++(coord &ob) ;
};

// Перегрузка оператора ++ с использованием дружественной функции
coord operator++(coord &ob) // использование ссылки
                            // в качестве параметра
{
    ob.x++;
    ob.y++;

    return ob; // возвращение объекта,
               // ставшего источником вызова
}

int main()
{
    coord o1(10, 10);
    int x, y;

    ++o1; // объект o1 передается по ссылке
    o1.get_xy(x, y);
    cout << "(++o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Если вы используете современный компилятор, то с помощью дружественной оператор-функции можно определить разницу между префиксной и постфиксной формами операторов инкремента и декремента точно так же, как это делалось с помощью функций-членов. Просто добавьте целый параметр при задании постфиксной версии. Например, здесь приводятся пре-

фиксная и постфиксная версии оператора инкремента относительно класса **coord:**

```
coord operator++(coord &ob); // префиксная версия
coord operator++(coord &ob, int notused); // постфиксная версия
```

Если оператор ++ находится перед операндом, то вызывается функция **coord operator++(coord** &ob). Однако, если оператор ++ находится после операнда, вызывается функция **coord operator++(coord &ob, int notused).** В этом случае переменной **notused** будет передано значение 0.

1. Перегрузите операторы — и / для класса **coord** посредством дружественных функций.
2. Перепишите класс **coord** так, чтобы можно было использовать объекты типа **coord** для умножения каждой из координат на целое. Должны быть корректными обе следующие инструкции: ob * int и int * ob.
3. Объясните, почему решение упражнения 2 требует использования дружественных оператор-функций.

Покажите, как с помощью дружественной оператор-функции перегрузить оператор-- относительно класса **coord.** Определите как префиксную, так и постфиксную формы.

6.6. Особенности использования оператора присваивания

Как уже отмечалось, относительно класса можно перегрузить оператор присваивания. По умолчанию, если оператор присваивания применяется к объекту, то происходит поразрядное копирование объекта, стоящего справа от оператора, в объект, стоящий слева от оператора. Если это то, что вам нужно, нет смысла создавать собственную функцию **operator=()**. Однако бывают случаи, когда точное поразрядное копирование нежелательно. В главе 3 при выделении памяти объекту вам было представлено несколько примеров подобного рода. В таких случаях требуется особая операция присваивания.

1. Здесь приведена новая версия класса **strtype**, различные формы которого изучались в предыдущих главах. В этой версии оператор = перегружается так, что указатель p при присваивании не перезаписывается.

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

class strtype {
    char *p;
    int len;
public:
    strtype(char*s);
    ~strtype() {
  cout << "Освобождение памяти по адресу " << (unsigned) p << '\n';
        delete []p;
    }
    char *get() { return p; }
    strtype &operator=(strtype &ob);
};

strtype::strtype(char*s)
{
    int 1;

    1 = strlen(s) + 1;

    p = new char [l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    len = 1;
    strcpy(p,s);
}
// Присваивание объекта
strtype &strtype::operator=(strtype &ob)
{
    // выяснение необходимости дополнительной памяти
    if(len< ob.len) { // требуется выделение дополнительной памяти
        delete[] p;
        p = new char [ob.len];
        if(!p) {
            cout << "Ошибка выделения памяти\n";
            exit(1);
        }
    }
    len = ob.len;
    strcpy(p,ob.p);
    return *this;
}
```

```
int main()
{
    strtypea("Привет"),b("Здесь");

    cout<<a.get()<< '\n' ;
    cout << b.get() << '\n';

    a = b; // теперь указатель p не перезаписывается

    cout<<a.get()<< '\n' ;
    cout << b.get() << '\n';

    return 0;
}
```

Как видите, перегрузка оператора присваивания предотвращает перезапись указателя p. При первом контроле выясняется, достаточно ли в объекте слева от оператора присваивания выделено памяти для хранения присваиваемой ему строки. Если это не так, то память освобождается и выделяется новый фрагмент. Затем строка копируется в эту память, а длина строки копируется в переменную **len**.

Отметьте два важных свойства функции **operator=()**. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, стоящего справа от оператора присваивания. Как известно по предыдущим главам, при передаче в функцию объекта создается его копия, и эта копия удаляется при завершении работы функции. В этом случае для удаления копии должен вызываться деструктор, который освобождает память, обозначенную указателем p. Однако память по адресу p все еще необходима объекту, который является аргументом. Параметр-ссылка помогает решить проблему.

Вторым важным свойством функции **operator=()** является то, что она возвращает не объект, а ссылку на него. Смысл этого тот же, что и при обычном использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после полного завершения ее работы. Это означает, что для временного объекта будет вызван деструктор, который вызовет освобождение памяти по адресу p, но указатель p (и память на которую он ссылается) все еще необходимы для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

Как вы узнали из главы 5, создание конструктора копий — это другой путь решения проблем, описанных в двух предыдущих разделах. Но конструктор копий может оказаться не столь эффективным решением, как ссылка в качестве параметра и ссылка в качестве возвращаемого значения функции. Это происходит потому, что использование ссылки исключает затраты ресурсов,

связанные с копированием объекта в каждом из двух указанных случаев. Как видите, в C++ часто имеется несколько способов достижения одной и той же цели. Понимание их преимуществ и недостатков — это часть процесса вашего становления как профессионального программиста C++.

Упражнения

1. Пусть дано следующее объявление класса, добавьте все необходимое для создания типа динамический массив. То есть выделите память для массива и сохраните указатель на эту память по адресу p. Размер массива в байтах сохраните в переменной **size.** Создайте функцию put(), возвращающую ссылку на заданный элемент массива и функцию **get(),** возвращающую значение заданного элемента. Обеспечьте контроль границ массива. Кроме этого перегрузите оператор присваивания так, чтобы выделенная каждому массиву такого типа память не была случайно повреждена при присваивании одного массива другому. (В следующем разделе будет показан более совершенный способ решения этого упражнения.)

```
class dynarray {
    int *p;
    int size;
public:
    dynarray(int s); // передача размера массива в переменной s
    int &put(int i); // возвращение ссылки на элемент i
    int get(int i); // возвращение значения переменной i
    // создайте функцию operator=()
};
```

6.7. Перегрузка оператора индекса массива []

Последним оператором, который мы научимся перегружать, будет оператор индекса массива []. В C++ при перегрузке оператор **[]** рассматривается как бинарный. Оператор **[]** можно перегружать только как функцию-член. Ниже представлена основная форма оператор-функции — члена класса **operator[]()**:

```
тип имя_класса::operator[](int индекс)
{
    // ...
}
```

С технической точки зрения тип параметра не обязательно должен быть целым, но поскольку оператор-функция **operator[](),** как правило, используется для получения индекса массива, то ее параметр обычно имеет тип **int.**

Чтобы понять, как работает оператор [], представим, что объект O индексируется следующим образом:

```
O[9]
```

Этот индекс транслируется в вызов функции **operator[]()**:

```
O.operator[](9)
```

Таким образом, значение выражения внутри оператора индексирования явно передается функции **operator[]()** в качестве параметра. При этом указатель **this** будет ссылаться на объект O, являющийся источником вызова.

1. В следующей программе объявляется состоящий из пяти целых массив **arraytype.** Каждый элемент массива инициализируется конструктором. Перегруженная функция **operator[]()** возвращает элемент, заданный ее параметром.

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

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for (i=o; i<SIZE; i++) a[i] = i;
    }
    int operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    return 0;
}
```

В результате работы программы на экран выводится следующее:

```
0 1 2 3 4
```

В этом и остальных примерах инициализация массива a с помощью конструктора выполнена исключительно в иллюстративных целях и на самом деле не требуется.

2. Имеется возможность перегрузить функцию **operator[]()** так, чтобы в инструкции присваивания оператор [] можно было располагать как слева, так и справа от оператора =. Для этого возвратите ссылку на индексируемый элемент. Следующая программа иллюстрирует такой подход.

```
ttinclude<iostream>
using namespace std;

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for (i=o; i<SIZE; i++) a[i] = i;
    }
    int &operator[](int i) { return a[i]; }
};

int main()
{
    arraytype ob;
    int i;

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    cout << "\n";

    // добавление значения 10 к каждому элементу массива
    for(i=0; i<SIZE; i++)
        ob[i] = ob[i]+10; // оператор [] слева от оператора =

    for(i=0; i<SIZE; i++)
        cout << ob[i] << " ";

    return 0;
}
```

В результате работы программы на экран выводится следующее:

```
0 1 2 3 4
10 11 12 13 14
```

Поскольку теперь функция **operator[]()** возвращает ссылку на элемент массива с индексом i, то для изменения этого элемента оператор [] можно рас-

положить слева в инструкции присваивания. (Естественно, что как и прежде его можно располагать справа.) Таким образом, с объектами типа **arraytype** можно обращаться так же, как и с обычными массивами.

3. Перегрузка оператора **[]** дает возможность по-новому взглянуть на задачу индексирования безопасного массива. Ранее в этой книге мы рассматривали простейший способ реализации безопасного массива, в котором для доступа к элементам массива использовались функции **get()** и **put()**. Перегрузка оператора **[]** позволит нам теперь создать такой массив гораздо проще. Вспомните, что безопасный массив — это массив, который инкапсулирован в классе, и при этом класс обеспечивает контроль границ массива. Такой подход предотвращает нарушение границ массива. Благодаря перегрузке оператора **[]**, работать с безопасным массивом можно так же, как с обычным.

   Для создания безопасного массива просто реализуйте в функции **operator[]()** контроль границ. Кроме этого, функция **operator[]()** должна возвращать ссылку на индексируемый элемент. Например, в представленном ниже примере в предыдущую программу добавлен контроль границ массива, что позволяет при нарушении границ генерировать соответствующую ошибку.

```
// Пример безопасного массива
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 5;

class arraytype {
    int a[SIZE];
public:
    arraytype() {
        int i;
        for (i=0; i<SIZE; i++) a[i] = i;
    }
    int &operator[](int i);
};

// Обеспечение контроля границ для массива типа arraytype
int &arraytype::operator[](int i)
{
    if(i<0 || i>SIZE-1) {
        cout << "\nЗначение индекса ";
        cout << i << " находится за пределами границ массива. \n";
        exit(1);
    }
    return a[i];
}
```

```
int main()
{
    arraytype ob;
    int i;

    // Здесь проблем нет
    for(i=0; i<SIZE; i++)
      cout << ob[i] << " ";

    /* А здесь при выполнении программы генерируется ошибка,
поскольку значение SIZE+100 не входит в заданный диапазон */
    ob[SIZE+100] = 99; // Ошибка!!!

    return 0;
}
```

Благодаря контролю границ, реализованному в функции **operator[]()**, при выполнении инструкции

```
ob[SIZE+100] = 99;
```

программа завершится еще до того, как будет повреждена какая-либо ячейка памяти.

Поскольку перегрузка оператора **[]** позволяет создавать безопасные массивы, которые выглядят и функционируют так же, как самые обычные массивы, их можно безболезненно добавить в вашу программную среду. Однако будьте внимательны. Безопасный массив увеличивает расход ресурсов, что не во всех ситуациях может оказаться приемлемым. Фактически, именно из-за непроизводительного расхода ресурсов в C++ отсутствует встроенный контроль границ массивов. Тем не менее, в тех приложениях, в которых желательно обеспечить целостность границ, реализация безопасного массива будет лучшим решением.

Упражнения

1. Переделайте пример 1 из раздела 6.6 так, чтобы относительно класса **strtype** перегрузить оператор **[].** Этот оператор должен возвращать символ по заданному индексу. Кроме этого, необходима возможность задавать оператор [] в левой части инструкции присваивания. Покажите, что ваша программа работает.
2. Измените ваше решение упражнения 1 из раздела 6.6 так, чтобы оператор **[]** использовать для индексирования динамического массива. То есть замените функции **get() и put()** оператором **[].**

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы:

1. Перегрузите операторы сдвига >> и << относительно класса **coord** так, чтобы стали возможными следующие типы операций:

```
ob << integer
ob >> integer
```

   Удостоверьтесь, что ваши операторы действительно сдвигают значения x и y на заданное количество разрядов.

2. Пусть дан класс

```
class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x = 0; y = 0; z = 0; }
    void get(int &i, int &j, int Sk)
    {
        i = x; j = y; k = z;
    }
};
```

   Перегрузите для этого класса операторы +, −, ++ и −−. (Для операторов инкремента и декремента перегрузите только префиксную форму.)

3. Измените ваше решение вопроса 2 так, чтобы в оператор-функциях вместо параметров-значений использовать параметры-ссылки. *(Подсказка.* Для операторов инкремента и декремента вам потребуются дружественные функции.)

4. Чем действие дружественной оператор-функции отличается от действия оператор-функции — члена класса?

5. Объясните, почему может потребоваться перегрузка оператора присваивания.

6. Может ли функция **operator=()** быть дружественной?

7. Перегрузите оператор + для класса **three_d** из вопроса 2 так, чтобы иметь возможность выполнять следующие типы операций:

```
ob + int;
int + ob;
```

8. Перегрузите операторы ==, != и || относительно класса **three_d** из вопроса 2.
9. Приведите главный довод в пользу перегрузки оператора [].

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Создайте класс **strtype**, который допускает следующие типы операций:
   - Конкатенацию строк с помощью оператора +
   - Присваивание строк с помощью оператора =
   - Сравнение строк с помощью операторов <, > и ==

Можете пользоваться строками фиксированной длины. На первый взгляд это может показаться непростой задачей, но, немного подумав (и поэкспериментировав), вы должны справиться.

Глава 7

Наследование

Ранее в этой книге вы познакомились с концепцией наследования. Сейчас пришло время осветить эту тему более детально. Наследование — это один из трех базовых принципов ООР, и потому является одним из важнейших инструментов C++. В C++ наследование используется не только для поддержки иерархии классов, но, как вы узнаете из главы 10, и для поддержки другого важнейшего инструмента ООР — полиморфизма.

Материал, который приведен в этой главе, включает в себя следующие темы: управление доступом к базовому классу, спецификатор доступа **protected,** множественное наследование, передача аргументов конструкторам базового класса, виртуальные базовые классы.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения. •

1. Теряет ли оператор при перегрузке что-либо из своей исходной функциональности?
2. Нужно ли перегружать оператор относительно определенного пользователем типа данных, например, класса?
3. Можно ли изменить приоритет перегруженного оператора? Можно ли изменить количество операндов?
4. Дана следующая, почти законченная программа, добавьте недостающие оператор-функции:

```
ftinclude<iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
```

```
    void set(int n[10]);
    void show();
    array operator+(array ob2);
    array operator-(array ob2);
    int operator==(array ob2) ;
};

array::array()
{
    int i;
    for(i = 0; i < 10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;
    for(i = 0; i < 10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;
    for(i = 0; i < 10; i++)
        cout << nums[i] << ' ';
    cout << "\n";
}

// Впишите оператор-функции

int main()
{
    array o1, o2, o3;

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

    o1.set(i);
    o2.set(i);

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o3;
    o3.show();

    if(o1==o2) cout << "o1 равно o2\n";
    else cout << "o1 не равно o2\n";

    if(o1==o3) cout << "o1 равно o3\n";
    else cout << "o1 не равно o3\n";

    return 0;
}
```

Перегруженный оператор + должен поэлементно складывать оба операнда. Перегруженный оператор — должен вычитать все элементы правого операнда из элементов левого. Перегруженный оператор == должен возвращать значение **true**, если все элементы обоих операндов равны, в противном случае он должен возвращать значение **false.**

5. Переработайте решение упражнения 4 так, чтобы перегрузить операторы с использованием дружественных функций.
6. Используя класс и функции из вопроса 4, перегрузите оператор ++ с помощью функции — члена класса, а оператор — с помощью дружественной функции. (Перегрузите только префиксные формы операторов ++ и --.)
7. Можно ли, используя дружественную функцию, перегрузить оператор присваивания?

7.1. Управление доступом к базовому классу

Когда один класс наследуется другим, используется следующая основная форма записи:

```
class имя_производного_класса: сп_доступа имя_базового_класса {
    // ...
}
```

Здесь *сп_доступа* — это одно из трех ключевых слов: **public, private** или **protected.** Обсуждение спецификатора доступа **protected** отложим до следующего раздела этой главы. Здесь рассмотрим спецификаторы **public и private.**

Спецификатор доступа (access specifier) определяет то, как элементы базового класса (base class) наследуются производным классом (derived class). Если спецификатором доступа наследуемого базового класса является ключевое слово **public,** то все открытые члены базового класса остаются открытыми и в производном. Если спецификатором доступа наследуемого базового класса является ключевое слово **private,** то все открытые члены базового класса в производном классе становятся закрытыми. В обоих случаях все закрытые члены базового класса в производном классе остаются закрытыми и недоступными.

Важно понимать, что если спецификатором доступа является ключевое слово **private,** то хотя открытые члены базового класса становятся закрытыми в производном, они остаются доступными для функций — членов производного класса.

Технически спецификатор доступа не обязателен. Если спецификатор доступа не указан и производный класс определен с ключевым словом **class,** то базовый класс по умолчанию наследуется как закрытый. Если спецификатор доступа не указан и производный класс определен с ключевым словом **struct,** то базовый класс по умолчанию наследуется как открытый. Тем не менее, для ясности большинство программистов предпочитают явное задание спецификатора доступа.

1. Здесь представлены базовый и наследующий его производный классы (наследование со спецификатором **public):**

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

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Класс наследуется как открытый
class derived: public base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setx(10); // доступ к члену базового класса
    ob.sety(20); // доступ к члену производного класса

    ob.showx(); // доступ к члену базового класса
    ob.showy(); // доступ к члену производного класса

    return 0;
}
```

Как показано в программе, поскольку класс **base** наследуется как открытый, открытые члены класса **base** — функции **setx() и showx()** — становятся открытыми производного класса **derived** и поэтому доступны из любой части

программы. Следовательно, совершенно правильно вызывать эти функции из функции **main()**.

2. Важно понимать, что наследование производным классом базового как открытого совсем не означает, что для производного класса станут доступными закрытые члены базового. Например, это небольшое изменение в классе **derived** из предыдущего примера неправильно:

```
class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};

// Класс наследуется как открытый
class derived: public base {
    int y;
public:
    void sety(int n) { y = n; }

    /* Закрытые члены базового класса недоступны. x — это закрытый
член базового класса и поэтому внутри производного класса он
недоступен */
    void show_sum() { cout << x+y << '\n'; } // Ошибка!!!

    void showy() { cout << y << '\n'; }
};
```

Здесь в производном классе **derived** сделана попытка доступа к переменной x, которая является закрытым членом базового класса **base.** Это неверно, поскольку закрытые члены базового класса остаются закрытыми, *независимо от того, как он наследуется.*

3. Ниже представлена слегка измененная версия программы из примера 1. Базовый класс **base** наследуется как закрытый, т. е. с ключевым словом **private.** Такое изменение, как показано в комментариях, при компиляции ведет к ошибке.

```
// В этой программе есть ошибка
ttinclude<iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int n) { x = n; }
    void showx() { cout << x << '\n'; }
};
```

```
// Класс наследуется как закрытый
class derived: private base {
    int y;
public:
    void sety(int n) { y = n; }
    void showy() { cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setx(10); // ОШИБКА — теперь закрыто для производного класса
    ob.sety(20); // правильный доступ к члену производного класса

    ob.showx(); // ОШИБКА — теперь закрыто для производного класса
    ob.showy(); // правильный доступ к члену производного класса

    return 0;
}
```

Как отражено в комментариях к этой (неправильной) программе, функции **showx()** и setx() становятся закрытыми в производном классе и недоступными вне его.

Запомните, что функции **showx()** и **setx()** в базовом классе **base** по-прежнему остаются открытыми независимо от того, как они наследуются производным классом. Это означает, что объект типа **base** мог бы получить доступ к этим функциям в любом месте программы. Однако для объектов типа **derived** они становятся закрытыми. Например, в данном фрагменте:

```
base base_ob;
    base_ob.setx(1); // правильно, поскольку объект base_ob
                     // имеет тип base
```

вызов функции **setx()** правилен, поскольку функция **setx()** — это открытый член класса **base.**

4. Как мы уже узнали, хотя открытые члены базового класса при наследовании с использованием спецификатора **private** в производном классе становятся закрытыми, *внутри* производного класса они остаются доступными. Например, ниже представлена исправленная версия предыдущей программы:

```
// Исправленная версия программы
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int n) { x = n; }
```

```
    void showx() { cout << x << '\n'; }
};

// Класс наследуется как закрытый
class derived: private base {
    int y;
public:
    // переменная setx доступна внутри класса derived
    void setxy(int n, int m) { setx(n); y = m; }
    // переменная showx доступна внутри класса derived
    void showxy() { showx(); cout << y << '\n'; }
};

int main()
{
    derived ob;

    ob.setxy(10, 20);

    ob.showxy();

    return 0;
}
```

В данном случае функции showx() и setx() доступны *внутри* производного класса, что совершенно правильно, поскольку они являются закрытыми членами этого класса.

1. Исследуйте следующую конструкцию:

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

class mybase {
    int a, b;
public:
    int c;
    void setab(int i, int j) { a = i; b = j; }
    void getab(int &i, int &j) { i = a; j = b; }
};

class derivedl: public mybase {
// ...
};
```

```
class derived2: private mybase {
// ...
};

int main()
{
    derivedl ol;
    derived2 o2;
    int i, j;

    // ...
}
```

Какая из следующих инструкций правильна внутри функции **main()**?

A. `o1.getab(i, j);`

B. `o2.getab(i, j);`

C. `o1.c = 10;`

D. `o2.c = 10;`

2. Что происходит, когда открытые члены базового класса наследуются как открытые? Что происходит, когда они наследуются как закрытые?
3. Если вы этого еще не сделали, попытайтесь выполнить все примеры, представленные в этом разделе. Поэкспериментируйте со спецификаторами доступа и изучите результаты.

7.2. Защищенные члены класса

Как вы узнали из предыдущего раздела, у производного класса нет доступа к закрытым членам базового. Это означает, что если производному классу необходим доступ к некоторым членам базового, то эти члены должны быть открытыми. Однако возможна ситуация, когда необходимо, чтобы члены базового класса, оставаясь закрытыми, были доступны для производного класса. Для реализации этой идеи в C++ включен спецификатор доступа **protected** (защищенный).

Спецификатор доступа **protected** эквивалентен спецификатору **private** с единственным исключением: защищенные члены базового класса доступны для членов всех производных классов этого базового класса. Вне базового или производных классов защищенные члены недоступны.

Спецификатор доступа **protected** может находиться в любом месте объявления класса, хотя обычно его располагают после объявления закрытых членов (задаваемых по умолчанию) и перед объявлением открытых членов. Ниже показана полная основная форма объявления класса:

```
class имя_класса {
    // закрытые члена
protected: // необязательный спецификатор
    // защищенные члена
public:
    // открытые члены
};
```

Когда базовый класс наследуется производным классом как открытый **(public),** защищенный член базового класса становится защищенным членом производного класса. Когда базовый класс наследуется как закрытый **(private),** то защищенный член базового класса становится закрытым членом производного класса.

Базовый класс может также наследоваться производным классом как защищенный **(protected).** В этом случае открытые и защищенные члены базового класса становятся защищенными членами производного класса. (Естественно, что закрытые члены базового класса остаются закрытыми, и они не доступны для производного класса.)

Спецификатор доступа **protected** можно также использовать со структурами.

Примеры

1. В этой программе проиллюстрирован доступ к открытым, закрытым и защищенным членам класса:

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

class samp {
    // члены класса, закрытые по умолчанию
    int a;
protected: // тоже закрытые члены класса samp
    int b;
public:
    int c;

    samp(int n, int m) { a = n; b = m; }
    int geta() { return a; }
    int getb() { return b; }
};

int main()
{
    samp ob(10, 20);
```

```
\
    // ob.b = 99; Ошибка! Переменная b защищена и поэтому закрыта
    ob.c = 30; // Правильно! Переменная c
               // является открытым членом класса samp

    cout << ob.geta() << ' ';
    cout << ob.getb() << ' ' << ob.c << '\n';

    return 0;
}
```

Как вы могли заметить, выделенная в комментарий строка содержит инструкцию, недопустимую в функции **main()**, поскольку переменная b является защищенной и таким образом по-прежнему закрытой для класса **samp.**

2. В следующей программе показано, что происходит, если защищенные члены класса наследуются как открытые:

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

class base {
protected:      // закрытые члены класса base,
    int a,b;    // но для производного класса они доступны
public:
    void setab(int n, int m) { a = n; b = m; }
};

class derived: public base {
    int c;
public:
    void setc(int n) { c = n; }

    // эта функция имеет доступ к переменным a и b класса base
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};

int main()
{
    derived ob;

    /* Переменные a и b здесь недоступны, поскольку являются
закрытыми членами классов base и derived
*/

    ob.setab(1,2);
    ob.setc(3);

    ob.showabc();

    return 0;
}
```

Поскольку переменные a и b в классе **base** защищены и наследуются производным классом **derived** как открытые члены, они доступны для использования функциями — членами класса **derived.** Однако вне двух этих классов они в полной мере закрыты и недоступны.

3. Как упоминалось ранее, если базовый класс наследуется как защищенный, открытые и защищенные члены базового класса становятся защищенными членами производного класса. Например, в слегка измененной версии программы из предыдущего примера класс **base** наследуется не как открытый, а как защищенный:

```
// Эта программа компилироваться не будет
#include <iostream>
using namespace std;

class base {
protected:      // закрытые члены класса base,
    int a,b;    // но для производного класса они доступны
public:
    void setab(int n, int m) { a = n; b = m; }
};

class derived: protected base { // класс base наследуется
                                // как защищенный
    int c;
public:
    void setc(int n) { c = n; }

    // эта функция имеет доступ к переменным a и b класса base
    void showabc() {
        cout << a << ' ' << b << ' ' << c << '\n';
    }
};

int main()
{
    derived ob;

    // ОШИБКА: теперь функция setab()
    // является защищенным членом класса base
    ob.setab(1, 2); // функция setab() здесь недоступна

    ob.setc(3);

    ob.showabc();

    return 0;
}
```

Как указано в комментариях, поскольку класс **base** наследуется как защищенный, его открытые и защищенные элементы становятся защищенными членами производного класса **derived** и следовательно внутри функции main() они недоступны.

1. Что происходит с защищенным членом класса, когда класс наследуется как открытый? Что происходит, когда он наследуется как закрытый?
2. Объясните, зачем нужна категория защищенности protected?
3. В вопросе 1 из раздела 7.1, если бы переменные a и b внутри класса myclass стали не закрытыми (по умолчанию), а защищенными членами, изменился бы какой-нибудь из ваших ответов на вопросы этого упражнения? Если да, то почему?

7.3. Конструкторы, деструкторы и наследование

Базовый класс, производный класс или оба класса вместе могут иметь конструкторы и/или деструкторы. В этой главе исследуется несколько следствий такого положения.

Если у базового и у производного классов имеются конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы — в обратном порядке. Таким образом, конструктор базового класса выполняется раньше конструктора производного класса. Для деструкторов правилен обратный порядок: деструктор производного класса выполняется раньше деструктора базового класса.

Последовательность выполнения конструкторов и деструкторов достаточно очевидна. Поскольку базовый класс "не знает" о существовании производного, любая инициализация выполняется в нем независимо от производного класса и возможно становится основой для любой инициализации, выполняемой в производном классе. Поэтому инициализация в базовом классе должна выполняться первой.

С другой стороны, деструктор производного класса должен выполняться раньше деструктора базового класса потому, что базовый класс лежит в основе производного. Если бы деструктор базового класса выполнялся первым, это бы разрушило производный класс. Таким образом, деструктор производного класса должен вызываться до того, как объект прекратит свое существование.

Пока что ни в одном из предыдущих примеров мы не передавали аргументы для конструктора производного или базового класса. Однако это вполне возможно. Когда инициализация проводится только в производном классе, аргументы передаются обычным образом. Однако при необходимости передать аргумент конструктору базового класса ситуация несколько усложняется. Во-первых, все необходимые аргументы базового и производного классов передаются конструктору производного класса. Затем, используя

расширенную форму объявления конструктора производного класса, соответствующие аргументы передаются дальше в базовый класс. Синтаксис передачи аргументов из производного в базовый класс показан ниже:

```
конструктор_произв_класса(список-арг): базов_класс(список_арг) {
    // тело конструктора производного класса
}
```

Для базового и производного классов допустимо использовать одни и те же аргументы. Кроме этого, для производного класса допустимо игнорирование всех аргументов и передача их напрямую в базовый класс.

Приме Р

1. В этой очень короткой программе показано, в каком порядке выполняются конструкторы и деструкторы базового и производного классов:

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

class base {
public:
    base() { cout << "Работа конструктора базового класса\n"; }
    ~base() { cout << "Работа деструктора базового класса\n"; }
};

class derived: public base {
public:
    derived() { cout << "Работа конструктора производного класса\n"; }
    ~derived() { cout << "Работа деструктора производного класса\n"; }
};

int main()
{
    derived o;

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
Работа конструктора базового класса
Работа конструктора производного класса
Работа деструктора производного класса
Работа деструктора базового класса
```

Как видите, конструкторы выполняются в порядке наследования, а деструкторы — в обратном порядке.

2. В этой программе показана передача аргумента конструктору производного класса:

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

class base {
public:
    base() { cout << "Работа конструктора базового класса\n"; }
    ~base() { cout << "Работа деструктора базового класса\n"; }
};

class derived: public base {
    int j;
public:
    derived(int n) {
        cout << "Работа конструктора производного класса\n";
        j = n;
    }
    ~derived() { cout << "Работа деструктора производного класса\n"; }
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showj();

    return 0;
}
```

Обратите внимание, что аргумент передается конструктору производного класса обычным образом.

3. В следующем примере у конструкторов производного и базового классов имеются аргументы. В этом особом случае оба конструктора используют один и тот же аргумент, и производный класс просто передает этот аргумент в базовый класс.

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

class base {
    int i;
public:
    base(int n) {
        cout << "Работа конструктора базового класса\n";
        i = n;
    }
```

```
    ~base() { cout << "Работа деструктора базового класса\n"; }
    void showi() { cout << i << '\n'; }
};

class derived: public base {
    int j;
public:
    derived(int n): base(n) { // передача аргумента
                              // в базовый класс
        cout << "Работа конструктора производного класса\n";
        j = n;
    }
    ~derived() { cout << "Работа деструктора производного класса\n";
}
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10);

    o.showi();
    o.showj();

    return 0;
}
```

Обратите особое внимание на объявление конструктора производного класса. Отметьте, как параметр n (который получает аргумент при инициализации) используется в конструкторе **derived()** и передается конструктору **base().**

4. Обычно конструкторы базового и производного классов *не* используют один и тот же аргумент. В этом случае, при необходимости передать каждому конструктору класса один или несколько аргументов, вы должны передать конструктору производного класса *все* аргументы, необходимые конструкторам *обоих* классов. Затем конструктор производного класса просто передает конструктору базового класса те аргументы, которые ему требуются. Например, в представленной ниже программе показано, как передать один аргумент конструктору производного класса, а другой — конструктору базового класса:

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

class base {
    int i;
public:
    base(int n) {
        cout << "Работа конструктора базового класса\n";
        i = n;
    }
```

```
    ~base() { cout << "Работа деструктора базового класса\n"; }
    void showi() { cout << i << '\n'; }
};

class derived: public base {
    int j;
public:
    derived(int n, int m) : base(m) { // передача аргумента
                                      // в базовый класс
        cout << "Работа конструктора производного класса\n";
        j = n;
    }
    ~derived() { cout << "Работа деструктора производного класса\n";
}
    void showj() { cout << j << '\n'; }
};

int main()
{
    derived o(10, 20);

    o.showi();
    o.showj();

    return 0;
}
```

5. Конструктору производного класса совершенно нет необходимости как-то обрабатывать аргумент, предназначенный для передачи в базовый класс. Если производному классу этот аргумент не нужен, он его просто игнорирует и передает в базовый класс. Например, в этом фрагменте параметр n конструктором **derived()** не используется. Вместо этого он просто передается конструктору **base()**:

```
class base {
    int i;
public:
    base(int n) {
        cout << "Работа конструктора базового класса\n";
        i = n;
    }
    ~base() { cout << "Работа деструктора базового класса\n"; }
    void showi() { cout << i << '\n'; }
};

class derived: public base {
    int j;
public:
    derived(int n): base(n) { // передача аргумента в базовый класс
```

```
        cout << "Работа конструктора производного класса\n";
        j = 0; // аргумент п здесь не используется
    }
    ~derived() { cout << "Работа деструктора производного класса\n";
}
    void showj() { cout << j << '\n'; }
    };
```

1. В приведенном ниже фрагменте добавьте конструктор для класса **myderived.** Он должен передать указатель на инициализируемую строку конструктору класса mybase. Кроме того, конструктор **myderived()** должен инициализировать переменную **len** длиной строки.

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

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived: public mybase {
    int len;
public:
    // добавьте здесь конструктор myderived()
    int getlen() { return len; }
    void show() { cout << get() << '\n'; }
};

int main()
{
    myderived ob("привет");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}
```

2. Используя следующий фрагмент, создайте соответствующие конструкторы car() и **truck()**. Они должны передавать необходимые аргументы объектам класса **vehicle.** Кроме этого конструктор car() должен при создании объекта

инициализировать переменную **passengers,** а конструктор **truck()** — переменную **loadlimit**.

```
tfinclude <iostream>
using namespace std;

// Базовый класс автомобилей для разных типов
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
        cout << "Число колес" << num_wheels << '\n';
        cout << "Грузоподъемность" << range << ' \n ' ;
    }
};

class car: public vehicle {
    int passengers;
public:
    // вставьте здесь конструктор car()
    void show()
        {
            showv();
            cout << "Пассажироемкость" << passengers << '\n';
        }
};

class truck: public vehicle {
    int loadlimit;
public:
    // вставьте здесь конструктор truck()
    void show()
        {
            showv();
            cout << "Пробег" << loadlimit << '\n';
        }
};

int main()
{
    car c(5, 4, 500);
    truck t(30000, 12, 1200);
```

```
    cout << "легковая машина:\n";
    c.show();
    cout << "\nгрузовик:\n";
    t.show();

    return 0;
}
```

Для конструкторов car() и **truck**() объекты должны объявляться следующим образом:

```
car ob(passengers, wheels, range);
    truck ob(loadlimit, wheels, range);
```

7.4. Множественное наследование

Имеются два способа, посредством которых производный класс может наследовать более одного базового класса. Во-первых, производный класс может использоваться в качестве базового для другого производного класса, создавая многоуровневую иерархию классов. В этом случае говорят, что исходный базовый класс является *косвенным (indirect)* базовым классом для второго производного класса. (Отметьте, что любой класс — независимо от того, как он создан — может использоваться в качестве базового класса.) Во-вторых, производный класс может прямо наследовать более одного базового класса. В такой ситуации созданию производного класса помогает комбинация двух или более базовых классов. Ниже исследуются результаты, к которым приводит наследование нескольких базовых классов.

Когда класс используется как базовый для производного, который, в свою очередь, является базовым для другого производного класса, конструкторы всех трех классов вызываются в порядке наследования. (Это положение является обобщением ранее исследованного принципа.) Деструкторы вызываются в обратном порядке. Таким образом, если класс **B1** наследуется классом **D1**, а D1 — классом **D2,** то конструктор класса **B1** вызывается первым, за ним конструктор класса **D1,** за которым, в свою очередь, конструктор класса **D2.** Деструкторы вызываются в обратном порядке.

Если производный класс напрямую наследует несколько базовых классов, используется такое расширенное объявление:

```
class имя_производного_класса: сп_доступа имя_базового_класса1,
                               сп_доступа имя_базового_класса2,
                            ..., сп_доступа имя_базового-классаN
{
    // ... тело класса
}
```

Здесь *имя_базового_класса1*... *имя_базового_классаN*— имена базовых классов, *сп_доступа* — спецификатор доступа, который может быть разным у разных базовых классов. Когда наследуется несколько базовых классов, конструкторы выполняются слева направо в том порядке, который задан в объявлении производного класса. Деструкторы выполняются в обратном порядке.

Когда класс наследует несколько базовых классов, конструкторам которых необходимы аргументы, производный класс передает эти аргументы, используя расширенную форму объявления конструктора производного класса:

```
констр_произв_класса(список_арг): имя_базового_класса1(список_арг),
                                  имя_базового_класса2(список_арг),
                                  ..., имя_базового_классаN(список_арг)
{
    // ... тело конструктора производного класса
}
```

Здесь *имя_базового_класса1*... *имя_базового_классаN*— имена базовых классов.

Если производный класс наследует иерархию классов, каждый производный класс должен передавать предшествующему в цепочке базовому классу все необходимые аргументы.

Примеры

1. В этом примере производный класс наследует класс, производный от другого класса. Обратите внимание, как аргументы передаются по цепочке от класса D2 к классу **B1.**

```
// Множественное наследование
#include <iostream>
using namespace std;

class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// Прямое наследование базового класса
class D1: public B1 {
    int b;
public:
    D1(int x, int y): B1(y) // передача переменной y классу B1
```

```
    {
        b = x;
    }
    int getb() { return b; }
};

// Прямое наследование производного класса
// и косвенное наследование базового класса
class D2: public D1 {
    int c;
public:
    D2(int x, int y, int z) : D1(y, z) /У передача аргументов
                                       // классу D1
    {
        c = x;
    }

/* Поскольку базовые классы наследуются как открытые, класс D2 имеет
доступ к открытым элементам классов B1 и D1 */
    void show() {
        cout << geta() << ' ' << getb() << ' ' ;
        cout << c << '\n';
    }
};

int main()
{
    D2 ob(1, 2, 3);

    ob.show();
    // функции geta() и getb() здесь тоже открыты
    cout << ob.geta() << ' ' << ob.getb() << '\n';

    return 0;
}
```

Вызов функции **ob.show()** выводит на экран значения 3 2 1. В этом примере класс **B1** является косвенным базовым классом для класса **D2.** Отметьте, что класс **D2** имеет доступ к открытым членам классов D1 и B1. Как вы уже должны знать, при наследовании открытых членов базового класса они становятся открытыми членами производного класса. Поэтому, если класс **D1** наследует класс **B1**, **то** функция geta() становится открытым членом класса D1 и затем открытым членом класса **D2.**

Как показано в программе, каждый класс иерархии классов должен передавать все аргументы, необходимые каждому предшествующему базовому классу. Невыполнение этого правила приведет к ошибке при компиляции программы.

Здесь показана иерархия классов предыдущей программы:

Перед тем как двигаться дальше, необходимо небольшое замечание о стиле изображения графов наследования в C++. Обратите внимание на то, что в предыдущем графе стрелки направлены не вниз, а вверх.

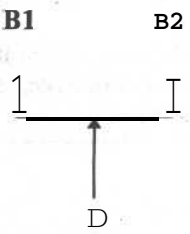
Традиционно программисты C++ изображают отношения наследования в виде прямых графов, стрелки которых направлены от производного к базовому классу. Хотя новички могут посчитать такой подход излишне схематичным, именно он обычно практикуется в C++.

2. Здесь представлена переработанная версия предыдущей программы, в которой производный класс прямо наследует два базовых класса:

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

// Создание первого базового класса
class B1 {
    int a;
public:
    B1(int x) { a = x; }
    int geta() { return a; }
};

// Создание второго базового класса
class B2 {
    int b;
public:
    B2(int x)
    {
        b = x;
    }
    int getb() { return b; }
};

// Прямое наследование двух базовых классов
class D: public B1, public B2 {
    int c;
```

```
public:
    // здесь переменные z и y
    // напрямую передаются классам B1 и B2
    D(int x, int y, int z) : B1(z), B2(y)
    {
        c = x;
    }

/* Поскольку базовые классы наследуются как открытые, класс D имеет
доступ к открытым элементам классов B1 и B2
*/
    void show() {
        cout << geta() << ' ' << getb() << ' ';
        cout << c << '\n';
    }
};

int main()
{
    D ob(1, 2, 3);

    ob.show();

    return 0;
}
```

В этой версии программы класс D передает аргументы по отдельности классам B1 и B2. Теперь иерархия классов выглядит таким образом:



3. В следующей программе показан порядок, в котором вызываются конструкторы и деструкторы, когда производный класс прямо наследует несколько базовых классов:

```
ttinclude<iostream>
using namespace std;

class Bl {
public:
    B1() { cout << "Работа конструктора класса B1\n"; }
    ~B1() { cout << "Работа деструктора класса B1\n"; }
};
```

```
class B2 {
    int b;
public:
    B2() { cout << "Работа конструктора класса B2\n"; }
    ~B2() { cout << "Работа деструктора класса B2\n"; }
};

// Наследование двух базовых классов
class D: public B1, public B2 {
public:
    D() { cout << "Работа конструктора класса D\n"; }
    ~D() { cout << "Работа деструктора класса D\n"; }
};

int main()
{
    D ob;

    return 0;
}
```

Эта программа выводит на экран следующее:

```
Работа конструктора класса B1
Работа конструктора класса B2
Работа конструктора класса D
Работа деструктора класса D
Работа деструктора класса B2
Работа деструктора класса B1
```

Как уже установлено, когда прямо наследуются несколько базовых классов, конструкторы вызываются слева направо в порядке, задаваемом списком. Деструкторы вызываются в обратном порядке.

1. Что выводит на экран следующая программа? (Попытайтесь определить это, не запуская программу.)

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

class A {
public:
    A() { cout << "Работа конструктора класса A\n"; }
    ~A() { cout << "Работа деструктора класса A\n"; }
};
```

```
class B {
public:
    B() { cout << "Работа конструктора класса B\n"; }
    ~B() { cout << "Работа деструктора класса B\n"; }
};

class C: public A, public B {
public:
    C() { cout << "Работа конструктора класса C\n"; }
    ~C() { cout << "Работа деструктора класса C\n"; }
};

int main()
{
    C ob;

    return 0;
}
```

2. Используя следующую иерархию классов, создайте конструктор класса C так, чтобы он инициализировал переменную k и передавал аргументы конструкторам A() и B().

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

class A {
    int i;
public:
    A(int a) { i = a; }.
};

class B {
    int j;
public:
    B(int a) { j = a; }
};

class C: public A, public B {
    int k;
public:
    /* Создайте такой конструктор C(), чтобы он инициализировал
переменную k и передавал аргументы конструкторам A() и B() */
};
```

7.5. Виртуальные базовые классы

При многократном прямом наследовании производным классом одного и того же базового класса может возникнуть проблема. Чтобы понять, что это за проблема, рассмотрим следующую иерархию классов:

Здесь базовый класс ***Базовый*** наследуется производными классами ***Производный1*** и ***Производный2***. Производный класс ***Производный3*** прямо наследует производные классы ***Производный1*** и ***Производный2***. Однако это подразумевает, что класс ***Базовый*** фактически наследуется классом ***Производный3*** дважды — первый раз через класс ***Производный1***, а второй через класс ***Производный2***. Однако, если член класса ***Базовый*** будет использоваться в классе ***Производный3***, это вызовет неоднозначность. Поскольку в классе ***Производный3*** имеется две копии класса ***Базовый***, то будет ли ссылка на элемент класса ***Базовый*** относиться к классу ***Базовый***, наследуемому через класс ***Производный1***, или к классу ***Базовый***, наследуемому через класс ***Производный2***? Для преодоления этой неоднозначности в C++ включен механизм, благодаря которому в классе ***Производный3*** будет включена только одна копия класса ***Базовый***. Класс, поддерживающий этот механизм, называется *виртуальным базовым классом (virtual base class)*.

В таких ситуациях, когда производный класс более одного раза косвенно наследует один и тот же базовый класс, появление двух копий базового класса в объекте производного класса можно предотвратить, если базовый класс наследуется как виртуальный для всех производных классов. Такое наследование не дает появиться двум (или более) копиям базового класса в любом следующем производном классе, косвенно наследующем базовый класс. В этом случае перед спецификатором доступа базового класса необходимо поставить ключевое слово **virtual**.

Примеры

1. В этом примере для предотвращения появления в классе **derived3** двух копий класса **base** используется виртуальный базовый класс.

```
// В этой программе используется виртуальный базовый класс
tfinclude <iostream>
using namespace std;
```

```
class base {
public:
    int i;
};

// Наследование класса base как виртуального
class derived1: virtual public base {
public:
    int j;
};

// Здесь класс base тоже наследуется как виртуальный
class derived2: virtual public base {
public:
    int k;
};

/* Здесь класс derived3 наследует как класс derived1, так и класс
derived2. Однако в классе derived3 создается только одна копия
класса base
*/
class derived3: public derived1, public derived2 {
public:
    int product() { return i * j * k; }
};

int main()
{
    derived3 ob;
    // Здесь нет неоднозначности, поскольку
    // представлена только одна копия класса base
    ob.i = 10;
    ob.j = 3;
    ob.k = 5;

    cout << "Результат равен " << ob.product() << '\n';

    return 0;
}
```

Если бы классы **derived1** и **derived2** наследовали класс **base** не как виртуальный, тогда инструкция

```
ob.i = 10;
```

вызывала бы неоднозначность и при компиляции возникла бы ошибка. (См. представленное ниже упражнение 1.)

2. Важно понимать, что даже если базовый класс наследуется производным как виртуальный, то копия этого базового класса все равно существует внутри

производного. Например, по отношению к предыдущей программе этот фрагмент совершенно правилен:

```
derived1 ob;
ob.i = 100;
```

Отличие между обычным и виртуальным базовыми классами проявляется только тогда, когда объект наследует базовый класс более одного раза. Если используются виртуальные базовые классы, то в каждом конкретном объекте присутствует копия только одного из них. В противном случае (при обычном наследовании) там было бы несколько копий.

Упражнения

1. В программе из примера 1 удалите ключевое слово **virtual** и попытайтесь откомпилировать программу. Обратите внимание на виды ошибок.
2. Объясните, зачем может понадобиться виртуальный базовый класс.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы:

1. Создайте исходный базовый класс **building** для хранения числа этажей и комнат в здании, а также общую площадь комнат. Создайте производный класс **house,** который наследует класс **building** и хранит число ванных комнат и число спален. Кроме этого создайте производный класс **office,** который наследует класс **building** и хранит число огнетушителей и телефонов. *(Замечание.* Ваше решение может отличаться от приведенного в конце книги. Однако, если функционально оно такое же, считайте его правильным.)
2. Когда базовый класс наследуется производным классом как открытый, что происходит с его открытыми членами? Что происходит с его закрытыми членами? Когда базовый класс наследуется производным классом как закрытый, что происходит с его закрытыми и открытыми членами?
3. Объясните, что означает ключевое слово **protected.** (Рассмотрите два случая: когда оно используется для задания элементов класса и когда оно используется в качестве спецификатора доступа.)
4. При наследовании одного класса другим, когда вызываются конструкторы классов? Когда вызываются их деструкторы?

5. Дан следующий фрагмент программы, впишите детали, как указано в комментариях:

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

class planet {
protected:
    double distance; // расстояние в милях от Солнца
    int revolve;     // полный оборот в днях
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth: public planet {
    double circumference; // окружность орбиты
public:
    /* Создайте конструктор earth (double d, int r). Он должен
передавать классу planet расстояние и число оборотов,
а также рассчитывать окружность орбиты (Подсказка:
окружность = 2r * 3.1416.)
*/
/* Создайте функцию show() для вывода информации на экран */
};

int main()
{
    earth ob(93000000, 365);

    ob.show();

    return 0;
}
```

6. Исправьте следующую программу:

```
/* Вариация иерархии классов из примера с классом vehicle. В
программе имеется ошибка. Найдите ее. Подсказка: попытайтесь
провести компиляции и изучите сообщения об ошибках */
#include <iostream>
using namespace std;

// Базовый класс для автомобилей разных типов
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
```

```
    }
    void showv()
    {
        cout << "Число колес" << num_wheels << '\n';
        cout << "Грузоподъемность" << range << '\n';
    }
};

enum motor {gas, electric, diesel};

class motorized: public vehicle {
    enum motor mtr;
public:
    motorized(enum motor m, int w, int r): vehicle(w, r)
    {
        mtr = m;
    }
    void showm() {
        cout << "Мотор:";
        switch(mtr) {
            case gas: cout << "На газе\n";
                break;
            case electric: cout << "На электроэнергии\n";
                break;
            case diesel: cout << "Дизельный\n";
                break;
        }
    }
};

class road_use: public vehicle {
    int passengers;
public:
    road_use(int p, int w, int r): vehicle(w, r)
    {
        passengers = p;
    }
    void showr()
        {
            cout << "Пассажироемкость" << passengers << '\n';
        }
};

enum steering {power, rack_pinion, manual};
class car: public motorized, public road_use (
    enum steering strng;
public:
    car(enum steering s, enum motor m, int w, int r, int p):
```

```
    road_use(p, w, r), motorized(m, w, r), vehicle(w, r)
    {
        strng = s;
    }
    void show() {
        showv(); showr(); showm();
        cout << "Управление:";
        switch(strng) {
            case power: cout << "Силовой привод\n";
                break;
            case rack_pinion: cout << "Механический привод\n";
                break;
            case manual: cout << "Ручной привод\n";
                break;
        }
    }
};

int main()
{
    car c(power, gas, 4, 500, 5);

    c.show();

    return 0;
}
```

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. При компиляции программы из вопроса 6 только что приведенных упражнений по проверке усвоения материала главы 7, вы могли увидеть предупреждающее сообщение (или, возможно, сообщение об ошибке), связанное с использованием инструкции **switch** внутри классов **car** и **motorised**. Почему?
2. Как вы знаете из предыдущей главы, большинство операторов, перегруженных в базовом классе, доступны для использования в производном. Для какого или для каких операторов это не так? Объясните, почему.
3. Следующей представлена переработанная версия класса **coord** из предыдущей главы. Теперь он используется в качестве базового для класса

quad, в котором помимо координат хранится номер квадранта, к которому принадлежит точка с этими координатами. Запустите программу и попытайтесь понять полученный результат.

```
/* Перегрузите операторы +, — и = относительно класса coord. Затем
используйте класс coord в качестве базового для класса quad */
#include <iostream>
using namespace std;

class coord {
public:
    int x, y; // значения координат
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator+(coord ob2);
    coord operator-(coord ob2);
    coord operator=(coord ob2);
};

// Перегрузка оператора + для класса coord
coord coord::operator+(coord ob2)
{
    coord temp;

    cout << "Использование функции operator+()\n";

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// Перегрузка оператора — относительно класса coord
coord coord::operator-(coord ob2)
{
    coord temp;

    cout << "Использование функции operator-()\n";

    temp.x = x — ob2.x;
    temp.y = y — ob2.y;

    return temp;
}

// Перегрузка оператора = относительно класса coord
coord coord::operator=(coord ob2)
```

```
{
    cout << "Использование функции operator=()\n";

    x = ob2.x;
    y = ob2.y;

    return *this; // возвращение присваиваемого объекта
}

class quad: public coord {
    int quadrant;
public:
    quad() { x = 0; y = 0; quadrant = 0; }
    quad(int x; int y): coord(x, y)
    {
        if(x>=0 && y>=0) quadrant = 1;
        else if(x<0 && y>=0) quadrant = 2;
        else if(x<0 && y<0) quadrant = 3;
        else quadrant = 4;
    }
    void showq()
    {
        cout << "Точка в квадранте: " << quadrant << '\n';
    }
    quad operator=(coord ob2);
};

quad quad::operator=(coord ob2)
{
    cout << "Использование функции operator=()\n";

    x = ob2.x;
    y = ob2.y;
    if(x>=0 && y>=0) quadrant = 1;
    else if(x<0 && y>=0) quadrant = 2;
    else if(x<0 && y<0) quadrant = 3;
    else quadrant = 4;

    return *this;
}
int main()
{
    quad o1(10, 10), o2(15, 3), o3;
    int x, y;

    o3 = o1 + o2; // сложение двух объектов
                  // вызов функции operator+()
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";
```

```
    o3 = o1 — o2; // вычитание двух объектов
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o1 - o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // присваивание объектов
    o3.get_xy(x, y);
    o3.showq();
    cout << "(o3 = o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

4. Переработайте программу из вопроса 3 так, чтобы в ней использовались дружественные оператор-функции.

Глава 8

Введение в систему ввода/вывода C++

Начиная с первой главы этой книги, создавая свои программы, мы пользовались стилем ввода/вывода C++. Теперь настало время изучить его более подробно. Как и в языке C, в C++ имеется развитая, гибкая и достаточно полная система ввода/вывода. Важно понимать, что в C++ по-прежнему поддерживается вся система ввода/вывода C. Кроме этого в C++ включен дополнительный набор объектно-ориентированных подпрограмм ввода/вывода. Главным преимуществом системы ввода/вывода C++ является то, что она может перегружаться для создаваемых вами классов. Это отличие позволяет легко встраивать в систему ввода/вывода C++ новые создаваемые вами типы данных.

Как и в C, в системе объектно-ориентированного ввода/вывода C++ имеется незначительная разница между консольным и файловым вводом/выводом. На самом деле, консольный и файловый ввод/вывод — это просто разный взгляд на один и тот же механизм. В этой главе в примерах используется ввод/вывод на консоль (в данном случае на экран монитора), но представленная информация вполне применима и для ввода/вывода в файл (ввод/вывод в файл более детально исследуется в главе 9).

К моменту написания этой книги использовались две версии библиотеки ввода/вывода C++: старая, основанная на изначальной спецификации C++, и новая, определенная единым международным стандартом Standard C++. С точки зрения программиста для решения подавляющего большинства задач обе эти библиотеки идентичны. Так происходит потому, что новая библиотека ввода/вывода — это по существу просто обновленная и усовершенствованная версия старой библиотеки. Фактически, почти все отличия двух версий скрыты от вас, поскольку касаются не способа использования библиотек, а способа их реализации. Для программиста главное отличие заключается в том, что новая библиотека ввода/вывода C++ имеет несколько дополнительных возможностей и определяет несколько новых типов данных. Таким образом, новая библиотека ввода/вывода — это по существу просто несколько улучшенная старая. Почти все уже написанные для старой библиотеки программы при использовании новой будут компилироваться без каких бы то ни было существенных изменений. Поскольку прежняя библиотека ввода/вывода ныне считается устаревшей, данная книга описывает только

новую библиотеку, как это определено стандартом Standard C++. Тем не менее, большая часть информации вполне применима и к старой библиотеке ввода/вывода.

Эта глава охватывает несколько аспектов системы ввода/вывода C++, включая форматируемый ввод/вывод, манипуляторы ввода/вывода и создание пользовательских функций ввода/вывода. Как вы увидите в дальнейшем, в системе ввода/вывода C++ имеется множество черт, характерных для системы ввода/вывода C.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Создайте иерархию классов для хранения информации о летательных аппаратах. Начните с общего базового класса **airship,** предназначенного для хранения количества перевозимых пассажиров и количества перевозимого груза (в фунтах). Затем создайте два производных от **airship** класса **airplane и balloon.** Класс **airplane** должен хранить тип самолетного двигателя (винтовой или реактивный) и дальность полета в милях. Класс **balloon** должен хранить информацию о типе газа, используемого для подъема дирижабля (водород или гелий), и его максимальный потолок (в футах). Создайте короткую программу для демонстрации работы этой иерархии классов. (Ваше решение несомненно будет несколько отличаться от ответа, приведенного в конце книги. Однако, если функционально оно будет соответствовать ответу, считайте его правильным.)
2. Для чего используется ключевое слово **protected?**
3. Дана следующая иерархия классов. В каком порядке вызываются конструкторы? В каком порядке вызываются деструкторы?

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

class A {
public:
    A() { cout << "Работа конструктора класса A\n"; }
    ~A() { cout << "Работа деструктора класса A\n"; }
};

class B: public A {
public:
    B() { cout << "Работа конструктора класса B\n"; }
    ~B() { cout << "Работа деструктора класса B\n"; }
};
```

```
class C: public B {
public:
    C() { cout << "Работа конструктора класса C\n"; }
    ~C() { cout << "Работа деструктора класса C\n"; }
};

int main()
{
    C ob;

    return 0;
}
```

4. Дан следующий фрагмент кода. В каком порядке вызываются конструкторы и деструкторы?

```
class myclass: public A, public B, public C { ...
```

5. Добавьте отсутствующие конструкторы в данную программу:

```
ttinclude<iostream>
using namespace std;

class base {
    int i, j;
public:
    // требуется конструктор
    void showij() { cout << i << ' ' << j << '\n'; }
};

class derived: public base {
    int k;
public:
    // требуется конструктор
    void show() { cout << k << ' '; showij(); }
};

int main()
{
    derived ob(1, 2, 3);

    ob.show();

    return 0;
}
```

6. Обычно, если вы задаете иерархию классов, вы начинаете с более _ класса и продолжаете более _ классом. (Впишите пропущенные слова.)

8.1. Некоторые базовые положения системы ввода/вывода C++

Перед тем как начать обсуждение системы ввода/вывода C++, несколько общих комментариев. Система ввода/вывода C++, так же, как система ввода/вывода C, действует через *потоки (streams)*. Поскольку вы программировали на C, вы уже должны знать, что такое поток ввода/вывода, однако несколько дополнительных замечаний обобщат ваши знания. Поток ввода/вывода — это логическое устройство, которое выдает и принимает пользовательскую информацию. Поток связан с физическим устройством с помощью системы ввода/вывода C++. Поскольку все потоки ввода/вывода действуют одинаково, то, несмотря на то, что программисту приходится работать с совершенно разными по характеристикам устройствами, система ввода/вывода предоставляет для этого единый удобный интерфейс. Например, функция, которая используется для записи информации на экран монитора, вполне подойдет как для записи в файл, так и для вывода на принтер.

Как вы знаете, если программа на C начинает выполняться, открываются три потока: **stdin,** stdout и **stderr**. Нечто похожее имеет место при запуске программ на C++. Когда запускается программа на C++, автоматически открываются четыре потока:

Поток	Значение	Устройство по умолчанию
cin	Стандартный ввод	Клавиатура
cout	Стандартный вывод	Экран
cerr	Стандартная ошибка	Экран
clog	Буферизуемая версия cerr	Экран

Как вы, наверное, уже догадались, потоки **cin, cout и cerr** соответствуют потокам **stdin, stdout и stderr** языка C. Потоками **cin** и **cout** вы уже пользовались. Поток **clog** — это просто буферизуемая версия потока **cerr**. В языке Standard C++ также открываются дополнительные потоки **wcin**, wcout, **wcerr** и **wclog**, предназначенные для широких (16-разрядных) символов, которые в данной книге не рассматриваются. Эти потоки обеспечивают передачу расширенных наборов символов (large character sets), что обеспечивает возможность работы с некоторыми экзотическими языками, такими как, например, китайский.

По умолчанию, стандартные потоки используются для связи с клавиатурой и экраном. Однако в среде, в которой поддерживается переопределение ввода/вывода, эти потоки могут быть перенаправлены на другие устройства.

Как отмечалось в главе 1, в C++ ввод/вывод обеспечивается подключением к программе заголовочного файла **<iostream>**. В этом файле определены сложные наборы иерархий классов, поддерживающие операции ввода/вывода. Классы ввода/вывода начинаются с системы *классов-шаблонов (template classes)*.

Подробно о **классах-шаблонах,** называемых также родовыми классами (generic classes), будет рассказано в главе 11, сейчас же ограничимся кратким комментарием. В классе-шаблоне определяется только форма класса без полного задания данных, с которыми он работает. После того как класс-шаблон определен, появляется возможность создавать отдельные экземпляры этого класса. Что касается библиотеки ввода/вывода, то Standard C++ создает две разные версии классов-шаблонов ввода/вывода: одну для 8-разрядных символов, а другую для широких (16-разрядных) символов. В данной книге рассказывается только о классах для 8-разрядных символов, поскольку именно они используются чаще всего.

Система ввода/вывода C++ строится на двух связанных, но различных иерархиях классов-шаблонов. Первая является производной от класса нижнего уровня **basic_streambuf**. Этот класс предоставляет базу для операций нижнего уровня по вводу и выводу, а также обеспечивает надлежащую поддержку всей системы ввода/вывода C++. До тех пор, пока вы не погрузитесь в самые основы программирования ввода/вывода, непосредственно использовать класс basic_streambuf вам не понадобиться. Иерархия классов, с которой вам чаще всего придется иметь дело, является производной от класса **basic_ios.** Это класс ввода/вывода верхнего уровня, который обеспечивает форматирование, контроль ошибок и информацию о состоянии потока ввода/вывода. Класс basic_ios является базовым для нескольких производных классов, среди которых классы **basicjstream, basic_ostream и basic_iostream**. Эти классы используются соответственно для создания потоков ввода, вывода и ввода/вывода.

Как уже говорилось, библиотека ввода/вывода создает две отдельные версии иерархий классов: одну для 8-разрядных символов и другую для широких символов. В представленной ниже таблице показано соответствие имен классов-шаблонов их версиям для 8-разрядных символов (включая и те, о которых будет рассказано в главе 9).

Класс-шаблон	Класс для 8-разрядных символов
basic_streambuf	streambuf
basic_ios	ios
basicjstream	istream
basic_ostream	ostream
basic_iostream	iostream
basicjstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

Имена классов для 8-разрядных символов будут употребляться далее на всем протяжении книги, поскольку как раз эти имена и следует указывать в

программах. Это именно те имена, которые использовались в прежней библиотеке ввода/вывода, и именно по этой причине на уровне исходного кода совместимы старая и новая библиотеки ввода/вывода.

И последнее замечание: в классе **ios** содержится множество функций и переменных — членов класса, которые контролируют или отображают основные операции потока ввода/вывода. Вы еще часто будете сталкиваться с классом **ios.** Запомните: чтобы получить доступ к этому важному классу, необходимо включить в программу заголовок **<iostream>**.

8.2. Форматируемый ввод/вывод

До сих пор во всех примерах этой книги для вывода информации на экран использовались форматы, заданные в C++ по умолчанию. Однако информацию можно выводить в широком диапазоне форм. При этом с помощью системы ввода/вывода C++ можно форматировать данные так же, как это делала в C функция **printf()**. Кроме того, можно изменять определенные параметры ввода информации.

Каждый поток ввода/вывода связан с набором флагов формата (format flags), которые управляют способом форматирования информации и представляют собой битовые маски (bitmasks). Эти маски объявлены в классе **ios** как данные перечислимого типа **fintflags,** в котором определены следующие значения:

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

Эти значения определены в классе ios и необходимы для установки или сброса флагов формата. Если вы пользуетесь устаревшим, нестандартным компилятором, может оказаться, что перечислимый тип данных **fintflags** в нем не определен. В таких компиляторах для хранения флагов формата отводится длинное целое.

Когда при вводе информации в поток установлен флаг **skipws,** начальные невидимые символы (пробелы, табуляции и символы новой строки) отбрасываются. Когда флаг **skipws** сброшен, невидимые символы не отбрасываются.

Когда установлен флаг **left,** происходит выравнивание вывода по левому краю. Когда установлен флаг **right,** происходит выравнивание вывода по правому краю. Когда установлен флаг **internal,** для заполнения поля вывода происходит вставка пробелов между всеми цифрами и знаками числа. Если все эти флаги не установлены, то по умолчанию используется выравнивание по правому краю.

По умолчанию числовые значения выводятся в десятичной системе счисления. Однако основание системы счисления можно поменять. Установка флага **oct** ведет к тому, что вывод будет осуществляться в восьмеричной системе счисления, а установка флага hex — в шестнадцатеричной. Чтобы вернуться к десятичной системе счисления, установите флаг **dec.**

Установка флага **showbase** ведет к выводу основания системы счисления. Например, шестнадцатеричное значение 1F с этим флагом будет выводиться как 0x1F.

По умолчанию при выводе значений в научной нотации символ "e" выводится в нижнем регистре. Кроме этого, при выводе шестнадцатеричного значения символ "x" тоже выводится в нижнем регистре. При установке флага **uppercase,** эти символы выводятся в верхнем регистре.

Установка флага **showpos** приводит к выводу знака + перед положительными значениями.

Установка флага **showpoint** ведет к появлению десятичной точки и последующих нулей при выводе любых значений с плавающей точкой.

При установке флага **scientific** числа с плавающей точкой выводятся в научной нотации. При установке флага **fixed** числа с плавающей точкой выводятся в обычной нотации. Если ни один из этих флагов не установлен, компилятор сам выбирает подходящий способ вывода.

Если установлен флаг **unitbuf**, то буфер очищается (flush) после каждой операции вставки (insertion operation).

При установленном флаге **boolalpha** значения булева типа выводятся в виде ключевых слов **true и false.**

Одновременно на все поля, определенные с флагами **oct,** dec и hex, можно сослаться с помощью флага **basefield**. Аналогично на поля, определенные с флагами **left, right и internal,** можно сослаться с помощью флага **adjustfield**. И наконец, на поля с флагами **scientific и fixed** можно сослаться с помощью флага **floatfield**.

Для установки флага формата пользуйтесь функцией **setf()**. Эта функция является членом класса **ios.** Здесь показана ее основная форма:

```
fmtflags setf(fmtflags флаги);
```

Эта функция возвращает предыдущие установки флагов формата и устанавливает новые, заданные значением ***флаги.*** (Значения всех остальных флагов не изменяются.) Например, для установки флага **showpos** можно воспользоваться следующей инструкцией:

```
поток_ввода/вывода.setf(ios::showpos);
```

Здесь ***поток ввода/вывода*** — это тот поток, на который вы хотите повлиять. Обратите внимание на использование оператора расширения области види-

мости. Запомните, флаг showpos — это перечислимая константа внутри класса ios. Следовательно, чтобы сообщить компилятору об этом, необходимо поставить перед флагом showpos имя класса и оператор расширения области видимости. Если этого не сделать, константа showpos просто не будет распознана компилятором.

Важно понимать, что функция **setf()** является членом класса ios и влияет на созданные этим классом потоки ввода/вывода. Поэтому любой вызов функции **setf()** делается относительно конкретного потока. Нельзя вызвать функцию **setf()** саму по себе. Другими словами, в C++ нет понятия глобального состояния формата. Каждый поток ввода/вывода поддерживает собственную информацию о состоянии формата.

Вместо повторных вызовов функции **setf()** в одном вызове можно установить сразу несколько флагов. Для объединения необходимых флагов используйте оператор OR. Например, в следующем вызове функции **setf()** для потока cout устанавливаются флаги showbase и **hex**:

```
cout.setf(ios::showbase I ios::hex);
```

Поскольку флаги формата определяются внутри класса ios, доступ к ним должен осуществляться через класс ios и оператор расширения области видимости. Например, сам по себе флаг ***showbase*** *задать нельзя, необходимо написать* ***ios::showbase****.*

Дополнением **setf()** является функция **unsetf()**. Эта функция-член класса ios сбрасывает один или несколько флагов формата. Здесь показана ее основная форма:

```
void unsetf(fmtflags флаги);
```

Флаги, заданные параметром *флаги,* сбрасываются. (Все остальные флаги остаются без изменений.)

Когда-нибудь вам понадобится только узнать текущее состояние флагов и при этом ничего не менять. Поскольку функции **setf()** и **unsetf()** меняют состояние одного или более флагов, в класс ios включена функция-член flags(), которая просто возвращает текущее состояние флагов формата. Здесь показан прототип этой функции:

```
fmtflags flags();
```

Функция flagsQ имеет и вторую форму, которая позволяет установить *все,* связанные с потоком ввода/вывода, флаги формата. Флаги задаются в аргументе функции flagsQ. Здесь показан прототип этой версии функции:

```
fmtflags flags(fmtflags f);
```

При использовании этой версии функции **flags()** битовый шаблон *f* копируется в переменную для хранения связанных с потоком флагов формата; при этом перезаписывается весь предшествующий набор флагов. Функция возвращает предыдущие установки флагов формата.

1. В этом примере показано, как установить несколько флагов формата.

```
tinclude <iostream>
using namespace std;

int main()
{
    // вывод с использованием установок по умолчанию
    cout << 123.23 << " привет " << 100 << '\n';
    cout << 10 << '  ' << -10 << '\n';
    cout << 100.0 << '\n';

    // теперь меняем формат
    cout.unsetf(ios::dec); // требуется не для всех компиляторов
    cout.setf(ios::hex | ios::scientific);
    cout << 123.23 << " привет " << 100 << '\n';

    cout.setf(ios::showpos);
    cout << 10 << '  ' << -10 << '\n';

    cout.setf(ios::showpoint | ios::fixed);
    cout << 100.0;

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
123.23 привет 100
10 -10
100

1.232300e+02 привет 64
a fffffff6
+100.000000
```

Обратите внимание, что флаг **showpos** влияет только на вывод десятичных значений. Он не влияет на число 10, когда оно выводится в шестнадцатеричной системе счисления. Кроме того, отметьте, что вызов функции **unsetf()** приводит к сбросу установленного по умолчанию флага **dec.** Этот вызов нужен не для всех компиляторов, а только для некоторых, для которых уста-

новка флага **dec** автоматически приводит к сбросу остальных флагов. Поэтому после сброса флага **dec** необходимо заново установить флаг **hex** или **oct.** Как правило, для лучшей переносимости программ лучше установить только то основание системы счисления, которое вы будете использовать и стереть остальные.

2. В следующей программе показано действие флага **uppercase.** В первую очередь устанавливаются флаги **uppercase, showbase** и **hex.** Затем выводится число 88 в шестнадцатеричной системе счисления. В этом случае символ шестнадцатеричной системы счисления "X" выводится в верхнем регистре. Далее с помощью функции **unsetf()** сбрасывается флаг **uppercase** и снова выводится шестнадцатеричное число 88. Теперь символ "x" оказывается в нижнем регистре.

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

int main()
{
    cout.unsetf(ios::dec);
    cout.setf(ios::uppercase | ios::showbase | ios::hex);

    cout << 88 << '\n';

    cout.unsetf(ios::uppercase);

    cout << 88 << '\n';

    return 0;
}
```

3. В следующей программе для вывода состояния флагов формата потока **cout** используется функция **flags().** Обратите особое внимание на функцию **showflags()**. Она может вам пригодиться при разработке собственных программ.

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

void showflags();

int main()
{
    // отображение состояния флагов формата по умолчанию
    showflags();

    cout.setf(ios::oct | ios::showbase | ios::fixed);

    showflags();

    return 0;
}
```

```
// Эта функция выводит состояние флагов формата
void showflags()
{
    ios::fmtflags f;

    f = cout.flags(); // получение установок флагов формата

    if(f & ios::skipws) cout << "skipws установлен\n";
    else cout << "skipws сброшен\n";

    if(f & ios::left) cout << "left установлен\n";
    else cout << "left сброшен\n";

    if(f & ios::right) cout << "right установлен\n";
    else cout << "right сброшен\n";

    if(f & ios::internal) cout << "internal установлен\n";
    else cout << "internal сброшен\n";

    if(f & ios::dec) cout << "dec установлен\n";
    else cout << "dec сброшен\n";

    if(f & ios::oct) cout << "oct установлен\n";
    else cout << "oct сброшен\n";

    if(f & ios::hex) cout << "hex установлен\n";
    else cout << "hex сброшен\n";

    if(f & ios::showbase) cout << "showbase установлен\n";
    else cout << "showbase сброшен\n";

    if(f & ios::showpoint) cout << "showpoint установлен\n";
    else cout << "showpoint сброшен\n";

    if(f & ios::showpos) cout << "showpos установлен\n";
    else cout << "showpos сброшен\n";

    if(f & ios::uppercase) cout << "uppercase установлен\n";
    else cout << "uppercase сброшен\n";

    if(f & ios::scientific) cout << "scientific установлен\n";
    else cout << "scientific сброшен\n";

    if(f & ios::fixed) cout << "fixed установлен\n";
    else cout << "fixed сброшен\n";

    if(f & ios::unitbuf) cout << "unitbuf установлен\n";
    else cout << "unitbuf сброшен\n";

    if(f & ios::boolalpha) cout << "boolalpha установлен\n";
    else cout << "boolalpha сброшен\n";

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

В функции **showflags()** объявляется локальная переменная f типа **fmtflags**. Если в вашем компиляторе тип данных **fmtflags** не определен, объявите переменную f типа **long.** Ниже показан результат выполнения программы:

```
skipws установлен
left сброшен
right сброшен
internal сброшен
dec установлен
oct сброшен
hex сброшен
showbase сброшен
showpoint сброшен
showpos сброшен
uppercase сброшен
scientific сброшен
fixed сброшен
unitbuf сброшен
boolalpha сброшен

skipws установлен
left сброшен
right сброшен
internal сброшен
dec установлен
oct установлен
hex сброшен
showbase установлен
showpoint сброшен
showpos сброшен
uppercase сброшен
scientific сброшен
fixed установлен
unitbuf сброшен
boolalpha сброшен
```

4. В следующей программе проиллюстрирована работа второй версии функции **flagsQ.** Сначала, устанавливая флаги **showpos, showbase, oct и right,** мы строим маску флагов. Затем с помощью функции **flags()** для потока **cout** маска связывается с переменной флагов. С помощью функции **showflags()** проверяется правильность установки флагов. (Это та же функция, которая использовалась в предыдущей программе.)

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

void showflags();
```

```
int main()
{
    // отображение состояния флагов формата по умолчанию
    showflags();

    // устанавливаются флаги showpos, showbase, oct и right;
    // остальные сбрасываются
    ios::fmtflags f = ios::showpos I ios::showbase |
                      ios:: oct | ios::right;

    cout.flags(f); // установка флагов

    showflags();

    return 0;
}
```

1. Напишите программу, которая бы устанавливала флаги для потока **cout** так, чтобы целые, если они положительны, выводились со знаком +. Покажите, что ваш набор флагов формата правилен.
2. Напишите программу, которая бы устанавливала флаги для потока **cout** так, чтобы всегда при выводе дробных значений были показаны десятичные точки. Кроме этого, значения с плавающей точкой должны выводиться в научной нотации с символом "E" в верхнем регистре.
3. Напишите программу, которая сохраняет текущее состояние флагов формата, устанавливает флаги **showbase** и **hex,** выводит на экран значение 100, а затем возвращает флаги в исходное состояние.

8.3. Функции *width()*, *precision()* и *fill()*

Кроме флагов формата в классе **ios** определены три функции-члена. Эти функции устанавливают следующие параметры формата: ширину поля, точность и символ заполнения. Этими функциями являются соответственно функции **width()**, **precision()** и **fill()**.

По умолчанию при выводе любого значения оно занимает столько позиций, сколько символов выводится. Однако с помощью функции **width()** можно задать минимальную ширину поля. Ниже показан прототип этой функции:

```
streamsize width(streamsize w);
```

Ширина поля задается параметром ***w***, а функция возвращает предыдущую ширину поля. Тип данных **streamsize** определен в заголовочном файле

<iostream> как одна из форм целого. В некоторых компиляторах при выполнении каждой операции вывода значение ширины поля возвращается к своему состоянию по умолчанию, поэтому перед каждой инструкцией вывода может понадобиться устанавливать минимальную ширину поля.

После установки минимальной ширины поля, если выводимое значение требует поле, меньшее заданной ширины, остаток поля заполняется текущим символом заполнения (по умолчанию пробелом) так, чтобы была занята вся ширина поля. Однако запомните, если размер выводимого значения превосходит минимальную ширину поля, будет занято столько символов, сколько нужно. Выводимое значение не усекается.

По умолчанию при выводе значений с плавающей точкой точность равна шести цифрам. Однако с помощью функции **precision()** это число можно изменить. Ниже показан прототип функции **precision()**:

```
streamsize precision(streamsize p);
```

Точность (число выводимых цифр после запятой) задается параметром *p, а* возвращает функция прежнее значение точности.

По умолчанию, если требуется заполнить свободные поля, используются пробелы. Однако с помощью функции **fillQ** можно изменить символ заполнения. Ниже показан прототип функции **fillQ:**

```
char fill(char ch);
```

После вызова функции **fill()** символ *ch* становится новым символом заполнения, а функция возвращает прежнее значение символа заполнения.

Примеры

1. В этом примере показана программа, которая иллюстрирует работу функций формата:

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

int main()
{
    cout.width(10); // установка минимальной ширины поля
    cout << "Привет" << '\n'; // по умолчанию выравнивание вправо
    cout.fill('%'); // установка символа заполнения
    cout.width(10); // установка ширины поля
    cout << "Привет" << '\n'; // по умолчанию выравнивание вправо
    cout.setf(ios::left); // выравнивание влево
    cout.width(10); // установка ширины поля
    cout << "Привет" << '\n'; // выравнивание влево
```

```
    cout.width(10); // установка ширины поля
    cout.precision(10); // установка точности в 10 цифр
    cout << 123.234567 << '\n';
    cout.width(10); // установка ширины поля
    cout.precision(6); // установка точности в 6 цифр
    cout << 123.234567 << '\n';

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
Привет
%%%%Привет
Привет%%%%
123.234567
123.235%%%
```

Обратите внимание, что ширина поля устанавливается перед каждой инструкцией вывода.

2. В следующей программе показано, как с помощью функций установки флагов формата ввода/вывода C++ создать выровненную таблицу чисел:

```
// Создание таблицы квадратных корней и квадратов
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double x;

    cout.precision(4);
    cout << "x     sqrt(x)     x^2\n\n";

    for(x = 2.0; x <= 20.0; x++) {
        cout.width(7);
        cout << x << " ";
        cout.width(7);
        cout << sqrt(x) << " ";
        cout.width(7);
        cout << x*x << "\n";
    }

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
x      sqrt(x)     x^2
2       1.414       4
3       1.732       9
```

```
 4       . 2     16
 5     2.236     25
 6     2.449     36
 7     2.646     49
 8     2.828     64
 9         3     81
10     3.162    100
11     3.317    121
12     3.464    144
13     3.606    169
14     3.742    196
15     3.873    225
16         4    256
17     4.123    289
18     4.243    324
19     4.359    361
20     4.472    400
```

1. Разработайте программу для печати таблицы натуральных и десятичных логарифмов чисел от 2 до 100. Формат таблицы следующий: правое выравнивание, ширина поля — 10 символов, точность — 5 десятичных позиций.
2. Создайте функцию **center()** со следующим прототипом:

   ```
   void center(char *s);
   ```

   Эта функция должна устанавливать заданную строку в центр экрана. Для реализации этой задачи воспользуйтесь функцией **width()**. Предполагается, что ширина экрана равна 80 символам. (Для простоты считайте, что длина строки не превышает 80 символов.) Напишите программу, иллюстрирующую работу этой функции.
3. Поэкспериментируйте с флагами и функциями формата. После того как вы ближе познакомитесь с системой ввода/вывода C++, вы никогда не ошибетесь в выборе нужного формата вывода.

8.4. Манипуляторы ввода/вывода

В системе ввода/вывода C++ имеется еще один способ форматирования информации. Этот способ подразумевает использование специальных функций — *манипуляторов ввода/вывода (I/O manipulators)*. Как вы увидите далее, манипуляторы ввода/вывода являются, в некоторых ситуациях, более удобными, чем флаги и функции формата класса **ios.**

Манипуляторы ввода/вывода являются специальными функциями формата ввода/вывода, которые, в отличие от функций — членов класса **ios,** могут располагаться *внутри* инструкций ввода/вывода. Стандартные манипуляторы показаны в табл. 8.1. Как можно заметить при изучении таблицы, значительная часть манипуляторов ввода/вывода по своим действиям аналогична соответствующим функциям — членам класса **ios.** Многие манипуляторы, представленные в табл. 8.1, стали частью языка совсем недавно, после появления стандарта Standard C++, и поддерживаются только современными компиляторами.

Для доступа к манипуляторам с параметрами (таким, как функция **setw()**), необходимо включить в программу заголовок **<iomanip>**. **В** этом заголовке нет необходимости при использовании манипуляторов без параметров.

Как уже установлено, манипуляторы можно задавать внутри цепочки операций ввода/вывода. Например:

```
cout << oct << 100 << hex << 100;
cout << setw(10) << 100;
```

Первая инструкция сообщает потоку **cout** о необходимости вывода целых в восьмеричной системе счисления и выводит число 100 в восьмеричной системе счисления. Затем она сообщает потоку ввода/вывода о необходимости вывода целых в шестнадцатеричной системе счисления и далее осуществляется вывод числа 100 уже в шестнадцатеричном формате. Во второй инструкции устанавливается ширина поля равная 10, и затем снова выводится 100 в шестнадцатеричном формате. Обратите внимание, если используется манипулятор без аргументов (в данном примере им является манипулятор **oct),** скобки за ним не ставятся, поскольку это на самом деле адрес манипулятора, передаваемый перегруженному оператору <<.

Таблица 8.1. *Манипуляторы ввода/вывода языка Standard C++*

Манипулятор	Назначение	Ввод/Вывод
boolalpha	Установка флага boolalpha	Ввод/Вывод
dec	Установка флага dec	Ввод/Вывод
endl	Вывод символа новой строки и очистка потока	Вывод
ends	Вывод значения NULL	Вывод
fixed	Установка флага fixed	Вывод
flush	Очистка потока	Вывод
hex	Установка флага hex	Ввод/Вывод
internal	Установка флага internal	Вывод
left	Установка флага left	Вывод
noboolalpha	Сброс флага boolalpha	Ввод/Вывод

***Таблица 8.1** (продолжение)*

Манипулятор	Назначение	Ввод/Вывод
noshowbase	Сброс флага showbase	Вывод
noshowpoint	Сброс флага showpoint	Вывод
noshowpos	Сброс флага showpos	Вывод
noskipws	Сброс флага skipws	Ввод
nounitbuf	Сброс флага unitbuf	Вывод
nouppercase	Сброс флага uppercase	Вывод
oct	Установка флага oct	Ввод/Вывод
resetiosflags(fmtflags *f*)	Сброс флагов, заданных параметром *f*	Ввод/Вывод
right	Установка флага right	Вывод
scientific	Установка флага scientific	Вывод
setbase (int *основание)*	Задание *основания* системы счисления	Ввод/Вывод
setfill(int*ch)*	Задание символа заполнения *ch*	Вывод
setiosflags(fmtflags *f*)	Установка флагов, заданных параметром *f*	Ввод/Вывод
setprecision(int p)	Задание числа цифр точности равным *p*	Вывод
setw(int *w)*	Задание ширины поля равным *w позиций*	Вывод
showbase	Установка флага showbase	Вывод
showpoint	Установка флага showpoint	Вывод
showpos	Установка флага showpos	Вывод
skipws	Установка флага skipws	Ввод
unitbuf	Установка флага unitbuf	Вывод
uppercase	Установка флага uppercase	Вывод
ws	Пропуск начальных пробелов	Ввод

Запомните, что манипулятор ввода/вывода влияет только на поток, частью которого является выражение ввода/вывода, содержащего манипулятор. Манипуляторы ввода/вывода *не* влияют на все, открытые в данный момент, потоки.

Как отмечалось в предыдущем примере, главным преимуществом манипуляторов по сравнению с функциями — членами класса **ios** является то, что манипуляторы обычно удобнее, так как позволяют писать более компактные программы.

Если вы с помощью манипулятора хотите установить конкретные флаги формата, используйте функцию **setiosflags()**. Этот манипулятор реализует ту же функцию, что и функция-член **setf()**. Для сброса флагов формата используйте манипулятор **resetiosflags()**. Этот манипулятор эквивалентен функции **unsetf()**.

Примеры

1. В этой программе представлено несколько манипуляторов ввода/вывода:

```
ttinclude<iostream>
ttinclude<iomanip>
using namespace std;

int main()
{
    cout << hex << 100 << endl;
    cout << oct << 10 << endl;

    cout << setfill('X') << setw(10);
    cout << 100 << " привет " << endl;

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
64
12
XXXXXXX144 привет
```

2. Здесь представлена другая версия программы, в которой на экран выводится таблица квадратов и квадратных корней чисел от 2 до 20. В этой версии вместо функций-членов и флагов формата используются манипуляторы ввода/вывода.

```
/* В этой версии для вывода таблицы квадратов и квадратных корней
используются манипуляторы
*/
ttinclude<iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double x;

    cout << setprecision(4);
    cout << "x      sqrt(x)      x^2\n\n";

    for(x = 2.0; x <= 20.0; x++) {
        cout << setw(7) << x << "  ";
        cout << setw(7) << sqrt(x) << "  ";
        cout << setw(7) << x*x << '\n';
    }
```

```
    return 0;
}
```

3. Одним из самых интересных флагов формата новой библиотеки ввода/вывода является флаг **boolalpha**. Этот флаг можно установить либо непосредственно, либо с помощью манипулятора **boolalpha.** Интересным этот флаг делает то, что он позволяет реализовать ввод и вывод значений булева типа, т. е. ключевых слов **true и false,** вместо которых раньше обычно приходилось использовать соответственно 1 для истинного значения и 0 для ложного.

```
// Использование флага формата boolalpha
#include <iostream>
using namespace std;

int main()
{
    bool b;

    cout << "Перед установкой флага формата boolalpha: ";
    b = true;
    cout << b << " ";
    b = false;
    cout << b << endl;

    cout << "После установки флага формата boolalpha: ";
    b = true;
    cout << boolalpha << b << " ";
    b = false;
    cout << b << endl;

    cout << "Введите значение булева типа: ";
    cin >> boolalpha >> b; // здесь можно ввести true или false
    cout << "Введенное значение: " << b;

    return 0;
}
```

Примерный результат работы программы:

```
Перед установкой флага формата boolalpha: 1 O
После установки флага формата boolalpha: true false
Введите значение булева типа: true
Введенное значение: true
```

Как видите, после установки флага формата **boolalpha**, для обозначения вводимых и выводимых значений булева типа используются ключевые слова **true и false.** Отметьте, что флаг формата **boolalpha** необходимо устанавливать отдельно для потока cin и отдельно для потока **cout.** Как и в случае с другими флагами формата, установка флага **boolalpha** для одного потока вовсе не подразумевает его автоматической установки для другого потока.

1. Выполните еще раз упражнения 1 и 2 из раздела 8.3, только теперь, вместо функций-членов и флагов формата, используйте манипуляторы ввода/вывода.
2. Составьте инструкцию для вывода числа 100 в шестнадцатеричной системе счисления с отображением основания системы счисления (Ox). Для выполнения задания воспользуйтесь манипулятором **setiosflags()**.
3. Объясните, что дает установка флага **boolalpha**.

8.5. Пользовательские функции вывода

Как уже отмечалось в этой книге, одним из доводов в пользу использования операторов ввода/вывода C++, вместо аналогичных им функций ввода/вывода C, является возможность перегрузки операторов ввода/вывода для создаваемых вами классов. В этом разделе вы узнаете, как перегрузить оператор вывода <<.

В языке C++ вывод иногда называется *вставкой (insertion), а* оператор << — *оператором вставки (insertion operator).* Когда вы для вывода информации перегружаете оператор <<, вы создаете *функцию вставки (inserter function* или *inserter).* Рациональность этим терминам дает то, что оператор вывода *вставляет (inserts)* информацию в поток. Во избежание путаницы мы будем называть функцию вставки пользовательской функцией вывода.

У всех пользовательских функций вывода следующая основная форма:

```
ostream &operator<< (ostream &stream, имя_класса объект)
{
    // тело пользовательской функции вывода
    return stream;
}
```

Первый параметр является ссылкой на объект типа **ostream.** Это означает, что поток ***stream*** должен быть потоком вывода. (Запомните, класс **ostream** является производным от класса ios.) Второй параметр получает выводимый объект. (Он, если для вашего приложения это нужно, тоже может быть параметром-ссылкой). Обратите внимание, что пользовательская функция вывода возвращает ссылку на поток ***stream,*** который имеет тип **ostream.** Это необходимо, если перегруженный оператор << должен использоваться в ряде последовательных выражений ввода/вывода:

```
cout << ob1 << ob2 << ob3;
```

Внутри пользовательской функции вывода можно выполнить любую процедуру. То, что будет делать эта функция, полностью зависит от вас. Однако

в соответствии с хорошим стилем программирования, следует ограничить задачи пользовательской функции вывода только вставкой информации в поток.

Хотя это на первый взгляд может показаться странным, но пользовательская функция вывода *не* может быть членом класса, для работы с которым она создана. И вот почему. Если оператор-функция любого типа является членом класса, то левый операнд, который неявно передается через указатель **this,** является объектом, генерирующим вызов оператор-функции. Это подразумевает, что левый операнд является объектом этого класса. Поэтому, если перегруженная оператор-функция является членом класса, тогда левый операнд должен быть объектом этого класса. Однако, когда вы создаете пользовательскую функцию вывода, левый операнд становится потоком, а не объектом класса, а правый операнд — объектом, который нужно вывести. Именно поэтому пользовательская функция вывода не может быть функцией-членом.

То, что пользовательская функция вывода не может быть функцией-членом на первый взгляд кажется серьезным изъяном C++, поскольку подразумевает, что все данные класса, выводимые в поток через эту функцию, должны быть открытыми, нарушая тем самым ключевой принцип инкапсуляции. Однако это не так. Несмотря на то, что пользовательские функции вывода не могут быть членами класса, для работы с которым они разработаны, они могут быть дружественными классу. В подавляющем большинстве реальных ситуаций, с которыми вам придется столкнуться при программировании ввода/вывода, перегружаемая пользовательская функция вывода будет дружественной классу, для которого она создана.

1. Для начала рассмотрим простой пример, в котором для класса **coord,** разработанного в предыдущей главе, создается пользовательская функция вывода:

```
// Использование дружественной функции вывода
// для объектов типа coord
#include <iostream>
using namespace std;

class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
};
```

```
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);

    cout << a << b;

    return 0;
}
```

В результате выполнения программы на экран выводится следующее:

```
1, 1
10, 23
```

Пользовательская функция вывода этой программы иллюстрирует одну очень важную для создания ваших собственных функций особенность: их нужно разрабатывать, возможно, более обобщенными. В данном конкретном случае инструкция ввода/вывода внутри функции вставляет значения x и y в поток **stream,** который и является передаваемым в функцию потоком. Как вы увидите в следующей главе, та же самая пользовательская функция вывода, которая в нашем примере используется для вывода информации на экран, может использоваться и для ее вывода в *любой* поток. Тем не менее, начинающие программисты иногда пишут пользовательскую функцию вывода для класса **coord** следующим образом:

```
ostream &operator<<(ostream &stream, coord ob)
{
    cout << ob.x << ", " << ob.y << '\n';
    return stream;
}
```

В этом случае выражение жестко запрограммировано на вывод информации на стандартное устройство вывода, связанное с классом **cout.** Это ведет к тому, что другие потоки не могут воспользоваться вашей функцией. Вывод очевиден, следует делать свои функции, возможно, более обобщенными, поскольку это никогда не повредит, а иногда может оказаться полезным.

2. В следующей версии предыдущей программы, пользовательская функция вывода *не* является дружественной классу **coord.** Поскольку у пользовательской функции вывода нет доступа к закрытой части класса **coord,** переменные x и y приходиться делать открытыми.

```
// Создание не дружественной функции вывода для объектов типа coord
#include <iostream>
using namespace std;
```

```
class coord {
public:
    int x, y; // должны быть открытыми
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
};

// Пользовательская функция вывода для объектов класса coord
ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);
    cout << a << b;

    return 0;
}
```

3. Возможности пользовательских функций вывода не ограничиваются выводом текстовой информации. Они могут выполнить любое действие или преобразование, необходимое для вывода информации в том виде, который требуется из-за особенностей устройства или ситуации. Например, совершенно разумно создать пользовательскую функцию вывода для отправки информации на плоттер. В этом случае, кроме собственно информации, функция передаст предназначенные для управления плоттером коды. Чтобы вы могли почувствовать вкус к такого рода функциям, в следующей программе создается класс **triangle,** в котором хранится ширина и высота прямоугольного треугольника. Пользовательская функция вывода этого класса выводит треугольник на экран.

```
// Эта программа рисует прямоугольные треугольники
#include <iostream>
using namespace std;

class triangle {
    int height, base;
public:
    triangle(int h, int b) { height = h; base= b; }
    friend ostream &operator<<(ostream&stream, triangle ob);
};

// рисование треугольника
ostream &operator<<(ostream &stream, triangle ob)
{
    int i, j, h, k;
```

```
    i = j = ob.base - 1;
    for(h=ob.height - 1; h; h--) {
        for(k=i; k; k--)
            stream << ' ';
        stream<<'* ';

        if(j!=i) {
            for(k=j - i - 1; k; k--)
                stream << ' ';
            stream<<'* ';
        }

        i--;
        stream << '\n';
    }
    for(k=0; k <ob.base; k++) stream << '*';
    stream<< ' \n';

    return stream;
}

int main()
{
    triangle t1(5, 5), t2 (10, 10), t3(12, 12);

    cout << t1;
    cout << endl << t2 << endl << t3;

    return 0;
}
```

Отметьте, что должным образом разработанная пользовательская функция вывода может быть целиком вставлена в "обычное" выражение ввода/вывода. После выполнения программы на экран выводится следующее:

```
    *
   * *
  *  *
 *   *
*****

          *
         * *
        *  *
       *   *
      *    *
     *     *
    *      *
   *       *
  *        *
 **********
```

```
                    *
                   * *
                  *  *
                 *   *
                *    *
               *     *
              *      *
             *       *
            *        *
           ************
```

1. В незавершенной программе имеется класс **strtype**. Для вывода строки на экран создайте пользовательскую функцию вывода:

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

class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype() {delete [] p; }
    friend ostream &operator<< (ostreamfistream, strtype &ob) ;
};

strtype::strtype(char*ptr)
{
    len = strlen(ptr) + 1;
    p = new char [len];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

// Здесь добавьте собственную функцию вывода

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    cout << s1 << '\n' << s2;

    return 0;
}
```

2. Замените в следующей программе функцию **show()** пользовательской функцией вывода:

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

class planet {
protected:
    double distance; // расстояние в милях от Солнца
```

```
    int revolve;      // полный оборот в днях
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth: public planet {
    double circumference; // окружность орбиты
public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2 * distance * 3.1416;
    }

/*Перепишите функцию show() так, чтобы информация выводилась с
помощью пользовательской функции вывода
*/
    void show() {
        cout << "Расстояние от Солнца: " << distance << '\n';
        cout << "Оборот вокруг Солнца: " << revolve << '\n';
        cout << "Окружность орбиты: " << circumference << '\n';
    }
};

int main()
{
    earth ob(93000000, 365);

    cout << ob;

    return 0;
}
```

3. Вспомните, почему пользовательская функция вывода не может быть функцией-членом.

8.6. Пользовательские функции ввода

Точно так же, как мы перефужали оператор вывода <<, можно перефузить и оператор ввода >>. В C++ оператор ввода >> иногда называют *оператором извлечения (extraction operator),* а функцию, перегружающую этот оператор, — функцией извлечения *(extractor).* Смысл этих терминов в том, что при вводе информации мы извлекаем данные из потока. Во избежание путаницы мы будем называть функцию извлечения пользовательской функцией ввода.

Здесь показана основная форма пользовательской функции ввода:

```
istream &operator>>(istream &stream, имя_класса &объект)
{
    // тело пользовательской функции ввода
```

```
    return stream;
}
```

Пользовательские функции ввода возвращают ссылку на поток istream, который является потоком ввода. Первый параметр тоже является ссылкой на поток ввода. Второй параметр — это ссылка на объект, получающий вводимую информацию.

Так же, как и пользовательская функция вывода, пользовательская функция ввода не может быть функцией-членом. Хотя внутри такой функции может быть выполнена любая операция, лучше ограничить ее работу вводом информации.

1. В этой программе к классу **coord** добавлена пользовательская функция ввода:

```
// Добавление дружественной функции вводи для объектов типа coord
#include <iostream>
using namespace std;

class coord {
    int x, y;
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob);
    friend istream &operator>>(istream &stream, coord &ob);
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ", " << ob.y << '\n';
    return stream;
}

istream &operator>>(istream &stream, coord &ob)
{
    cout << "Введите координаты: ";
    stream >> ob.x >> ob.y;
    return stream;
}

int main()
{
    coord a(1, 1), b(10, 23);

    cout << a << b;
```

```
    cin>>a;
    cout << a;

    return 0;
}
```

Обратите внимание, как пользовательская функция ввода формирует строку-приглашение для ввода данных. Хотя во многих случаях этого не требуется (или даже это нежелательно), пользовательская функция ввода показывает, как в случае необходимости почти без усложнения программы можно выдать приглашающее сообщение.

2. Здесь создается класс **inventory** (инвентарь), в котором хранится название какого-то предмета, количество выданных на руки штук и стоимость одной штуки. В программу для этого класса включены пользовательские функции ввода и вывода.

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

class inventory {
    char item[40]; // название предмета
    int onhand;    // количество предметов на руках
    double cost;   // цена предмета
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream sstream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": " << ob.onhand;
    stream << " на руках по цене $" << ob.cost << '\n';

    return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
    cout << "Введите название предмета: ";
    stream >> ob.item;
    cout << "Введите число выданных на руки экземпляров: ";
    stream >> ob.onhand;
```

```
    cout << "Введите стоимость экземпляра: ";
    stream >> ob.cost;
    return stream;
}
int main()
{
    inventory ob("hammer", 4, 12.55);
    cout << ob;
    cin>>ob;
    cout << ob;
    return 0;
}
```

Упражнения

1. Добавьте пользовательскую функцию ввода в класс **strtype** из упражнения 1 предыдущего раздела.
2. Создайте класс для хранения целого и его наименьшего делителя. Создайте для этого класса пользовательские функции ввода и вывода.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы:

1. Напишите программу для вывода числа 100 в десятичной, шестнадцатеричной и восьмеричной системе счисления. (Используйте флаги формата класса **ios.)**
2. Напишите программу для вывода значения 1000.5364 в 20-символьном поле, с выравниванием влево, с двумя позициями после запятой и символом * в качестве символа заполнения. (Используйте флаги формата класса **ios.)**
3. Перепишите ваши ответы на упражнения 1 и 2 так, чтобы использовались манипуляторы ввода/вывода.
4. Покажите, как записать и как отобразить для класса **cout** флаги формата. Используйте функции-члены либо манипуляторы.

5. Создайте для следующего класса пользовательские функции ввода и вывода:

```
class pwr {
    int base;
    int exponent;
    double result; // результат возведения в степень
public:
    pwr(int b, int e)
};

pwr::pwr(int b, int e)
{
    base = b;
    exponent = e;

    result = 1;
    for(; e; e--) result = result * base;
}
```

6. Создайте класс **box** для хранения размеров квадрата. Для вывода изображения квадрата на экран создайте пользовательскую функцию вывода. (Способ изображения выберите любой.)

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Используя показанный здесь класс **stack,** создайте пользовательскую функцию вывода для вставки в поток содержимого стека. Покажите, что функция работает.

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

#define SIZE 10

// Объявление класса stack для хранения символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos;         // индекс вершины стека
public:
    stack();
    void push(char ch); // помещение символа в стек
    char pop();         // выталкивание символа из стека
};
```

```
// Инициализация стека
stack::stack()
{
    tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон";
        return;
    }
    stck[tos]=ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}
```

2. Напишите программу с классом **watch,** который бы играл роль обычных часов. Используя стандартные функции времени, создайте конструктор класса, который должен считывать и запоминать системное время. Для вывода этого времени на экран создайте пользовательскую функцию вывода.
3. На основе класса, созданного для преобразования футов в дюймы, напишите пользовательскую функцию ввода, формирующую строку-приглашение для записи числа футов. Кроме этого, напишите пользовательскую функцию вывода для отображения на экране как числа футов, так и числа дюймов. Включите указанные функции в программу и продемонстрируйте их работоспособность.

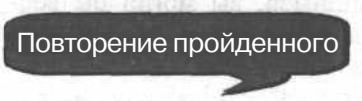
```
class ft_to_inches {
    double feet;
    double inches;
public:
    void set(double f); {
        feet = f;
        inches = f * 12;
    }
};
```

Глава 9

Дополнительные возможности ввода/вывода в C++

В этой главе продолжается изучение системы ввода/вывода C++. Здесь вы узнаете, как создать пользовательские манипуляторы ввода/вывода и как реализовать ввод/вывод в файл. Запомните, система ввода/вывода C++ очень богата, гибка и многофункциональна. Рассмотрение всех ее возможностей выходит за рамки данной книги, мы коснемся только наиболее важных из них.

Представленная в этой главе система ввода/вывода C++ определена в стандарте Standard C++ и совместима с подавляющим большинством современных компиляторов C++. Если у вас устаревший или несовместимый с современной системой ввода/вывода компилятор, то не все описанные здесь возможности будут вам доступны.



Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Напишите программу для вывода предложения: "C++ прекрасен" в поле шириной 40 символов с использованием двоеточия (:) в качестве символа заполнения.
2. Напишите программу для вывода результата деления 10/3 с четырьмя знаками после десятичной точки. Для этого используйте функции-члены класса **ios.**
3. Перепишите предыдущую программу, используя манипуляторы ввода/вывода.
4. Что такое пользовательская функция вывода? Что такое пользовательская функция ввода?

5. Дан следующий класс. Создайте для него пользовательские функции ввода и вывода.

```
class date {
    char date[9]; // дата хранится в виде строки: mm/dd/yy
public:
    // добавьте пользовательские функции ввода и вывода
};
```

6. Какой заголовок должен быть включен в программу для использования манипуляторов ввода/вывода с параметрами?
7. Какие создаются встроенные потоки, когда начинается выполнение программы на C++?

9.1. Создание пользовательских манипуляторов

В дополнение к перегрузке операторов ввода и вывода вы можете создать свою подсистему ввода/вывода C++, определив для этого собственные манипуляторы. Использование пользовательских манипуляторов важно по двум причинам. Во-первых, можно объединить последовательность нескольких отдельных операций по вводу/выводу в один манипулятор. Например, нередки ситуации, в которых в программе встречаются одинаковые последовательности операций ввода/вывода. Для выполнения такой последовательности можно создать пользовательский манипулятор. Этим вы упрощаете исходную программу и исключаете случайные ошибки. Во-вторых, пользовательский манипулятор может понадобиться, когда необходимо выполнить ввод/вывод на нестандартном оборудовании. Например, вы могли бы воспользоваться манипулятором для посылки управляющих кодов на специальный принтер или в систему оптического распознавания.

Пользовательские манипуляторы — это те элементы языка, которые обеспечивают поддержку в C++ объектно-ориентированного программирования, но они также удобны и для обычных, не объектно-ориентированных программ. Как вы увидите, пользовательские манипуляторы могут помочь сделать любую программу ввода/вывода понятней и эффективней.

Как вы знаете, имеется два базовых типа манипуляторов: те, которые работают с потоками ввода, и те, которые работают с потоками вывода. Однако кроме этих двух категорий имеются еще две: манипуляторы с аргументами и без них. Есть несколько важных отличий в способе создания манипуляторов с параметрами и без. Более того, создание манипуляторов с параметрами является существенно более трудной задачей, чем создание манипуляторов без параметров, и в этой книге не рассматривается. С другой стороны, создать пользовательский манипулятор без параметров достаточно просто, и вы скоро узнаете, как это сделать.

Все манипуляторы без параметров для вывода имеют следующую конструкцию:

```
ostream &имя_манипулятора (ostream &поток)
{
    // Код программы манипулятора

    return поток;
}
```

Здесь ***имя_манипулятора*** — это имя создаваемого вами пользовательского манипулятора, а ***поток*** — ссылка на вызывающий поток. Возвращаемым значением функции является ссылка на поток. Это необходимо в случае, когда манипулятор является частью большого выражения ввода/вывода. Важно понимать, что хотя у манипулятора и имеется в качестве единственного аргумента ссылка на поток, с которым он работает, но, когда манипулятор используется в операции вывода, его аргумент не используется.

Все манипуляторы без параметров для ввода имеют следующую конструкцию:

```
istream &имя_манипулятора (istream &поток)
{
    // Код программы манипулятора

    return поток;
}
```

Манипулятор ввода получает в качестве параметра ссылку на поток, для которого он вызывается. Манипулятор должен возвращать этот поток.

То, что манипулятор возвращает ссылку на вызывающий поток весьма важно. Если этого не сделать, то ваши манипуляторы нельзя будет использовать в последовательностях операций ввода или вывода.

1. Рассмотрим первый простой пример. В следующей программе создается манипулятор setup(), который устанавливает ширину поля вывода, равную 10, точность, равную 4, и символ заполнения *.

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

```
ostream ssetup (ostream &stream)
{
    stream.width(10);
    stream.precision(4);
    stream.fill('*');

    return stream;
}

int main()
{
    cout << setup << 123.123456;

    return 0;
}
```

Как можно заметить, созданный вами манипулятор **setup** используется в качестве части выражения ввода/вывода точно так же, как это делается с любым встраиваемым манипулятором.

2. Пользовательские манипуляторы не обязательно должны быть сложными. Например, простые манипуляторы **atn() и note()** обеспечивают простой и удобный способ вывода часто встречающихся слов и фраз.

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

// Внимание:
ostream &atn (ostream &stream)
{
    stream << "Внимание: ";
    return stream;
}

// Пожалуйста, не забудьте:
ostream &note (ostream &stream)
{
    stream << "Пожалуйста, не забудьте: ";
    return stream;
}

int main()
{
    cout << atn << "Высокое напряжение\n";
    cout << note << "Выключить свет\n";

    return 0;
}
```

Несмотря на простоту, такие манипуляторы оградят вас от необходимости частого набора одних и тех же слов и фраз.

3. В следующей программе создается манипулятор **getpass()**, который вызывает гудок динамика и затем предлагает ввести пароль:

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

// Простой манипулятор ввода
istream &getpass (istream &stream)
{
    cout << '\a'; // гудок динамика
    cout << "Введите пароль: ";

    return stream;
}

int main()
{
    char pw[80];

    do {
        cin >> getpass >> pw;
    } while (strcmp(pw, "пароль"));

    cout << "Пароль введен верно\n";

    return 0;
}
```

Упражнения

1. Создайте манипулятор вывода для отображения текущего системного времени и даты. Назовите манипулятор **td().**
2. Создайте манипулятор вывода **sethex()**, который осуществляет вывод в шестнадцатеричной системе счисления и устанавливает флаги **uppercase** и showbase. Кроме того, создайте манипулятор вывода **reset()**, который отменяет изменения, сделанные манипулятором **sethex()**.
3. Создайте манипулятор ввода **skipchar()**, который поочередно то считывает, то пропускает каждые десять символов потока ввода.

9.2. Основы файлового ввода/вывода

Как было отмечено в предыдущей главе, файловый и консольный ввод/вывод очень близко связаны. Фактически файловый ввод/вывод поддерживается той же иерархией классов, что и консольный ввод/вывод.

Таким образом, все, что вы уже узнали о вводе/выводе, вполне применимо и к файлам. Естественно, что обработка файлов предполагает и кое-что новое.

Для реализации файлового ввода/вывода, необходимо включить в программу заголовок **<fstream>**. В нем определено несколько классов, включая классы **ifstream, ofstream и fstream**. Эти классы являются производными от классов **istream** и **ostream**. Вспомните, что классы **istream и ostream**, в свою очередь, являются производными от класса **ios,** поэтому классы **ifstream, ofstream и fstream** также имеют доступ ко всем операциям, определяемым классом **ios** (это обсуждалось в предыдущей главе).

В C++ файл открывается посредством его связывания с потоком. Имеется три типа потоков: ввода, вывода и ввода/вывода. Перед тем как открыть файл, нужно, во-первых, создать поток. Для создания потока ввода необходимо объявить объект типа **ifstream.** Для создания потока вывода — объект типа **ofstream.** Потоки, которые реализуют одновременно ввод и вывод, должны объявляться как объекты типа **fstream.** Например, в следующем фрагменте создается один поток для ввода, один поток для вывода и еще один поток одновременно для ввода и для вывода:

```
ifstream in;  // ввод
ofstream out; // вывод
fstream io;   // ввод и вывод
```

После создания потока, одним из способов связать его с файлом является функция **open()**. Эта функция является членом каждого из трех потоковых классов. Здесь показаны ее прототипы для каждого класса:

```
void ifstream::open(const char *имя_файла,
                    openmode режим = ios::in);
void ofstream::open(const char *имя_файла,
                    openmode режим = ios::out I ios::trunc);
void fstream::open(const char *имя_файла,
                    openmode режим = ios::in | ios:: out);
```

Здесь ***имя_файла*** — имя файла, в которое может входить и спецификатор пути. Значение ***режим*** задает режим открытия файла. Оно должно быть значением типа **openmode,** которое является перечислением, определенным в классе **ios.** Значение ***режим*** может быть одним из следующих:

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Вы можете объединить два или более этих значения с помощью оператора OR. Рассмотрим, что означает каждое из этих значений.

Значение **ios::app** вызывает открытие файла в режиме добавления в конец файла. Это значение может применяться только к файлам, открываемым для вывода. Значение **ios::ate** задает режим поиска конца файла при его открытии. Хотя значение **ios::ate** вызывает поиск конца файла, тем не менее, операции ввода/вывода могут быть выполнены в любом месте файла.

Значение **ios::in** задает режим открытия файла для ввода. Значение **ios::out** задает режим открытия файла для вывода.

Значение **ios::binary** вызывает открытие файла в двоичном режиме. По умолчанию все файлы открываются в текстовом режиме. В текстовом режиме имеет место преобразование некоторых символов, например, последовательность символов "возврат каретки/перевод строки" превращается в символ новой строки. Если же файл открывается в двоичном режиме, такого преобразования не выполняется. Запомните, что любой файл, независимо от того, что в нем содержится — отформатированный текст или необработанные данные — может быть открыт как в текстовом, так и в двоичном режиме. Отличие между ними только в отсутствии или наличии упомянутого символьного преобразования.

Значение **ios::trunc** приводит к удалению содержимого ранее существовавшего файла с тем же названием и усечению его до нулевой длины. При создании потока вывода с помощью ключевого слова **ofstream** любой ранее существовавший файл с тем же именем автоматически усекается до нулевой длины.

В следующем фрагменте для вывода открывается файл **test:**

```
ofstream mystream;
mystream.open("test");
```

В этом примере параметр ***режим*** функции **open()** по умолчанию устанавливается в значение, соответствующее типу открываемого потока, поэтому нет необходимости указывать его явно.

Если выполнение функции **open()** завершилось с ошибкой, в булевом выражении поток будет равен значению **false.** Этот факт можно использовать для проверки правильности открытия файла с помощью, например, такой инструкции:

```
if(!mystream) {
    cout << "Файл открыть невозможно\n";
    // программа обработки ошибки открытия файла
}
```

Как правило, перед тем как пытаться получить доступ к файлу, следует проверить результат выполнения функции **open()**.

Проверить правильность открытия файла можно также с помощью функции **is_open(),** являющейся членом классов **ifstream, ofstream и fstream**. Ниже показан прототип этой функции:

```
bool is_open();
```

Функция возвращает истину, если поток удалось связать с открытым файлом, в противном случае функция возвращает ложь. Например, в следующем фрагменте проверяется, открыт ли файл, связанный с потоком **mystream**:

```
if(!mystream.is_open()) {
    cout << "Файл не открыт\n";
    // ...
}
```

Хотя использовать функцию **open()** для открытия файла в целом правильно, часто вы этого делать не будете, поскольку у классов **ifstream, ofstream и fstream** есть конструкторы, которые открывают файл автоматически. Конструкторы имеют те же параметры, в том числе и задаваемые по умолчанию, что и функция **open().** Поэтому чаще вы будете пользоваться таким способом открытия файла:

```
ifstream mystream("myfile"); // открытие файла для ввода
```

Как уже установлено, если по каким-то причинам файл не открывается, переменная, соответствующая потоку, в условной инструкции будет равна значению **false.** Поэтому, независимо от того, используете ли вы конструктор или явно вызываете функцию **open()**, вам потребуется убедиться в успешном открытии файла путем проверки значения потока.

Для закрытия файла используйте функцию-член **close()**. Например, чтобы закрыть файл, связанный с потоком **mystream,** необходима следующая инструкция:

```
mystream.close();
```

Функция **close()** не имеет параметров и возвращаемого значения.

С помощью функции **eof(),** являющейся членом класса **ios,** можно определить, был ли достигнут конец файла ввода. Ниже показан прототип этой функции:

```
bool eof();
```

Функция возвращает истину, если был достигнут конец файла; в противном случае функция возвращает ложь.

После того как файл открыт, очень легко считать из него или записать в него текстовые данные. Просто используйте операторы << и >> так же, как это делалось для консольного ввода/вывода, только замените поток **cin** или **cout** тем потоком, который связан с файлом. Так же, как и операторы << и >> для чтения из файла и записи в файл годятся функции C — **fprintf() и fscanf()**. Вся информация в файле хранится в том же формате, как если бы она находилась на экране. Следовательно, файл, созданный с помощью оператора <<, представляет из себя файл с отформатированным текстом, и

наоборот, любой файл, содержимое которого считывается с помощью оператора >>, должен быть файлом с отформатированным текстом. То есть, как правило, файлы с отформатированным текстом, которые вы будете обрабатывать, используя операторы << и >>, следует открывать в текстовом, а не в двоичном режиме. Двоичный режим больше подходит для неотформатированных файлов, о которых в этой главе будет рассказано позже.

1. В представленной ниже программе создается файл для вывода, туда записывается информация, и файл закрывается. Затем файл снова открывается уже в качестве файла для ввода, и записанная ранее информация оттуда считывается:

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

int main()
{
    ofstream fout("test"); // создание файла для вывода

    if(!fout) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    fout << "Привет!\n";
    fout << 100 <<  '  ' << hex << 100 << endl;

    fout.close();

    ifstream fin("test"); // открытие файла для ввода
    if(!fin) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char str[80];
    int i;

    fin>>str>>i;
    cout << str << '  ' << i << endl;

    fin.close();

    return 0;
}
```

После того как программа завершится, проанализируйте содержимое файла **test.** Оно будет следующим:

```
Привет!
100  64
```

Как уже установлено, при использовании операторов << и >> для реализации файлового ввода/вывода, информация форматируется так же, как если бы она находилась на экране.

2. Рассмотрим другой пример файлового ввода/вывода. В этой программе введенные с клавиатуры строки считываются и записываются в файл. Программа завершается при вводе знака доллара $ в качестве первого символа строки. Для использования программы в командной строке задайте имя файла для вывода.

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Введите <имя_файла>\n";
        return 1;
    }

    ofstream out(argv[1]); // файл для вывода

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char str[80];
    cout << "Вводите строки; для окончания ввода введите $\n";

    do {
        cout << ": ";
        cin>>str;
        out << str << endl;
    } while (*str != '$');

    out.close();
    return 0;
}
```

3. В следующей программе копируется текстовый файл и при этом пробелы превращаются в символы |. Обратите внимание, как для контроля конца файла для ввода используется функция **eof().** Также обратите внимание, как

поток ввода **fin** воспринимает сброс флага **skipws.** Это предотвращает пропуск пробелов в начале строк.

```
// Превращение пробелов в вертикальные линии |
#include <iostream>
ttinclude <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Преобразование <файл_ввода> <файл_вывода>\n";
        return 1;
    }

    ifstream fin(argv[1]); // открытие файла для ввода
    ofstream fout(argv[2]); // создание файла для вывода

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    if(!fin) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char ch;
    fin.unsetf(ios::skipws); // не пропускать пробелы
    while(!fin.eof()) {
        fin>>ch;
        if(ch==' ') ch = '|';
        if(!fin.eof()) fout << ch;
    }

    fin.close();
    fout.close();

    return 0;
}
```

4. Между исходной библиотекой ввода/вывода C++ и библиотекой ввода/вывода современного стандарта Standard C++ имеются некоторые отличия, которые необходимо учитывать при модернизации старых программ. Во-первых, в исходной библиотеке ввода/вывода C++ у функции **open()** имеется третий параметр, задающий режим защиты файла. По умолчанию это режим обычного файла. В современной библиотеке C++ указанный параметр не поддерживается.

   Во-вторых, при работе со старой библиотекой для открытия потока ввода/вывода **fstream** необходимо явно указать значения режима открытия файла

ios::in и ios::out. Значения режима открытия файла по умолчанию не поддерживаются. Это относится как к конструктору класса fstream, так и к функции open(). Например, при работе со старой библиотекой ввода/вывода C++, чтобы открыть файл для ввода и вывода с помощью функции **open()**, необходимо использовать следующую конструкцию:

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

В современной библиотеке ввода/вывода C++, если режим открытия файла не указан, любой объект типа fstream автоматически открывает файл для ввода и вывода.

И последнее. При работе со старой библиотекой ввода/вывода к ошибке выполнения функции open() ведет значение режима открытия файла, равное ios::noncreatc, если указанный файл не существует, или равное **ios::noreplace**, если, наоборот, указанный файл уже существует. В стандарте Standard C++ данные значения режима открытия файла не поддерживаются.

1. Напишите программу для копирования текстового файла. Эта программа должна подсчитывать число копируемых символов и выводить на экран полученный результат. Почему это число отличается от того, которое выводится при просмотре списка файлов каталога?
2. Напишите программу для заполнения информацией следующей таблицы в файле phone.

   Исаак Ньютон, 415 555-3423
   Роберт Годдард, 213 555-2312
   Энрико Ферми, 202 555-1111

3. Напишите программу для подсчета числа слов в файле. Для простоты считайте, что словом является все, имеющее с двух сторон пробелы.
4. Какие действия выполняет функция is_open?

9.3. Неформатируемый двоичный ввод/вывод

Хотя текстовые файлы (т. е. файлы, информация в которых представлена в кодах ASCII, — *примеч. пер.)* полезны во многих ситуациях, у них нет гибкости неформатированных двоичных файлов. Неформатированные файлы содержат те самые исходные или "сырые" двоичные данные, которые непосредственно используются вашей профаммой, а не удобный для восприятия

человека текст, данные для которого транслируются операторами << и >>. Поэтому о неформатируемом вводе/выводе иногда говорят как о "сыром" (raw) вводе/выводе. В C++ для двоичных файлов поддерживается широкий диапазон функций ввода/вывода. Эти функции дают возможность точно контролировать процессы считывания из файлов и записи в файлы.

На нижнем уровне двоичного ввода/вывода находятся функции **get()** и **put(). С** помощью функции-члена **put()** можно записать байт; а с помощью функции-члена **get()** — считать. Эти функции являются членами всех потоковых классов соответственно для ввода и для вывода. Функции **get() и put()** имеют множество форм. Ниже приведены их наиболее часто встречающиеся версии:

```
istream figet(char &символ);
ostream &put(char символ);
```

Функция **get()** считывает один символ из связанного с ней потока и передает его значение аргументу ***символ.*** Ее возвращаемым значением является ссылка на поток. При считывании символа конца файла функция возвратит вызывающему потоку значение **false.** Функция **put()** записывает ***символ*** в поток и возвращает ссылку на поток.

Для считывания и записи блоков двоичных данных используются функции **read()** и **write()**, которые также являются членами потоковых классов соответственно для ввода и для вывода. Здесь показаны их прототипы:

```
istream &read(char *буфер, streamsize число_байт);
ostream &write(const char *буфер, streamsize число_байт);
```

Функция **read()** считывает из вызывающего потока столько байтов, сколько задано в аргументе ***число_байт*** и передает их в буфер, определенный указателем ***буфер.*** Функция **write()** записывает в соответствующий поток из буфера, который определен указателем ***буфер,*** заданное в аргументе ***число_байт*** число байтов. Значения типа **streamsize** представляют собой некоторую форму целого.

Если конец файла достигнут до того, как было считано ***число_байт*** символов, выполнение функции **read()** просто прекращается, а в буфере оказывается столько символов, сколько их было в файле. Узнать, сколько символов было считано, можно с помощью другой функции-члена **gcount()**, прототип которой приведен ниже:

```
streamsize gcount();
```

Функция возвращает количество символов, считанных во время последней операции двоичного ввода.

Естественно, что при использовании функций, предназначенных для работы с двоичными файлами, файлы обычно открывают в двоичном, а не в тек-

стовом режиме. Смысл этого понять легко, значение режима открытия файла **ios::binary** предотвращает какое бы то ни было преобразование символов. Это важно, когда в файле хранятся двоичные данные, например, целые, вещественные или указатели. Тем не менее, для файла, открытого в текстовом режиме, хотя в нем содержится только текст, двоичные функции также вполне доступны, но при этом помните о возможности нежелательного преобразования символов.

1. В следующей программе на экран выводится содержимое файла. Используется функция **get().**

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Содержимое: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    while(!in.eof()) {
        in.get(ch);
        cout << ch;
    }

    in.close();

    return 0;
}
```

2. В данной программе для записи в файл вводимых пользователем символов используется функция **put().** Программа завершается при вводе знака доллара $.

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

```
int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Запись: <имя_файла>\n";
        return 1;
    }

    ofstream out(argv[1], ios::out | ios::binary);
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    cout << "Для остановки введите символ $\n";
    do {
        cout << ": ";
        cin.get(ch);
        out.put(ch);
    } while (ch!='$');

    out.close();

    return 0;
}
```

Обратите внимание, что для считывания символов из потока cin в программе используется функция **get().** Это предотвращает игнорирование начальных пробелов.

3. В следующей программе для записи строки и числа типа **double** в файл **test** используется функция **wtite()**:

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

int main()
{
    ofstream out("test", ios::out I ios::binary);
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    double num = 100.45;
    char str[] = "Это проверка";
```

```
    out.write((char *) &num, sizeof(double));
    out.write(str, strlen(str));

    out.close();

    return 0;
}
```

*Приведение типа к **(char*)** при вызове функции **write()** необходимо, если буфер вывода не определен как символьный массив. Поскольку в C++ осуществляется строгий контроль типов, указатель на один тип не преобразуется автоматически в указатель на другой тип.*

4. В следующей программе для считывания из файла, созданного в программе примера 3, используется функция **read():**

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

int main()
{
    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    double num;
    char str[80];

    in.read((char *) Snum, sizeof(double));
    in.read(str, 15) ;
    str[14] = '\0';
    cout << num << ' ' << str;

    in.close();

    return 0;
}
```

Как и в программе из предыдущего примера, приведение типов внутри функции **read()** необходимо, поскольку в C++ указатель одного типа автоматически не преобразуется в указатель другого типа.

5. В следующей программе сначала массив **double** записывается в файл, а затем считывается обратно. Кроме того, отображается число считанных символов.

```
// Демонстрация работы функции gcount ()
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    double nums[4] = {1.1, 2.2, 3.3, 4.4};

    out.write((char *) nums, sizeof(nums));
    out.close();

    ifstream in("test", ios::in | ios::binary);

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    in.read((char *) &nums, sizeof(nums));

    int i;
    for(i=0; i<4; i++)
        cout << nums[i] << ' ';

    cout << '\n';

    cout << in.gcount() << " символов считано\n";
    in.close();

    return 0;
}
```

Упражнения

1. Измените ответы на упражнения 1 и 3 раздела 9.2 так, чтобы в них использовались функции **get(), put(), read()** и/или **write()**. (Используйте эти функции там, где они, по вашему мнению, принесут наибольшую отдачу.)

2. Дан следующий класс. Напишите программу для вывода содержимого класса в файл. Для этой цели создайте пользовательскую функцию вывода.

```
class account {
    int custnum;
    char name[80];
    double balance;
public:
    account(int c, char *n, double b)
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }
    // здесь нужна пользовательская функция вывода
};
```

9.4. Дополнительная информация о функциях двоичного ввода/вывода

Кроме представленной ранее формы, функцию **get()** можно перегрузить еще несколькими способами. Здесь показаны прототипы трех наиболее часто перегружаемых форм:

```
istream &get(char *буфер, streamsize число_байт);
istream &get(char *буфер, streamsize число_байт,
                                    char ограничитель);
int get();
```

Первая функция **get()** считывает символы в массив, определенный указателем ***буфер,*** до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт*** — **1,** либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер,*** функция **get()** помещает ноль. Если в потоке ввода встретится символ новой строки, он *не* извлекается, а остается в потоке до следующей операции ввода.

Вторая функция **get()** считывает символы в массив, определенный указателем ***буфер,*** до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт*** — \, либо не встретился символ, заданный параметром ***ограничитель,*** либо не встретился символ конца файла. В конце массива, заданного указателем ***буфер,*** функция **get()** помещает ноль. Если в потоке ввода встретится символ ***ограничитель,*** он *не* извлекается, а остается в потоке до следующей операции ввода.

Третья функция **get()** возвращает из потока следующий символ. Она возвращает символ EOF, если достигнут конец файла. Эта форма функции **get()** напоминает функцию **getc()** языка C.

Другой функцией для реализации ввода является функция getline(). Эта функция — член всех потоковых классов ввода. Ниже показаны ее прототипы:

```
istream &getline(char *буфер, streamsize число_байт);
istream &getline(char *буфер, streamsize число_байт,
                                  char ограничитель);
```

Первая функция считывает символы в массив, обозначенный указателем *буфер,* до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт*** — 1, либо не встретился символ новой строки, либо не встретился символ конца файла. В конце массива, заданного указателем *буфер,* функция **getline()** помещает ноль. Если в потоке ввода встретится символ новой строки, он извлекается, но не помещается в массив.

Вторая функция считывает символы в массив, обозначенный указателем *буфер,* до тех пор, пока либо не считано столько символов, сколько задано параметром ***число_байт*** — 1, либо не встретился символ *ограничитель,* либо не встретился символ конца файла. В конце массива, заданного указателем *буфер,* функция getlineQ помещает ноль. Если в потоке ввода встретится символ *ограничитель,* он извлекается, но не помещается в массив.

Как можно заметить, обе версии функции **getline()** фактически тождественны версиям **get(*буфер, число_байт*)** и **get(*буфер, число_байт, ограничитель*)** функции get(). Обе считывают символы из потока ввода и помещают их в массив, обозначенный указателем *буфер* до тех пор, пока либо не считано ***число_байт*** — \ символов, либо не встретился символ *ограничитель* или символ конца файла. Отличие между функциями get() и **getline()** в том, что функция **getline()** считывает и удаляет из потока ввода символ *ограничитель,* а функция get() — нет.

Используя функцию **peek()**, можно получить следующий символ из потока ввода без его удаления из потока. Функция является членом потоковых классов ввода и имеет следующий прототип:

```
int peek();
```

Функция возвращает следующий символ из потока или, если достигнут конец файла, символ EOF.

С помощью функции putback(), являющейся членом потоковых классов ввода, можно возвратить последний считанный из потока символ обратно в поток. Ниже показан прототип этой функции:

```
istream fiputback (char c);
```

Здесь *с* — это последний считанный из потока символ.

При выполнении вывода данные не сразу записываются на связанное с потоком физическое устройство, а информация временно сохраняется во внутреннем буфере. Только после заполнения буфера его содержимое переписывается на диск. Однако вызов функции flush() вызывает физическую

запись информации на диск до заполнения буфера. Ниже показан прототип функции **flushQ,** являющейся членом потоковых классов вывода:

```
ostream &flush();
```

Вызовы функции **flush()** оправданы при работе в неблагоприятной обстановке (например, в ситуациях, когда часто случаются сбои по питанию).

Примеры

1. Как вы знаете, при использовании для считывания строки оператора >>, считывание прекращается при встрече первого разделительного символа. При наличии в строке пробелов такое считывание становится невозможным. Однако, как показано в программе, с помощью функции **getline()** можно решить эту проблему:

```
// Использование функции getline() для считывания строки с пробелами
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char str[80];

    cout << "Введите ваше имя: ";
    cin.getline(str, 79);

    cout << str << '\n';

    return 0;
}
```

   В данном случае ограничителем для функции getline() является символ новой строки. Это делает выполнение функции **getline()** очень похожей на выполнение стандартной функции gets().

2. В реальном программировании особенно полезны функции peek() и putback(). Они позволяют упростить управление, когда неизвестен тип вводимой в каждый конкретный момент времени информации. Следующая программа иллюстрирует это. В ней из файла считываются строки либо целые. Строки и целые могут следовать в любом порядке.

```
// Демонстрация работы функции peek()
#include <iostream>
linclude <fstream>
#include <cctype>
using namespace std;

int main()
```

```
{
    char ch;
    ofstream out("test", ios::out | ios::binary);

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char str[80], *p;

    out << 123 << "this is a test" << 23;
    out << "Hello there!" << 99 << "sdf" << endl;
    out.close();

    ifstream in("test",ios::in | ios::binary);

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    do {
        p = str;
        ch = in.peek(); // выяснение типа следующего символа
        if(isdigit(ch)) {
            while(isdigit(*p=in.get()) ) p++; // считывание целого
            in.putback(*p); // возврат символа в поток
            *p = '\0'; // заканчиваем строку нулем
            cout << "Целое: " << atoi(str);
        }
        else if(isalpha(ch)) { // считывание строки
            while(isalpha(*p=in.get()) ) p++;
            in.putback(*p); // возврат символа в поток
            *p = '\0'; // заканчиваем строку нулем
            cout << "Строка: " << str;
        }
        else in.get(); // пропуск
        cout <<'\n';
    }while(!in.eof());

    in.close();
    return 0;
}
```

1. Перепишите программу из примера 1 так, чтобы вместо функции **getline()** использовать функцию **get()**. Будет ли отличаться работа программы?

2. Напишите программу для построчного считывания текстового файла и вывода каждой считанной строки на экран. Используйте функцию **getline()**.
3. Подумайте о ситуациях, в которых может оказаться полезным вызов функции flush().

9.5. Произвольный доступ

В системе ввода/вывода C++ *произвольный доступ (random access)* реализуется с помощью функций **seekg()** и **seekp()**, являющихся соответственно потоковыми функциями ввода и вывода. Здесь показаны их основные формы:

```
istream &seekg(off_type смещение, seekdir задание);
ostream fiseekp(off_type смещение, seekdir задание);
```

Здесь **off_type** — это целый тип данных, определенный в классе ios и совместимый с максимальным правильным значением, которое способен хранить параметр *смещение*. Тип seekdir — это перечисление, определенное в классе ios и содержащее следующие значения:

Значение	Смысл
ios::beg	Поиск с начала файла
ios::cur	Поиск от текущей позиции в файле
ios::end	Поиск с конца файла

Система ввода/вывода C++ управляет двумя указателями, связанными с файлом. Первый — это *указатель считывания (get pointer),* который задает следующее место в файле, откуда будет вводиться информация. Второй — это *указатель записи (put pointer),* который задает следующее место в файле, куда будет выводиться информация. При каждом вводе или выводе соответствующий указатель последовательно продвигается дальше. Однако с помощью функций **seekg()** и **seekp()** возможен непоследовательный доступ к файлу.

Функция seekg() устанавливает указатель считывания соответствующего файла в позицию, отстоящую на величину *смещение* от заданного места *задание*. Функция **seekp()** устанавливает указатель записи соответствующего файла в позицию, отстоящую на величину *смещение* от заданного места *задание*.

Как правило, файлы, доступные для функций seekgQ и **seekp()**, должны открываться в режиме операций для двоичных файлов. Таким образом предотвращается возможное неожиданное преобразование символов внутри файла.

Определить текущую позицию каждого из двух указателей можно с помощью функций:

```
pos_type tellg();
pos_type tellp();
```

Здесь **pos_type** — это целый тип данных, определенный в классе ios и способный хранить наибольшее возможное значение указателя.

Для перемещения файловых указателей считывания и записи на позицию, заданную возвращаемыми значениями функций **tellg() и tellp()**, используются перегруженные версии функций **seekg() и seekp()**. Прототипы этих функций представлены ниже:

```
istream fiseekg(pos_type позиция);
ostream fiseekp(pos_type позиция);
```

1. В следующей программе показана работа функции seekp(). Она позволяет заменить в файле заданный символ. Укажите в командной строке имя файла, затем номер байта в файле, который вы хотите изменить, и, наконец, новый символ для замены. Обратите внимание: файл открывается для операций чтения и записи.

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

int main(int argc, char *argv[])
{
    if(argc!=4) {
        cout << "Замена: <файл> <байт> <символ>\n";
        return 1;
    }

    fstream out(argv[1], ios::in | ios::out | ios::binary);

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);

    out.put(*argv[3]);
    out.close();

    return 0;
}
```

2. В следующей программе функция **seekg()** используется для установки указателя считывания в заданную позицию внутри файла и для вывода содержимого файла, начиная с этой позиции. Имя файла и позиция начала считывания задаются в командной строке.

```
// Демонстрация работы функции seekg()
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Поиск: <файл> <позиция>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg);

    while(!in.eof()) {
        in.get(ch);
        cout << ch;
    }

    in.close();

    return 0;
}
```

Упражнения

1. Напишите программу для вывода на экран содержимого текстового файла в обратном порядке. *(Подсказка:* Обдумайте задание перед началом программирования. Решение проще, чем может показаться на первый взгляд.)
2. Напишите программу, которая попарно меняет местами символы в текстовом файле. Например, если в файле содержится "1234", то после выполнения программы там должно содержаться "2143". (Для простоты считайте, что в файле содержится четное число символов.)

9.6. Контроль состояния ввода/вывода

В системе ввода/вывода C++ поддерживается информация о состоянии после каждой операции ввода/вывода. Текущее состояние потока ввода/вывода, которое хранится в объекте типа **iostate,** является перечислением, определенным в классе ios и содержащим следующие члены:

Название	Значение
goodbit	Ошибок нет
eofbit	Достигнут конец файла
failbit	Имеет место нефатальная ошибка
badbit	Имеет место фатальная ошибка

В устаревших компиляторах флаги состояния ввода/вывода хранятся как целые, а не как объекты типа **iostate.**

Имеются два способа получения информации о состоянии ввода/вывода. Во-первых, можно вызвать функцию **rdstate(),** являющуюся членом класса **ios.** Прототип этой функции:

```
iostate rdstate();
```

Функция возвращает текущее состояние флагов ошибки. Как вы, вероятно, догадываетесь, глядя на приведенный выше список флагов, функция **rdstate()** возвращает флаг **goodbit** при отсутствии какой бы то ни было ошибки. В противном случае она возвращает флаг ошибки.

Другим способом определения того, имела ли место ошибка, является использование одной или нескольких следующих функций — членов класса **ios:**

```
bool bad();
bool eof();
bool fail();
bool good();
```

Функция **eof()** уже обсуждалась. Функция **bad()** возвращает истину, если установлен флаг **badbit.** Функция **fail()** возвращает истину, если установлен флаг failbit. Функция **good()** возвращает истину при отсутствии ошибок. В противном случае функции возвращают ложь.

После появления ошибки может возникнуть необходимость сбросить это состояние перед тем, как продолжить выполнение программы. Для этого используется функция **clear(),** являющаяся членом класса **ios.** Ниже приведен прототип этой функции:

```
void clear(iostate флаги = ios::goodbit);
```

Если параметр ***флаги*** равен **goodbit** (значение по умолчанию), то сбрасываются флаги всех ошибок. В противном случае переменной ***флаги*** присваиваются значения тех флагов, которые вы хотите сбросить.

1. В следующей программе иллюстрируется выполнение функции **rdstate().** Программа выводит на экран содержимое текстового файла. При наличии ошибки функция сообщает об этом с помощью функции **checkstatus()**.

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

void checkstatus(ifstream &in);

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Содержимое: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char c;
    while(in.get(c)) {
        cout << c;
        checkstatus(in);
    }

    checkstatus(in); // контроль финального состояния
    in.close();
    return 0;
}

void checkstatus(ifstream &in)
{
    ios::iostate i;
    i = in.rdstate();

    if(i & ios::eofbit)
        cout << "Достигнут EOF\n";
    else if(i & ios::failbit)
        cout << "Нефатальная ошибка ввода/вывода\n";
```

```
    else if(i & ios::badbit)
        cout << "Фатальная ошибка ввода/вывода\n";
}
```

Эта программа всегда будет выводить сообщение по крайней мере об одной ошибке. После окончания цикла **while** последний вызов функции **checkstatus()**, как и ожидается, выдаст сообщение о достижении конца файла (символа EOF).

2. В следующей программе с помощью функции **good()** файл проверяется на наличие ошибки:

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {"
        cout << "Содержимое: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    while(!in.eof()){
        in.get(ch);
        // контроль ошибки
        if(!in.good() && !in.eof()) {
            cout << "Ошибка ввода/вывода ... прерывание работы\n";
            return 1;
        }
        cout << ch;
    }

    in.close();

    return 0;
}
```

1. Добавьте контроль ошибок в ваши ответы на вопросы предыдущего раздела.

9.7. Пользовательский ввод/вывод и файлы

В предыдущей главе вы изучили перегрузку операторов ввода и вывода для создаваемых вами классов. При этом рассматривался только консольный ввод/вывод. Однако поскольку все потоки C++ одинаковы, то одинаково перегруженная, например, функция вывода, может использоваться без каких-либо изменений для вывода как на экран, так и в файл. Это одна из наиболее важных и полезных возможностей ввода/вывода в C++.

Как установлено в предыдущей главе, перегруженные функции ввода/вывода так же, как и манипуляторы ввода/вывода могут использоваться с любым потоком. Если вы "жестко" зададите конкретный поток в функции ввода/вывода, область ее применения, несомненно, будет ограничена только этим потоком. Следует, по возможности, разрабатывать такие функции ввода/вывода, чтобы они могли одинаково работать с любыми потоками.

1. В следующей программе относительно класса **coord** перегружаются операторы << и >>. Обратите внимание, что одни и те же оператор-функции можно использовать для вывода как на экран, так и в файл.

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

class coord {
    int x, y;
public:
    coord(int i, int j) { x = i; y = j; }
    friend ostream &operator<<(ostream &stream, coord ob) ;
    friend istream &operator>>(istream &stream, coord &ob) ;
};

ostream &operator<<(ostream &stream, coord ob)
{
    stream << ob.x << ' ' << ob.y << '\n';

    return stream;
}

istream &operator>>(istream &stream, coord &ob)
'{
    stream >> ob.x >> ob.y;

    return stream;
}
```

```
int main()
{
    coord o1(1, 2), o2(3, 4);
    ofstream out("test");

    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    out << o1 << o2;

    out.close();

    ifstreamin("test");

    if(!in) {
        cout << "Файл открыть невозможно\n";
        return 1 ;
    }

    coord o3(0, 0) , o4 (0, 0) ;
    in>>o3 >>o4 ;

    cout << o3 << o4;

    in.close();

    return 0 ;
}
```

2. Все манипуляторы ввода/вывода подходят и для файлового ввода/вывода. Например, в представленной ниже переработанной версии одной из программ этой главы, тот манипулятор, который выводит информацию на экран, используется и для ее записи в файл.

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

// Внимание:
ostream &atn(ostream &stream)
 (
    stream << "Внимание: ";
    return stream;
}

// Пожалуйста, не забудьте:
ostream &note (ostream &stream)
{
    stream << "Пожалуйста, не забудьте: ";
```

```
    return stream;
}
int main()
{
    ofstream out("test");
    if(!out) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    // вывод на экран
    cout << atn << "Высокое напряжение \n";
    cout << note << "Выключить свет\n";
    // вывод в файл
    out << atn << "Высокое напряжение\n";
    out << note << "Выключить свет\n";
    out.close();
    return 0;
}
```

Упражнения

1. Попытайтесь адаптировать программы предыдущей главы для работы с файлами.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Создайте манипулятор для вывода трех символов табуляции и установки ширины поля равной 20. Продемонстрируйте работу манипулятора.
2. Создайте манипулятор для ввода, который должен считывать и отбрасывать все неалфавитные символы. При считывании первого алфавитного символа, манипулятор должен возвратить его во входной поток и закончить работу. Назовите манипулятор **findalpha.**
3. Напишите программу копирования текстового файла. При копировании измените регистр всех букв.

4. Напишите программу, которая считывает текстовый файл, а затем сообщает, сколько раз каждая буква алфавита появляется в файле.
5. Если вы еще этого не сделали, добавьте в ваши решения упражнений 3 и 4 полный контроль ошибок.
6. Какая функция перемещает указатель считывания? Какая функция перемещает указатель записи?

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Ниже приведена переработанная версия класса **inventory** из предыдущей главы. Добавьте функции **store() и retrieve()**. Затем создайте небольшой файл, содержащий несколько инвентарных записей. Далее, используя произвольный доступ, по номеру записи отобразите на экране информацию об одном из элементов.

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

tfdefine SIZE 40

class inventory {
    char item[SIZE]; // название предмета
    int onhand;      // количество выданных на руки экземпляров
    double cost;     // цена предмета
public:
    inventory(char *i, int o, double c);
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    void store(fstream &stream) ;
    void retrieve(fstream &stream) ;
    friend ostream &operator<<(ostream &stream, inventory ob) ;
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": " << ob.onhand;
    stream << "на руках по цене ." << ob.cost << '\n';
```

```
    return stream;
}

istream &operator>>(istream &stream, inventory &ob)
{
    cout << "Введите название предмета: ";
    stream >> ob.item;
    cout << "Введите число выданных экземпляров: ";
    stream >> ob.onhand;
    cout << "Введите стоимость экземпляра: ";
    stream >> ob.cost;

    return stream;
}
```

2. Необязательное задание. Создайте класс **stack** для хранения символов *в* файле, а не в массиве.

Глава 10

Виртуальные функции

В этой главе рассматривается следующий важный аспект C++: *виртуальные функции (virtual functions)*. Виртуальные функции важны потому, что они используются для поддержки динамического полиморфизма (run-time polymorphism). Как вы знаете, в C++ полиморфизм поддерживается двумя способами. Во-первых, при компиляции он поддерживается посредством перегрузки операторов и функций. Во-вторых, во время выполнения программы он поддерживается посредством виртуальных функций. Здесь вы узнаете, как с помощью динамического полиморфизма можно повысить гибкость программ.

Основой виртуальных функций и динамического полиморфизма являются указатели на производные классы. Поэтому эта глава начинается с обсуждения указателей на производные классы.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Создайте манипулятор для вывода чисел в научной нотации с символом E в верхнем регистре.
2. Напишите программу для копирования текстового файла. В процессе копирования преобразуйте каждый символ табуляции в соответствующее число пробелов.
3. Напишите программу для поиска в текстовом файле слова, заданного в командной строке. После выполнения программы на экране должно появиться число, обозначающее, сколько раз данное слово найдено в файле. Для простоты считаем следующее: все, что с обеих сторон окружено пробелами, есть слово.
4. Напишите инструкцию, которая устанавливает указатель записи на 234-й байт в файле, связанном с потоком out.
5. Какие функции выдают информацию о состоянии системы ввода/вывода C++?

6. Приведите хотя бы одно преимущество от использования функций ввода/вывода C++ по сравнению с соответствующими функциями системы ввода/вывода языка C.

10.1. Указатели на производные классы

Хотя в главе 4 довольно обстоятельно обсуждались указатели C++, одна их специфическая особенность до сих пор опускалась, поскольку она тесно связана с виртуальными функциями. Этой особенностью является следующее: указатель, объявленный в качестве указателя на базовый класс, также может использоваться, как указатель на любой класс, производный от этого базового. В такой ситуации представленные ниже инструкции являются правильными:

```
base *p; // указатель базового класса

base base_ob; // объект базового класса
derived derived ob; // объект производного класса

// Естественно, что указатель p может указывать
// на объект базового класса
p = &base_ob; // указатель p для объекта базового класса

// Кроме базового класса указатель p может указывать
// на объект производного класса
p = &derived_ob; // указатель p для объекта производного класса
```

Как отмечено в комментариях, указатель базового класса может указывать на объект любого класса, производного от этого базового и при этом ошибка несоответствия типов генерироваться не будет.

Для указания на объект производного класса можно воспользоваться указателем базового класса, при этом доступ может быть обеспечен только к тем объектам производного класса, которые были унаследованы от базового. Объясняется это тем, что базовый указатель "знает" только о базовом классе и ничего не знает о новых членах, добавленных в производном классе.

Указатель базового класса можно использовать для указания на объект производного класса, но обратный порядок недействителен. Указатель производного класса нельзя использовать для доступа к объектам базового класса. (Чтобы обойти это ограничение, можно использовать приведение типов, но на практике так действовать не рекомендуется.)

И последнее: запомните, что арифметика указателей связана с типом данных (т. е. классом), который задан при объявлении указателя. Таким образом, если указатель базового класса указывает на объект производного класса, а затем инкрементируется, то он уже не будет указывать на следующий объект производного класса. Этот указатель будет указывать на следующий объект базового класса. Помните об этом.

Примеры

1. В этой короткой программе показано, как указатель базового класса может использоваться для доступа к объекту производного класса:

```
// Демонстрация указателя на объект производного класса
#include <iostream>
using namespace std;

class base {
    int x;
public:
    void setx(int i) { x = i; }
    int getx() { return x; }
};

class derived: public base {
    int y;
public:
    void sety(int i) { y = i; }
    int gety() {return y; }
};

int main()
{
    base *p;      // указатель базового класса
    base b_ob;    // объект базового класса
    derived d_ob; // объект производного класса

    // использование указателя p
    // для доступа к объекту базового класса
    p = &b_ob;
    p->setx(10); // доступ к объекту базового класса
    cout << "Объект базового класса x: " << p->getx() << '\n';

    // использование указателя p
    // для доступа к объекту производного класса
    p = &d_ob;   // указывает на объект производного класса
    p->setx(99) ; // доступ к объекту производного класса

    // т. к. p нельзя использовать для установки y,
    // делаем это напрямую
    d_ob.sety(88);
cout << "Объект производного класса x: " << p->getx() << ' ' ;
cout << "Объект производного класса y: " << d_ob.gety() << '\n';

    return 0;
}
```

Нет смысла использовать указатели на объекты базового класса так, как показано в этом примере. Однако в следующем разделе вы увидите, почему для объектов производного класса столь важны указатели на объекты базового класса.

1. Попытайтесь запустить рассмотренную выше программу и поэкспериментируйте с ней. Например, попробуйте, объявив указатель на производный класс, получить доступ к объекту базового класса.

10.2. Знакомство с виртуальными функциями

Виртуальная функция (virtual function) является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы функция стала виртуальной, перед объявлением функции ставится ключевое слово **virtual.** Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется. По существу, виртуальная функция реализует идею "один интерфейс, множество методов", которая лежит в основе полиморфизма. Виртуальная функция внутри базового класса определяет *вид интерфейса* этой функции. Каждое переопределение виртуальной функции в производном классе определяет ее реализацию, связанную со спецификой производного класса. Таким образом, переопределение создает *конкретный метод.* При переопределении виртуальной функции в производном классе, ключевое слово **virtual** не требуется.

Виртуальная функция может вызываться так же, как и любая другая функция-член. Однако наиболее интересен вызов виртуальной функции через указатель, благодаря чему поддерживается динамический полиморфизм. Из предыдущего раздела вы знаете, что указатель базового класса можно использовать в качестве указателя на объект производного класса. Если указатель базового класса ссылается на объект производного класса, который содержит виртуальную функцию и для которого виртуальная функция вызывается через этот указатель, то компилятор определяет, какую версию виртуальной функции вызвать, основываясь при этом на *типе объекта, на который ссылается указатель.* При этом определение конкретной версии виртуальной функции имеет место не в процессе компиляции, а в процессе выполнения программы. Другими словами, тип объекта, на который ссылается указатель, и определяет ту версию виртуальной функции, которая будет выполняться. Поэтому, если два или более различных класса являются производными от базового, содержащего виртуальную функцию, то, если указатель базового класса ссылается на разные объекты этих производных

классов, выполняются различные версии виртуальной функции. Этот процесс является реализацией принципа динамического полиморфизма. Фактически, о классе, содержащем виртуальную функцию, говорят как о *полиморфном классе (polymorphic class).*

1. Рассмотрим короткий пример использования виртуальной функции:

```
// Простой пример использования виртуальной функции
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Выполнение функции func() базового класса: ";
        cout << i << '\n';
    }
};

class derived1 : public base {
public:
    derived1(int x) : base(x) { }
    void func()
    {
        cout << "Выполнение функции func() класса derived1: ";
        cout << i * i << '\n';
    }
};

class derived2: public base {
public: .
    derived2(int x) : base(x) { }
    void func()
    {
        cout << "Выполнение функции func() класса derived2: ";
        cout << i + i << '\n';
    }
};

int main()
{
    base *p;
```

```
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);

    p = &ob;
    p->func(); // функция func() класса base

    p = &d_ob1;
    p->func(); // функция func() производного класса derived1

    p = &d_ob2;
    p->func(); /У функция func() производного класса derived2

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Выполнение функции func() базового класса: 10
Выполнение функции func() класса derived1: 100
Выполнение функции func() класса derived2: 20
```

Переопределение виртуальной функции внутри производного класса может показаться похожим на перегрузку функций. Однако эти два процесса совершенно различны. Во-первых, перегружаемая функция должна отличаться типом и/или числом параметров, а переопределяемая виртуальная функция должна иметь точно такой же тип параметров, то же их число, и такой же тип возвращаемого значения. (На самом деле, если при переопределении виртуальной функции вы изменяете число или тип параметров, она просто становится перегружаемой функцией и ее виртуальная природа теряется.) Далее, виртуальная функция должна быть членом класса. Это не относится к перегружаемым функциям. Кроме этого, если деструкторы могут быть виртуальными, то конструкторы нет. Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными функциями, для описания переопределения виртуальной функции используется термин *подмена (overriding)*.

В рассмотренном примере создается три класса. В классе **base** определяется виртуальная функция **func()**. Затем этот класс наследуется двумя производными классами: **derivedl и derived2**. Каждый из этих классов переопределяет функцию **func()** по-своему. Внутри функции **main()** указатель базового класса p поочередно ссылается на объекты типа **base, derivedl и derived2**. Первым указателю p присваивается адрес объекта **ob** (объекта типа **base).** При вызове функции **func()** через указатель p используется ее версия из класса **base.** Следующим указателю p присваивается адрес объекта **d_obl** и функция **func()** вызывается снова. Поскольку версия вызываемой виртуальной функции определяется типом объекта, на который ссылается указатель, то вызывается та версия функции, которая переопределяется в классе **derivedl.** Наконец, указателю p присваивается адрес объекта **d_ob2,** и снова вызывается функция **func().** При этом выполняется та версия функции func(), которая определена внутри класса **derived2**.

Ключевым для понимания предыдущего примера является тот факт, что, во-первых, тип адресуемого через указатель объекта определяет вызов той или иной версии подменяемой виртуальной функции, во-вторых, выбор конкретной версии происходит уже в процессе выполнения программы.

2. Виртуальные функции имеют иерархический порядок наследования. Кроме того, если виртуальная функция *не* подменяется в производном классе, то используется версия функции, определенная в базовом классе. Например, ниже приведена слегка измененная версия предыдущей программы:

```
// Иерархический порядок виртуальных функций
#include <iostream>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Выполнение функции func() базового класса: ";
        cout << i << '\n';
    }
};

class derived1: public base {
public:
    derived1(int x): base(x) { }
    void func()
    {
        cout << "Выполнение функции func() класса derived1: ";
        cout << i * i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) { }
    // в классе derived2 функция func() не подменяется
};

int main()
{
    base *p;
    base ob(10);
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    p = &ob;
    p->func(); // функция func() базового класса
```

```
    p = &d_ob1;
    p->func(); // функция func() производного класса derivedl

    p = &d_ob2;
    p->func(); // функция func() базового класса

    return 0;
}
```

После выполнения программы на экран выводится следующее:

```
Выполнение функции func О базового класса: 10
Выполнение функции func() класса derivedl: 100
Выполнение функции func() базового класса: 10
```

В этой программе в классе derived2 функция func() не подменяется. Когда указателю p присваивается адрес объекта d_ob2 и вызывается функция func(), используется версия функции из класса base, поскольку она следующая в иерархии классов. Обычно, если виртуальная функция не переопределена в производном классе, используется ее версия из базового класса.

3. **В следующем примере показано, как случайные события во время работы программы влияют на вызываемую версию виртуальной функции. Программа выбирает между объектами d_ob1 и d_ob2 на основе значений, возвращаемых стандартным генератором случайных чисел rand(). Запомните, выбор конкретной версии функции func() происходит во время работы программы. (Действительно, при компиляции этот выбор сделать невозможно, поскольку он основан на значениях, которые можно получить только во время работы программы.)**

```
/* В этом примере показана работа виртуальной функции при наличии
случайных событий во время выполнения программы.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

class base {
public:
    int i;
    base(int x) { i = x; }
    virtual void func()
    {
        cout << "Выполнение функции f unc ( ) базового класса : " ;
        cout << i << ' \n ' ;
    }
};

class derivedl: public base {
public:
    derivedl (intx) : base(x) { }
```

```
    void func()
    {
        cout << "Выполнение функции func() класса derived1: ";
        cout << i * i << '\n';
    }
};

class derived2 : public base {
public:
    derived2(int x) : base(x) { }
    void func()
    {
        cout << "Выполнение функции func() класса derived2 : ";
        cout << i + i << ' \n ' ;
    }
};

int main()
{
    base *p;
    derived1 d_ob1(10);
    derived2 d_ob2(10);
    int i, j;

    for(i=0; i<10; i++) {
        j = rand() ;
        if((j%2)) p = &d_ob1; // если число нечетное
                              // использовать объект d_ob1
        else p = &d_ob2;      // если число четное
                              // использовать объект d_ob2
        p->func();            // вызов подходящей версии функции
    }

    return 0;
}
```

4. Теперь более реальный пример использования виртуальной функции. В этой программе создается исходный базовый класс **area,** в котором сохраняются две размерности фигуры. В нем также объявляется виртуальная функция **getarea()**, которая, при ее подмене в производном классе, возвращает площадь фигуры, вид которой задается в производном классе. В этом случае определение функции **getarea()** внутри базового класса задает интерфейс. Конкретная реализация остается тем классам, которые наследуют класс **area.** В этом примере рассчитывается площадь треугольника и прямоугольника.

```
// Использование виртуальной функции для определения интерфейса
#include <iostream>
using namespace std;
```

```
class area {
    double dim1, dim2; // размеры фигуры
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double Sd2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea()
    {
        cout << "Вам необходимо подменить эту функцию\n";
        return 0.0;
    }
};

class rectangle: public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle: public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);
```

```
    P = &r;
    cout << "Площадь прямоугольника: " << p->getarea() << '\n';

    P = &t;
    cout << "Площадь треугольника: " << p->getarea() << '\n';

    return 0;
}
```

Обратите внимание, что определение **getarea()** внутри класса **area** является только "заглушкой" и, в действительности, не выполняет никаких действий. Поскольку класс **area** не связан с фигурой конкретного типа, то нет значимого определения, которое можно дать функции **getarea()** внутри класса **area.** При этом, для того чтобы нести полезную нагрузку, функция **getarea()** должна быть переопределена в производном классе. В следующем разделе вы узнаете об этом подробнее.

1. Напишите программу создания базового класса пит. В этом классе должно храниться целое и определяться виртуальная функция **shownum()**. Создайте два производных класса **outhex и outoct,** которые наследуют класс num. Функция **shownum()** должна быть переопределена в производных классах так, чтобы осуществлять вывод на экран значений, в шестнадцатеричной и восьмеричной системах счисления соответственно.
2. Напишите программу, в которой базовый класс **dist** используется для хранения в переменной типа **double** расстояния между двумя точками. В классе **dist** создайте виртуальную функцию **trav_time()**, которая выводит на экран время, необходимое для прохождения этого расстояния с учетом того, что расстояние задано в милях, а скорость равна 60 миль в час. В производном классе **metric** переопределите функцию **trav_time()** так, чтобы она выводила на экран время, необходимое для прохождения этого расстояния, считая теперь, что расстояние задано в километрах, а скорость равна 100 километров в час.

10.3. Дополнительные сведения о виртуальных функциях

Как показано в примере 4 предыдущего раздела, иногда, когда виртуальная функция объявляется в базовом классе, она не выполняет никаких значимых действий. Это вполне обычная ситуация, поскольку часто в базовом классе законченный тип данных не определяется. Вместо этого в нем просто содержится базовый набор функций-членов и переменных, для которых в производном классе определяется все недостающее. Когда в виртуальной

функции базового класса отсутствует значимое действие, в любом классе, производном от этого базового, такая функция *обязательно* должна быть переопределена. Для реализации этого положения в C++ поддерживаются так называемые *чистые виртуальные функции (pure virtualfunction)*.

Чистые виртуальные функции не определяются в базовом классе. Туда включаются только прототипы этих функций. Для чистой виртуальной функции используется такая основная форма:

```
virtual тип имя_функции(список_параметров)= 0;
```

Ключевой частью этого объявления является приравнивание функции нулю. Это сообщает компилятору, что в базовом классе не существует тела функции. Если функция задается как чистая виртуальная, это предполагает, что она обязательно должна подменяться в *каждом* производном классе. Если этого нет, то при компиляции возникнет ошибка. Таким образом, создание чистых виртуальных функций — это путь, гарантирующий, что производные классы обеспечат их переопределение.

Если класс содержит хотя бы одну чистую виртуальную функцию, то о нем говорят как об *абстрактном классе (abstract class)*. Поскольку в абстрактном классе содержится, по крайней мере, одна функция, у которой отсутствует тело функции, технически такой класс неполон, и ни одного объекта этого класса создать нельзя. Таким образом, абстрактные классы могут быть только наследуемыми. Они никогда не бывают изолированными. Важно понимать, однако, что по-прежнему можно создавать указатели абстрактного класса, благодаря которым достигается динамический полиморфизм. (Также допускаются и ссылки на абстрактный класс.)

Если виртуальная функция наследуется, то это соответствует ее виртуальной природе. Это означает, что если производный класс используется в качестве базового для другого производного класса, то виртуальная функция может подменяться в последнем производном классе (так же, как и в первом производном классе). Например, если базовый класс B содержит виртуальную функцию f(), и класс **D1** наследует класс B, а класс **D2** наследует класс **D1,** тогда функция **f()** может подменяться как в классе **D1,** так и в классе **D2.**

1. Здесь представлена несколько усовершенствованная версия программы, показанной в примере 4 предыдущего раздела. В этой версии программы в базовом классе **area** функция **getarea()** объявляется как чистая виртуальная функция.

```
// Создание абстрактного класса
#include <iostream>
using namespace std;
```

```
class area {
    double dim1, dim2; // размеры фигуры
public:
    void setarea(double d1, double d2)
    {
        dim1 = d1;
        dim2 = d2;
    }
    void getdim(double &d1, double &d2)
    {
        d1 = dim1;
        d2 = dim2;
    }
    virtual double getarea() = 0; // чистая виртуальная функция
};

class rectangle: public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return d1 * d2;
    }
};

class triangle: public area {
public:
    double getarea()
    {
        double d1, d2;

        getdim(d1, d2);
        return 0.5 * d1 * d2;
    }
};

int main()
{
    area *p;
    rectangle r;
    triangle t;

    r.setarea(3.3, 4.5);
    t.setarea(4.0, 5.0);

    p = &r;
    cout << "Площадь прямоугольника: " << p->getarea() << '\n';
```

```
    P = &t;
    cout << "Площадь треугольника: " << p->getarea() << '\n';

    return 0;
}
```

Теперь то, что функция **getarea()** является чистой **виртуальной**, гарантирует ее обязательную подмену в каждом производном классе.

2. В следующей программе показано, как при наследовании сохраняется виртуальная природа функции:

```
/* Виртуальная функция при наследовании сохраняет свою виртуальную природу
*/
#include <iostream>
using namespace std;

class base {
public:
    virtual void func()
    {
        cout << "Выполнение функции func() базового класса \n";
    }
};

class derived1: public base {
public:
    void func()
    {
        cout << "Выполнение функции func() класса derived1\n";
    }
};

// Класс derived1 наследуется классом derived2
class derived2 : public derived1 {
public:
    void func()
    {
        cout << " Выполнение функции func() класса derived2\n";
    }
};

int main()
{
    base *p;
    base ob;
    derived1 d_ob1;
    derived2 d_ob2;
```

```
    p = &ob;
    p->func(); // функция func() базового класса

    p = &d_ob1;
    p->func(); // функция func() производного класса derived1

    p = &d_ob2;
    p->func(); // функция func() производного класса derived2

    return 0;
}
```

В этой программе виртуальная функция **func()** сначала наследуется классом **derivedl,** в котором она подменяется. Далее класс **derivedl** наследуется классом **derived2**. В классе **derived2** функция **func()** снова подменяется.

Поскольку виртуальные функции являются иерархическими, то если бы в классе **derived2** функция **func()** не подменялась, при доступе к объекту **d_ob!** использовалась бы переопределенная в классе **derivedl** версия функции func(). Если бы функция funcQ не подменялась ни в классе **derivedl,** ни в классе **derived2**, то все ссылки на функцию **func()** относились бы к ее определению в классе **base.**

1. Проведите эксперимент с двумя программами из предыдущих примеров. Попытайтесь создать объект, используя класс **area** из примера 1, и проанализируйте сообщение об ошибке. В примере 2 попытайтесь удалить переопределение функции **func()** внутри класса **derived2**. Убедитесь, что тогда действительно будет использоваться та версия функции funcQ, переопределение которой находится в классе **derivedl.**
2. Почему нельзя создать объект абстрактного класса?
3. Что произойдет в примере 2 при удалении переопределения функции func() из класса **derivedl?** Будет ли при этом программа компилироваться и запускаться? Если да, то почему?

10.4. Применение полиморфизма

Теперь, когда вы знаете, как использовать виртуальные функции для реализации динамического полиморфизма, самое время рассмотреть, зачем это нужно. Как уже много раз в этой книге отмечалось, полиморфизм является процессом, благодаря которому общий интерфейс применяется к двум или более схожим (но технически разным) ситуациям, т. е. реализуется философия "один интерфейс, множество методов". Полиморфизм важен потому,

что может сильно упростить сложные системы. Один хорошо определенный интерфейс годится для доступа к некоторому числу разных, но связанных по смыслу действий, и таким образом устраняется искусственная сложность. Уточним: полиморфизм позволяет сделать очевидной логическую близость схожих действий; поэтому программа становится легче для понимания и сопровождения. Если связанные действия реализуются через общий интерфейс, вам нужно гораздо меньше помнить.

Имеются два термина, которые часто ассоциируются с объектно-ориентированным программированием вообще и с C++ в частности. Этими терминами являются *раннее связывание (early binding)* и *позднее связывание (late binding)*. Важно понимать, что означают указанные термины. Раннее связывание относится к событиям, о которых можно узнать в процессе компиляции. Особенно это касается вызовов функций, которые настраиваются при компиляции. Функции раннего связывания — это "нормальные" функции, перегружаемые функции, невиртуальные функции-члены и дружественные функции. При компиляции функций этих типов известна вся необходимая для их вызова адресная информация. Главным преимуществом раннего связывания (и доводом в пользу его широкого использования) является то, что оно обеспечивает высокое быстродействие программ. Определение нужной версии вызываемой функции во время компиляции программы — это самый быстрый метод вызова функций. Главный недостаток — потеря гибкости.

Позднее связывание относится к событиям, которые происходят в процессе выполнения программы. Вызов функции позднего связывания — это вызов, при котором адрес вызываемой функции до запуска программы неизвестен. В C++ виртуальная функция является объектом позднего связывания. Если доступ к виртуальной функции осуществляется через указатель базового класса, то в процессе работы программа должна определить, на какой тип объекта он ссылается, а затем выбрать, какую версию подменяемой функции выполнить. Главным преимуществом позднего связывания является гибкость во время работы программы. Ваша программа может легко реагировать на случайные события. Его основным недостатком является то, что требуется больше действий для вызова функции. Это обычно делает такие вызовы медленнее, чем вызовы функций раннего связывания.

В зависимости от нужной эффективности, следует принимать решение, когда лучше использовать раннее связывание, а когда — позднее.

1. Ниже представлена программа, которая иллюстрирует принцип "один интерфейс, множество методов". В ней определен исходный базовый класс для связанного списка целых. Интерфейс списка определяется с помощью чистых виртуальных функций **store()** и **retrieve()**. Для хранения значения в списке вызывается функция store(). Для выборки значения из списка вызы-

вается функция **retrieve()**. В базовом классе **list** для выполнения этих действий никакого встроенного метода не задается. Вместо этого в каждом производном классе явно определяется, какой тип списка будет поддерживаться. В программе реализованы списки двух типов: очередь и стек. Хотя способы работы с этими двумя списками совершенно различны, для доступа к каждому из них применяется один и тот же интерфейс. Вам следует тщательно изучить эту программу.

```
// Демонстрация виртуальных функций
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
    list *head; // указатель на начало списка
    list *tail; // указатель на конец списка
    list *next; // указатель на следующий элемент списка
    int num;    // число для хранения

    list() { head = tail = next = NULL; }
    virtual void store(int i) = 0;
    virtual int retrieve() = 0;
};

// Создание списка типа очередь
class queue: public list {
public:
    void store(int i);
    int retrieve();
};

void queue::store(int i)
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item->num = i;

    // добавление элемента в конец списка
    if(tail) tail->next = item;
    tail = item;
    item->next = NULL;
    if(!head) head = tail;
}
```

```
int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }

    // удаление элемента из начала списка
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

// Создание списка типа стек
class stack: public list {
public:
    void store(int i);
    int retrieve();
};

void stack::store(int i)
{
    list *item;

    item = new stack;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item->num = i;

    // добавление элемента в начало списка
    if(head) item->next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
```

```
        return 0;
    }

    // удаление элемента из начала списка
    i = head->num;
    p = head;
    head = head->next;
    delete p;

    return i;
}

int main()
{
    list *p;

    // демонстрация очереди
    queue q_ob;
    p = &q_ob; // указывает на очередь

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Очередь: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    // демонстрация стека
    stack s_ob;
    p = &s_ob; // указывает на стек

    p->store(1);
    p->store(2);
    p->store(3);

    cout << "Стек: ";
    cout << p->retrieve();
    cout << p->retrieve();
    cout << p->retrieve();

    cout << '\n';

    return 0;
}
```

2. **Функция main() в программе со списками только иллюстрирует работу классов. Однако для изучения динамического полиморфизма попробуйте использовать в предыдущей программе следующую функцию main():**

```
int main()
{
    list *p;
    stack s_ob;
    queue q_ob;
    char ch;
    int i;

    for(i=0; i<10; i++) {
        cout << "Стек или Очередь? (C/O): ";
        cin>>ch;
        ch = tolower(ch);
        if(ch=='o') p = &q_ob;
        else p = &s_ob;
        p->store(i);
    }

    cout << "Введите К для завершения работы\n";
    for(;;) {
        cout << "Извлечь элемент из Стека или Очереди? (C/O): ";
        cin>>ch;
        ch = tolower(ch);
        if(ch=='к') break;
        if(ch=='o') p = &q_ob;
        else p = &s_ob;
        cout << p->reteieve() << '\n';
    }

    cout << '\n';

    return 0;
}
```

Функция **main()** показывает, как случайные события, возникающие при выполнении программы, могут быть легко обработаны, если использовать виртуальные функции и динамический полиморфизм. В программе выполняется цикл **for** от 0 до 9. В течение каждой итерации предлагается выбор типа списка — стек или очередь. В соответствии с ответом, базовый указатель p устанавливается на выбранный объект (очередь или стек), и это значение запоминается. После завершения цикла начинается другой цикл, в котором предлагается выбрать список для извлечения запомненного значения. В соответствии с ответом пользователя выбирается число из указанного списка.

Несмотря на то, что этот пример достаточно прост, он позволяет понять, как полиморфизм может упростить программу, которой необходимо реагировать на случайные события. Например, операционная система Windows взаимодействует с программой посредством сообщений. Эти сообщения генерируются как случайные, и ваша программа должна как-то реагировать на каждое

получаемое сообщение. Одним из возможных способов обработки этих сообщений и является использование чистых виртуальных функций.

Упражнения

1. Добавьте список другого типа к программе из примера 1. Эта версия должна поддерживать отсортированный (в порядке возрастания) список. Назовите список **sorted.**
2. Обдумайте случаи, в которых следует использовать динамический полиморфизм, чтобы упростить решение разного рода проблем.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Что такое виртуальная функция?
2. Какие функции не могут быть виртуальными?
3. Как виртуальные функции помогают реализовывать динамический полиморфизм? Ответьте подробно.
4. Что такое чистая виртуальная функция?
5. Что такое абстрактный класс? Что такое полиморфный класс?
6. Правилен ли следующий фрагмент? Если нет, то почему?

```
class base {
public:
    virtual int f(int a) = 0;
    // ...
};

class derived: public base {
public:
    int f(int a, int b) { return a*b; }
    // ...
};
```

7. Наследуется ли виртуальность?
8. Поэкспериментируйте с виртуальными функциями. Это важное понятие и необходимо освоить технические приемы, связанные с ним.

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Расширьте пример со списком, пример 1 из раздела 4 так, чтобы в нем перегружались операторы + и --. Используйте оператор + для внесения элемента в список, а оператор — для выборки элемента из списка.
2. Что отличает виртуальные функции от перегружаемых функций?
3. Вернитесь к представленным ранее в книге примерам перегрузки функций. Определите, какие из этих функций можно превратить в виртуальные. Кроме этого, подумайте, как с помощью виртуальных функций решить ваши собственные программные задачи.

Глава 11

Шаблоны и обработка исключительных ситуаций

В этой главе вы познакомитесь с двумя важнейшими характеристиками C++ верхнего уровня: *шаблонами (templates)* и *обработкой исключительных ситуаций (exception handling)*. Ни та, ни другая характеристики не входили в изначальную спецификацию C++, а были добавлены туда несколько лет назад и определены в стандарте Standard C++. Эти характеристики поддерживаются всеми современными компиляторами и позволяют достичь двух наиболее заманчивых целей программирования: создания многократно используемых и отказоустойчивых программ.

С помощью шаблонов можно создавать родовые функции (generic functions) и родовые классы (generic classes). В родовой функции или классе тип данных, с которыми функция или класс работают, задается в качестве параметра. Это позволяет одну и ту же функцию или класс использовать с несколькими различными типами данных без необходимости программировать новую версию функции или класса для каждого конкретного типа данных. Таким образом шаблоны дают возможность создавать многократно используемые программы. В данной главе рассказывается как о родовых функциях, так и о родовых классах.

Система обработки исключительных ситуаций встроена в C++ и позволяет работать с ошибками, которые возникают во время работы программы, заранее предусмотренным и управляемым образом. С помощью системы обработки исключительных ситуаций C++ ваша программа при возникновении ошибки может автоматически вызвать процедуру ее обработки. Принципиальное преимущество обработки исключительных ситуаций состоит в том, что она автоматически в зависимости от ситуации запускает одну из множества подпрограмм обработки ошибок, которые предварительно "вручную" встраиваются в основную программу. Должным образом запрограммированная обработка исключительных ситуаций помогает создавать действительно отказоустойчивые программы.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Что такое виртуальная функция?
2. Что такое чистая виртуальная функция? Если в объявлении класса имеется чистая виртуальная функция, как называется такой класс и какие ограничения налагаются на его использование?
3. Динамический полиморфизм достигается посредством использования _____функций и указателей_____класса. (Вставьте пропущенные слова.)
4. Если при наличии иерархии классов в производном классе опущена перегрузка (не чистой) виртуальной функции, что происходит, когда объект этого производного класса вызывает такую функцию?
5. В чем основное преимущество динамического полиморфизма? Каков его потенциальный недостаток?

11.1. Родовые функции

Родовая функция определяет базовый набор операций, которые будут применяться к разным типам данных. Родовая функция оперирует с тем типом данных, который она получает в качестве параметра. С помощью этого механизма одна и та же процедура может применяться к самым разным данным. Как известно, многие алгоритмы логически одинаковы, независимо от того, для обработки каких типов данных они предназначены. Например, алгоритм быстрой сортировки одинаков как для массивов целых, так и для массивов действительных чисел. Это именно тот случай, когда сортируемые данные отличаются только по типам. Благодаря созданию родовой функции вы можете независимо от типа данных определить суть алгоритма. После того как это сделано, компилятор автоматически генерирует правильный код для фактически используемого при выполнении функции типа данных. По существу, при создании родовой функции вы создаете функцию, которая может автоматически перегружаться сама.

Родовая функция создается с помощью ключевого слова **template.** Обычное значение этого слова (т. е. шаблон) точно отражает его назначение в C++. Оно предназначено для создания шаблона (или каркаса), который описывает то, что будет делать функция, при этом компилятору остается дополнить каркас необходимыми деталями. Ниже представлена типовая форма определения функции-шаблона:

```
template <class Фтип> возвр_значение имя_функции(список_параметров)
{
    // тело функции
}
```

Здесь вместо ***Фтип*** указывается тип используемых функцией данных. Это имя можно указывать внутри определения функции. Однако это всего лишь фиктивное имя, которое компилятор автоматически заменит именем реального типа данных при создании конкретной версии функции.

Хотя традиционно для задания родового типа данных в объявлении шаблона указывается ключевое слово **class,** вы также можете использовать ключевое слово **typename.**

1. В следующей программе создается родовая функция, которая меняет местами значения двух переменных, передаваемых ей в качестве параметров. Поскольку в своей основе процесс обмена двух значений не зависит от типа переменных, этот процесс удачно реализуется с помощью родовой функции.

```
// Пример родовой функции или шаблона
#include <iostream>
using namespace std;

// Это функция-шаблон
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;

  cout << "Исходные значения i, j равны: " << i << ' ' << j << endl;
  cout << "Исходные значения x, y равны: " << x << ' ' << y << endl;

    swapargs (i, j);      // обмен целых
    swapargs (x, y);      // обмен действительных

    cout << "Новые значения i, j равны: " << i << ' ' << j << endl;
    cout << "Новые значения x, y равны: " << x << ' ' << y << endl;

    return 0;
}
```

Ключевое слово **template** используется для определения родовой функции. Строка:

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору две вещи: во-первых, создается шаблон, и, во-вторых, начинается определение родовой функции. Здесь X — это родовой тип данных, используемый в качестве фиктивного имени. После строки с ключевым

словом template функция **swapargs()** объявляется с именем X в качестве типа данных обмениваемых значений. В функции main() функция **swapargs()** вызывается с двумя разными типами данных: целыми и действительными. Поскольку функция **swapargs()** — это родовая функция, компилятор автоматически создает две ее версии: одну — для обмена целых значений, другую для обмена действительных значений. Теперь попытайтесь скомпилировать программу.

Имеются и другие термины, которые можно встретить при описании шаблонов в литературе по C++. Во-первых, родовая функция (то есть функция, в определении которой имеется ключевое слово template) также называется *функция-шаблон (template Junction).* Когда компилятор создает конкретную версию этой функции, говорят, что он создал *порожденную функцию (generatedfunction).* Процесс генерации порожденной функции называют *созданием экземпляра (instantiating)* функции. Другими словами, порожденная функция — это конкретный экземпляр функции-шаблона.

2. Ключевое слово template в определении родовой функции не обязательно должно быть в той же строке, что и имя функции. Например, ниже приведен еще один вполне обычный формат определения функции **swapargs()**:

```
template <class X>
void swapargs (X sa, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

При использовании такого формата важно понимать, что никаких других инструкций между инструкцией template и началом определения родовой функции быть не может. Например, следующий фрагмент программы компилироваться не будет:

```
// Этот фрагмент компилироваться не будет
template <class X>
int i;                          // это неправильно
void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

Как указано в комментариях, инструкция с ключевым словом template должна находиться сразу перед определением функции.

3. Как уже упоминалось, для задания родового типа данных в определении шаблона вместо ключевого слова **class** можно указывать ключевое слово **typename**. Например, ниже приведено еще одно объявление функции **swapargs()**:

```
// Использование ключевого слова typename
template <typename X> void swapargs(X sa, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

Ключевое слово **typename** можно также указывать для задания неизвестного типа данных внутри шаблона, но такое его использование выходит за рамки данной книги.

4. С помощью инструкции **template** можно определить более одного родового типа данных, отделяя их друг от друга запятыми. Например, в данной программе создается родовая функция, в которой имеются два родовых типа данных:

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << endl;
}

int main()
{
    myfunc(10, "hi");

    myfunc(0.23, 10L);

    return 0;
}
```

В данном примере при генерации конкретных экземпляров функции **myfunc()**, фиктивные имена типов **type1 и type2** заменяются компилятором на типы данных **int** и **char*** или **double и long** соответственно.

Когда вы создаете родовую функцию, то по существу предписываете компилятору генерировать столько разных версий этой функции, сколько нужно для обработки всех способов вызова этой функции в вашей программе.

5. Родовые функции похожи на перегруженные функции за исключением того, что они более ограничены по своим возможностям. При перегрузке функции внутри ее тела можно выполнять совершенно разные действия. С другой стороны, родовая функция должна выполнять одни и те же базовые действия для всех своих версий. Например, следующие перегруженные функции *нельзя* заменить на родовую функцию, поскольку они делают не одно и то же.

```
void outdata(int i)
{
    cout << i;
}

void outdata(double d)
{
    cout << setprecision(10) << setfill('#');
    cout << d;
    cout << setprecision(6) << setfill(' ');
}
```

6. Несмотря на то, что функция-шаблон при необходимости перегружается сама, ее также можно перегрузить явно. Если вы сами перегружаете родовую функцию, то перегруженная функция подменяет (или "скрывает") родовую функцию, которую бы создал компилятор для этой конкретной версии. Рассмотрим такой вариант примера 1:

```
// Подмена функции-шаблона
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}

// Здесь переопределяется родовая версия функции swapargs()
void swapargs(int a, int b)
{
    cout << "это печатается внутри функции swapargs(int, int)\n";
}

int main()
{
    int i = 10, j = 20;
    float x = 10.1, y = 23.3;

  cout << "Исходные значения i, j равны: " << i << ' ' << j << endl;
  cout << "Исходные значения x, y равны: " << x << ' ' << y << endl;
```

```
    swapargs(i, j);     // вызов явно перегруженной функции swapargs()
    swapargs(x, y);     // обмен действительными числами

    cout << "Новые значения i, j равны: " << i << ' ' << j << endl;
    cout << "Новые значения x, y равны: " << x << ' ' << y << endl;

    return 0;
}
```

Как отмечено в комментариях, при вызове функции **swapargs(i,** j) вызывается определенная в программе явно перегруженная версия функции **swapargs()**. Таким образом, компилятор не генерирует этой версии родовой функции **swapargs()**, поскольку родовая функция подменяется явно перегруженной функцией.

Ручная перегрузка шаблона, как показано в данном примере, позволяет изменить версию родовой функции так, чтобы приспособить ее к конкретной ситуации. Однако в большинстве случаев, если вам нужно несколько разных версий функции для разных типов данных, вместо шаблонов лучше использовать перегруженные функции.

Упражнения

1. Если этого еще не сделано, попытайтесь откомпилировать каждый из предыдущих примеров.

2. Напишите родовую функцию **min(),** возвращающую меньший из двух своих аргументов. Например, версия функции min(3, 4) должна возвратить 3, а версия **min('c', 'a')** — a. Продемонстрируйте работу функции с помощью программы.

3. Прекрасным кандидатом на функцию-шаблон является функция **find()**. Эта функция ищет объект в массиве. Она возвращает либо индекс найденного объекта (если его удалось найти), либо −1, если заданный объект не найден. Ниже представлен прототип конкретной версии функции **find()**. Переделайте функцию **find()** в родовую функцию и проверьте ваше решение в программе. (Параметр size задает количество элементов массива.)

```
int find(int object, int *list, int size)
{
    // ...
}
```

4. Объясните своими словами, зачем нужны родовые функции и как они могут упростить исходный код ваших программ.

11.2. Родовые классы

В дополнение к родовым функциям можно определить и родовые классы. При этом создается класс, в котором определены все необходимые алгоритмы, а фактические типы обрабатываемых данных задаются в качестве параметров позже, при создании объектов этого класса.

Родовые классы полезны, когда класс содержит общую логику работы. Например, алгоритм, который реализует очередь целых, будет также работать и с очередью символов. Кроме того, механизм, который реализует связанный список почтовых адресов, будет также поддерживать связанный список запасных частей к автомобилям. С помощью родового класса можно создать класс, реализующий очередь, связанный список и т. д. для любых типов данных. Компилятор будет автоматически генерировать правильный тип объекта на основе типа, заданного при создании объекта.

Ниже представлена основная форма объявления родового класса:

```
template <class Фтип> class имя_класса {
.
.
.
}
```

Здесь ***Фтип*** — это фиктивное имя типа, который будет задан при создании экземпляра класса. При необходимости можно определить более одного родового типа данных, разделяя их запятыми.

После создания родового класса с помощью представленной ниже формы можно создать конкретный экземпляр этого класса:

```
имя_класса <тип> объект;
```

Здесь *тип* — это имя типа данных, с которым будет оперировать класс.

Функции-члены родового класса сами автоматически становятся родовыми. Для них не обязательно явно задавать ключевое слово **template.**

Как вы увидите в главе 14, в C++ имеется встроенная библиотека классов-шаблонов, которая называется библиотекой стандартных шаблонов (Standard Template Library, STL). Эта библиотека предоставляет родовые версии классов для наиболее часто используемых алгоритмов и структур данных. Чтобы научиться пользоваться библиотекой стандартных шаблонов с максимальной эффективностью, вам необходимо иметь твердые знания по классам-шаблонам и их синтаксису.

1. В следующей программе создается очень простой родовой класс, реализующий односвязный список. Затем демонстрируются возможности такого класса путем создания связанного списка для хранения символов.

```
// Простой родовой связанный список
#include <iostream>
using namespace std;

template <class data_t> class list {
    data_t data;
    list *next;
public:
    list (data_t d) ;
    void add (list *node) { node->next = this; next = 0; }
    list *getnext() { return next; }
    data_t getdata() { return data; }
}

template <class data_t> list<data_t>: : list (data_t d)
{
    data = d;
    next = 0;
}

int main()
{
    list<char> start ('a');
    list<char> *p, *last;
    int i;

    // создание списка
    last - &start;
    for(i=1; i<26; i++) {
        p = new list<char> ('a' + i) ;
        p->add(last);
        last = p;
    }

    // вывод списка
    p = &start;
    while (p) {
        cout << p->getdata ( ) ;
        P = p->getnext ( ) ;
    }

    return 0;
}
```

Как видите, объявление родового класса похоже на объявление родовой функции. Тип данных, хранящихся в списке, становится родовым в объявлении класса. Но он не проявляется, пока не объявлен объект, который и задает реальный тип данных. В данном примере объекты и указатели создаются внутри функции **main()**, где указывается, что типом хранящихся в списке данных является тип **char**. Обратите особое внимание на следующее объявление:

```
list<char> start ('a');
```

Отметьте, что необходимый тип данных задается между угловыми скобками.

Наберите и выполните эту программу. В ней создается связанный список с символами алфавита, который затем выводится на экран. Путем простого изменения типа данных, который указывается при создании объектов, можно изменить тип данных, хранящихся в списке. Например, с помощью следующего объявления можно создать другой объект, где можно было бы хранить целые:

```
list<int> int_start(1);
```

Можно также использовать список **list** для хранения создаваемых вами типов данных. Например, для хранения адресной информации можно воспользоваться следующей структурой:

```
struct addr {
    char name[40];
    char street[40];
    char city[30];
    char state[3];
    char zip[12];
}
```

Теперь, чтобы с помощью списка **list** хранить объекты типа **addr,** используйте такое объявление (предположим, что объект **structvar** содержит правильную структуру **addr**):

```
list<addr> obj(structvar);
```

2. Ниже представлен другой пример родового класса. Это переработанный класс **stack,** впервые приведенный в главе 1. Однако в данном случае класс **stack** реализован как шаблон. Следовательно, в нем можно хранить объекты любого типа. В представленном ниже примере создаются стек символов и стек действительных чисел:

```
// Здесь показан родовой стек
ttinclude <iostream>
using namespace std;

ttdefine SIZE 10

// Создание родового класса stack
template <class StackType> class stack {
    StackType stck[SIZE];    // содержит стек
    int tos;                 // индекс вершины стека

public:
    void init() { tos = 0; } // инициализация стека
    void push(StackType ch); // помещает объект в стек
    StackType pop();         // выталкивает объект из стека
};
```

```
// Помещение объекта в стек
template <class StackType>
void stack<StackType>::push(StackType ob)
{
    if (tos==SIZE) {
        cout << "Стек полон";
        return;
    }
    stck[tos] = ob;
    tos++;
}

// Выталкивание объекта из стека
template <class StackType>
StackType stack<StackType>::pop()
{
    if (tos==0) {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // Демонстрация символьных стеков
    stack<char> s1, s2; // создание двух стеков
    int i;

    // инициализация стеков
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Из стека 1:" << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Из стека 2:" << s2.pop() << "\n";

    // Демонстрация стеков со значениями типа double
    stack<double> ds1, ds2; // создание двух стеков

    // инициализация стеков
    ds1.init();
    ds2.init();
```

```
    ds1.push(1.1);
    ds2.push(2.2);
    ds1.push(3.3);
    ds2.push(4.4);
    ds1.push(5.5);
    ds2.push(6.6);

    for(i=0; i<3; i++) cout << " Из стека 1:" << ds1.pop() << "\n";
    for(i=0; i<3; i++) cout << " Из стека 2:" << ds2.pop() << "\n";

    return 0;
}
```

Как показано на примере класса **stack** (и предыдущего класса **list),** родовые функции и классы обеспечивают мощный инструмент экономии времени при программировании, поскольку они позволяют определить общую форму алгоритма, который затем можно использовать с данными любого типа. Таким образом, вы избегаете однообразных операций по созданию отдельных процедур для каждого типа данных, с которыми должен работать ваш алгоритм.

3. Класс-шаблон может иметь более одного родового типа данных. Просто объявите все необходимые для класса типы данных внутри спецификации **template,** перечислив их через запятую. Например, в следующем коротком примере создается класс с двумя родовыми типами данных:

```
// Здесь в определении класса используется два родовых типа данных
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
    Type1 i;
    Type2 j;
public:
    myclass(Type1 a, Type2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Это проверка");

    ob1.show();     // вывод значений типа int и double
    ob2.show();     // вывод значений типа char и char *
}
```

После выполнения программы на экран выводится следующая информация:

```
10 0.23
X Это проверка
```

В программе объявлено два типа объектов. В объекте **ob1** используются целое и значение двойной точности. В объекте **ob2** — символ и указатель на символ. В обоих случаях компилятор автоматически генерирует необходимые данные и функции в соответствии со способом создания объектов.

1. Если этого еще не сделано, откомпилируйте и выполните два примера программ с родовыми классами. Попытайтесь объявлять списки и/или стеки для разных типов данных.
2. Создайте и продемонстрируйте родовой класс, реализующий очередь.
3. Создайте родовой класс **input,** который при вызове конструктора делает следующее:
   - выводит на экран строку-приглашение,
   - получает данные от пользователя,
   - повторно выводит на экран строку-приглашение, если вводимые данные не соответствуют заданному диапазону.

   Объекты типа **input** должны объявляться следующим образом:

```
input ob("строка_приглашение", мин_значение, макс_значение)
```

   Здесь ***строка_приглашение*** — это сообщение, появляющееся на экране в качестве приглашения для ввода. Минимальное и максимальное допустимые значения задаются с помощью параметров ***мин_значение*** **и** ***макс_значение*** соответственно. (Тип данных, вводимых пользователем, будет тем же самым, что и тип значений ***мин_значение*** и ***макс_значение***.)

11.3. Обработка исключительных ситуаций

C++ обеспечивает встроенный механизм обработки ошибок, называемый *обработкой исключительных ситуаций (exception handling).* Благодаря обработке исключительных ситуаций можно упростить управление и реакцию на ошибки во время выполнения программ. Обработка исключительных ситуаций в C++ организуется с помощью трех ключевых слов: **try, catch и throw.** В самых общих словах, инструкции программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке try. Если исключительная ситуация (т. е. ошибка) имеет место внутри блока **try**, она возбуждается (ключевое слово **throw),** перехватывается (ключевое слово **catch)** и обрабатывается. Ниже поясняется приведенное здесь общее описание.

Как уже отмечалось, любая инструкция, которая возбуждает исключительную ситуацию, должна выполняться внутри блока **try**. (Функции, которые вызываются из блока **try** также могут возбуждать исключительную ситуацию.) Любая исключительная ситуация должна перехватываться инструкцией **catch,** которая располагается непосредственно за блоком try, возбуждающем исключительную ситуацию. Далее представлена основная форма инструкций **try** и **catch:**

```
try {
    // блок возбуждения исключительной ситуации
}
catch (type1 arg) {
    // блок перехвата исключительной ситуации
}
catch (type2 arg) {
    // блок перехвата исключительной ситуации
}
catch (type3 arg) {
    // блок перехвата исключительной ситуации
}
.
.
.
catch (typeN arg) {
    // блок перехвата исключительной ситуации
}
```

Блок **try** должен содержать ту часть вашей программы, в который вы хотите отслеживать ошибки. Это могут быть как несколько инструкций внутри одной функции, так и все инструкции внутри функции **main()** (что естественно ведет к отслеживанию ошибок во всей программе).

После того как исключительная ситуация возбуждена, она перехватывается соответствующей этой конкретной исключительной ситуации инструкцией **catch,** которая ее обрабатывает. С блоком try может быть связано более одной инструкции **catch.** То, какая именно инструкция **catch** используется, зависит от типа исключительной ситуации. То есть, если тип данных, указанный в инструкции **catch,** соответствует типу исключительной ситуации, выполняется данная инструкция **catch.** При этом все оставшиеся инструкции блока try игнорируются (т. е. сразу после того, как какая-то инструкция в блоке try вызвала появление исключительной ситуации, управление передается соответствующей инструкции **catch,** минуя оставшиеся инструкции блока try, — *примеч. пер.*). Если исключительная ситуация перехвачена, аргумент ***arg*** получает ее значение. Если вам не нужен доступ к самой исключительной ситуации, можно в инструкции **catch** указать только ее тип *type,* аргумент ***arg*** указывать не обязательно. Можно перехватывать

любые типы данных, включая и типы создаваемых вами классов. Фактически в качестве исключительных ситуаций часто используются именно типы классов.

Далее представлена основная форма инструкции **throw:**

throw *исключительная_ситуация;*

Инструкция **throw** должна выполняться либо внутри блока try, либо в любой функции, которую этот блок вызывает (прямо или косвенно). Здесь ***исключительная_ситуация*** — это возбуждаемая инструкцией **throw** исключительная ситуация.

Если вы возбуждаете исключительную ситуацию, для которой нет соответствующей инструкции **catch,** может произойти ненормальное завершение программы. Если ваш компилятор функционирует в соответствии со стандартом Standard C++, то возбуждение необрабатываемой исключительной ситуации приводит к вызову стандартной библиотечной функции **terminate()**. По умолчанию для завершения программы функция **terminate()** вызывает функцию **abort()**, однако при желании можно задать собственную процедуру завершения программы. За подробностями обращайтесь к справочной документации вашего компилятора.

1. В следующем очень простом примере показано, как в C++ функционирует система обработки исключительных ситуаций:

```
// Простой пример обработки исключительной ситуации
#include <iostream>
using namespace std;

int main()
{
    cout << "начало\n";

    try {          // начало блока try
        cout << "Внутри блока try\n";
        throw 10;   // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
    }
    catch (int i) { // перехват ошибки
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }

    cout << "конец";

    return 0;
}
```

После выполнения программы на экран будет выведено следующее:

```
начало
Внутри блока try
Перехвачена ошибка номер: 10
конец
```

Внимательно изучите программу. Как видите, здесь имеется блок **try,** содержащий три инструкции, и инструкция **catch (int i),** обрабатывающая исключительную ситуацию целого типа. Внутри блока **try** будут выполнены только две из трех инструкций — инструкции **cout и throw.** После того как исключительная ситуация возбуждена, управление передается выражению **catch** и выполнение блока **try** завершается. Таким образом, инструкция **catch** вызывается *не* явно, управление выполнением программы просто передается этой инструкции. (Для этого стек автоматически сбрасывается.) Следовательно, следующая за инструкцией **throw** инструкция **cout** не будет выполнена никогда.

После выполнения инструкции **catch,** управление программы передается следующей за ней инструкции. Тем не менее, обычно блок **catch** заканчивают вызовом функции **exit(), abort()** или какой-либо другой функции, принудительно завершающей программу, поскольку, как правило, система обработки исключительных ситуаций предназначена для обработки катастрофических ошибок.

2. Как уже упоминалось, тип исключительной ситуации должен соответствовать типу, заданному в инструкции **catch.** Например, в предыдущем примере, если изменить тип данных в инструкции **catch** на **double,** то исключительная ситуация не будет перехвачена и будет иметь место ненормальное завершение программы. Это продемонстрировано в следующем фрагменте:

```
// Этот пример работать не будет
#include <iostream>
using namespace std;

int main()
{
    cout << "начало\n";

    try {              // начало блока try
        cout << "Внутри блока try\n";
        throw 10;      // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
    }
    catch (double i) { // Эта инструкция не будет работать
                       // с исключительной ситуацией целого типа
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }

    cout << "конец";

    return 0;
}
```

Поскольку исключительная ситуация целого типа не будет перехвачена инструкцией catch типа double, на экран программа выведет следующее:

```
начало
Внутри блока try
Abnormal program termination
```

3. **Исключительная ситуация может быть возбуждена не входящей в блок try инструкцией, если сама эта инструкция входит в функцию, которая вызывается из блока try. Например, ниже представлена совершенно правильная программа:**

```
/* Возбуждение исключительной ситуации из функции, находящейся вне
блока try
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
    cout << "Внутри функции Xtest, test равно: " << test << "\n";
    if(test) throw test;
}

int main()
{
    cout << "начало\n";

    try {           // начало блока try
        cout << "Внутри блока try\n";
        Xtest(0);
        Xtest(1);
        Xtest(2);
    }
    catch (int i) { // перехват ошибки
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Внутри блока try
Внутри функции Xtest, test равно: 0
Внутри функции Xtest, test равно: 1
```

```
Перехвачена ошибка номер: 1
конец
```

4. Блок try можно располагать внутри функции. В этом случае при каждом входе в функцию обработчик исключительной ситуации устанавливается снова. Например, рассмотрим следующую программу:

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

// Блоки try и catch могут находиться не только в функции main()
void Xhandler(int test)
{
    try {
        if(test) throw test;
    }
    catch(int i) {
        cout << "перехвачена ошибка номер: " << i << '\n';
    }
}

int main()
{
    cout << "начало\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка номер: 1
Перехвачена ошибка номер: 2
Перехвачена ошибка номер: 3
конец
```

Как видите, обработаны три исключительные ситуации. После вызова каждой исключительной ситуации функция возвращает свое значение. При повторном вызове функции обработчик исключительной ситуации устанавливается вновь.

5. Как упоминалось ранее, с блоком **try** можно связать более одной инструкции **catch.** Как правило, так и делается. При этом каждая инструкция **catch** пред-

назначена для перехвата своего типа исключительной ситуации. Например, в следующей программе перехватываются две исключительных ситуации, одна для целых и одна для строки:

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

// Можно перехватывать разные типы исключительных ситуаций
void Xhandler(int test)
{
    try {
        if(test) throw test;
        else throw "Значение равно нулю";
    }
    catch(int i) {
        cout << "Перехвачена ошибка номер: " << i << '\n';
    }
    catch(char *str) {
        cout << "Перехвачена строка: ";
        cout << str << '\n';
    }
}

int main()
{
    cout << "начало\n";

    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка номер: 1
Перехвачена ошибка номер: 2
Перехвачена строка: Значение равно нулю
Перехвачена ошибка номер: 3
конец
```

Как видите, каждая инструкция **catch** перехватывает только исключительные ситуации соответствующего ей типа.

Обычно выражения инструкций **catch** проверяются *в* том порядке, в котором они появляются в программе. Выполняется только та инструкция, которая

совпадает по типу данных с исключительной ситуацией. Все остальные блоки **catch** игнорируются.

1. Лучший способ понять, как функционирует система обработки исключительных ситуаций в C++ — это поработать с ней. Введите, откомпилируйте и запустите предыдущие примеры программ. Затем поэкспериментируйте с ними, меняя фрагменты и исследуя результаты.
2. Что неправильно в данном фрагменте?

```
int main()
{
    throw 12.23;
```

3. Что неправильно в данном фрагменте?

```
try {
    // ...
    throw 'a';
    // ...
}
catch(char *) {
    // ...
}
```

4. Что может произойти при возбуждении исключительной ситуации, для которой не задано соответствующей инструкции **catch?**

11.4. Дополнительная информация об обработке исключительных ситуаций

В системе обработки исключительных ситуаций имеется несколько дополнительных аспектов и нюансов, которые могут сделать ее понятнее и удобней для применения.

В некоторых случаях необходимо настроить систему так, чтобы перехватывать все исключительные ситуации, независимо от их типа. Сделать это достаточно просто. Для этого используйте следующую форму инструкции **catch:**

```
catch(...) {
    // обработка всех исключительных ситуаций
}
```

Здесь многоточие соответствует любому типу данных.

Для функции, вызываемой из блока **try,** вы можете ограничить число типов исключительных ситуаций, которые способна возбудить такая функция. Фактически можно даже запретить функции вообще возбуждать какую бы то ни было исключительную ситуацию. Для этого необходимо добавить в определение функции ключевое слово **throw.** Здесь представлена основная форма такого определения:

```
возвращаемый_тип имя_функции(список_аргументов) throw (список_типов)
{
    // ...
}
```

Здесь в поле ***список_типов*** перечисляются через запятые только те типы данных исключительных ситуаций, которые могут быть возбуждены функцией. Возбуждение любого другого типа исключительной ситуации приведет к аварийному завершению программы. Если вы хотите, чтобы функция не возбуждала никаких исключительных ситуаций, то оставьте поле ***список_типов*** пустым.

Если ваш компилятор работает в соответствии с современным стандартом Standard C++, то попытка возбуждения неподдерживаемой исключительной ситуации приведет к вызову стандартной библиотечной функции **unexpected()**. По умолчанию функция **unexpected()** вызывает функцию **abort()**, что ведет к аварийному завершению программы. Однако при желании вы можете задать собственную процедуру завершения программы. За подробностями обращайтесь к справочной документации вашего компилятора.

Если вы хотите в процедуре обработки исключительной ситуации возбудить повторную исключительную ситуацию, можно просто использовать инструкцию **throw** без параметров. Это приводит к тому, что текущая исключительная ситуация передается внешней последовательности инструкций **try/catch.**

1. В следующей программе иллюстрируется использование инструкции **catch(...)**:

```
// В этой программе перехватываются все типы исключительных ситуаций
#include <iostream>
using namespace std;

void Xhandler(int test)
{
    try {
        // возбуждение исключительной ситуации типа int
        if(test==0) throw test;
```

```
        // возбуждение исключительной ситуации типа char
        if(test==1) throw 'a';
        // возбуждение исключительной ситуации типа double
        if(test==2) throw 123.23;
    }
    catch (...) { // перехват исключительных ситуаций всех типов
        cout << "Перехвачена ошибка!\n";
    }
}

intmain()
{
    cout << "начало\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка!
Перехвачена ошибка!
Перехвачена ошибка!
конец
```

Как видите, во всех трех случаях возбуждения исключительной ситуации в инструкции throw, она перехватывается с помощью единственной инструкции catch.

2. **Очень удобно инструкцию catch(...) использовать в качестве последней в группе инструкций catch. В этом случае инструкция catch(...) по умолчанию становится инструкцией, которая "перехватывает все". Например, далее представлена слегка измененная версия предыдущей программы, где исключительные ситуации целого типа перехватываются явно, а все другие — с помощью инструкции catch(...):**

```
/* В этом примере инструкция catch(...) по умолчанию перехватывает
все типы исключительных ситуаций , .
*/
#include <iostream>
using namespace std;

void Xhandler(int test)
{
```

```
try {
        // возбуждение исключительной ситуации типа int
        if(test==0) throw test;
        // возбуждение исключительной ситуации типа char
        if(test==1) throw 'a';
        // возбуждение исключительной ситуации типа double
        if(test==2) throw 123.23;
    }
    catch(int i) { // перехват исключительной ситуации типа int
        cout << "Перехвачен " << i << '\n';
    }
    catch(...) { // перехват исключительных ситуаций остальных типов
        cout << "Перехвачена ошибка!\n";
    }
}

int main()
{
    cout << "начало\n";

    Xhandler(0);
    Xhandler(1);
    Xhandler(2);

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачен 0
Перехвачена ошибка!
Перехвачена ошибка!
конец
```

Как показано в этом примере, использование инструкции **catch(...)** таким образом — это хороший способ перехватывать те исключительные ситуации, которые вы не хотите обрабатывать явно. Кроме этого, путем перехвата всех исключительных ситуаций вы предотвращаете аварийное завершение программы из-за случайно необработанной исключительной ситуации.

3. В следующей программе показано, как ограничить число типов исключительных ситуаций, которые возбуждаются функцией:

```
/* Ограничение числа возбуждаемых функцией типов исключительных
ситуаций
V
tinclude <iostream>
using namespace std;
```

```
// Этой функцией могут возбуждаться только
// исключительные ситуации типов int, char и double
void Xhandler(int test) throw(int, char, double)
{
        // возбуждение исключительной ситуации типа int
        if(test==0) throw test;
        // возбуждение исключительной ситуации типа char
        if(test==1) throw 'a';
        // возбуждение исключительной ситуации типа double
        if(test==2) throw 123.23;
}

int main()
{
    cout << "начало\n";

    try {
        Xhandler(0); // попробуйте также передать в
                     // функцию Xhandler() значения 1 и 2
    }
    catch(int i) {
        cout << "Перехват int\n";
    }
    catch(char c) {
        cout << "Перехват char\n";
    }
    catch(double d) {
        cout << "Перехват double\n";
    }

    cout << "конец";

    return 0;
}
```

В этой программе функция **XhandlerO** может возбуждать только исключительные ситуации типа **int, char и double.** При попытке возбудить исключительную ситуацию другого типа произойдет аварийное завершение программы. (То есть будет вызвана функция **unexpected().**) Чтобы убедиться в этом, удалите из списка допустимых исключительных ситуаций тип int и повторите запуск программы.

Важно понимать, что ограничить типы возбуждаемых исключительных ситуаций можно только после того, как функция вызвана из блока **try.** То есть *внутри* функции блок **try** может возбудить любой тип исключительной ситуации, коль скоро она перехватывается *внутри* этой функции. Ограничения вступают в силу только тогда, когда исключительная ситуация не перехвачена функцией.

4. Следующее небольшое изменение в функции **Xhandler()** запрещает возбуждение любой исключительной ситуации:

```
// Эта функция НЕ может вызывать никаких исключительных ситуаций
void Xhandler(int test) throw()
{
/*Следующие инструкции больше не работают. Наоборот, попытка их
выполнения ведет к ненормальному завершению программы
*/
        // возбуждение исключительной ситуации типа int
        if(test==0) throw test;
        // возбуждение исключительной ситуации типа char
        if(test==1) throw 'a';
        // возбуждение исключительной ситуации типа double
        if(test==2) throw 123.23;
}
```

5. Вы уже знаете, что можно повторно возбудить исключительную ситуацию. Смысл этого в том, чтобы предоставить возможность обработки исключительной ситуации нескольким процедурам. Например, предположим, что одна процедура обрабатывает один аспект исключительной ситуации, а вторая — другой. Повторно исключительная ситуация может быть возбуждена только внутри блока **catch** (или любой функцией, которая вызывается из этого блока). Когда вы повторно возбуждаете исключительную ситуацию, она перехватывается не той же инструкцией **catch,** а переходит к другой, внешней к данной инструкции. В следующей программе иллюстрируется повторное возбуждение исключительной ситуации: возбуждается исключительная ситуация типа **char *.**

```
/* Пример повторного возбуждения исключительной ситуации одного и
того же типа
*/
#include <iostream>
using namespace std;

void Xhandler()
{
    try {
        // возбуждение исключительной ситуации типа char *
        throw "привет";
    }
    // перехват исключительной ситуации типа char *
    catch(char *) {
        cout << "Перехват char * внутри функции Xhandler()\n";
        // повторное возбуждение исключительной ситуации
        // типа char *, но теперь уже не в функции Xhandler()
        throw;
    }
}
```

```
int main()
{
    cout << "начало\n";

    try {
        Xhandler();
    }
    catch(char *) {
        cout << " Перехват char * внутри функции main()\n";
    }

    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехват char * внутри функции Xhandler()
Перехват char * внутри функции main()
конец
```

1. Перед тем как двинуться дальше, откомпилируйте и запустите все примеры текущего раздела. Убедитесь, что вы понимаете, почему каждая программа выводит на экран ту или иную информацию.
2. Что неправильно в данном фрагменте?

```
try {
    // ...
    throw 10;
}
catch(int *p) {
    // ...
}
```

3. Предложите способ исправления предыдущего фрагмента.
4. Какая инструкция **catch** перехватывает все типы исключительных ситуаций?
5. Далее представлен каркас функции divide().

```
double divide(double a, double b)
{
    // добавьте обработку ошибок
    return a/b;
}
```

Эта функция возвращает результат деления a на **b**. Добавьте в функцию процедуру обработки исключительных ситуаций, а конкретно предусмотрите обработку ошибки деления на ноль. Покажите, что ваша процедура работает.

11.5. Обработка исключительных ситуаций, возбуждаемых оператором new

В главе 4 вы получили представление о том, что в соответствии с современной спецификацией оператора new, он возбуждает исключительную ситуацию при неудачной попытке выделения памяти. Поскольку в главе 4 об исключительных ситуациях мы еще не знали, описание того, как они обрабатываются было отложено. Теперь настало время подробно исследовать ситуацию неудачной попытки выделения памяти с помощью оператора **new.**

В материале этого раздела атрибуты оператора new описаны так, как это определено в современном едином международном стандарте Standard C++. Как уже упоминалось в главе 4, с момента появления языка C++ точное определение действий, которые должны выполняться при неудачной попытке выделения памяти с помощью оператора new, менялось несколько раз. Сразу после разработки языка при неудачной попытке выделения памяти оператор new возвращал нуль, несколькими годами позднее — возбуждал исключительную ситуацию. Кроме того, неоднократно менялось имя этой исключительной ситуации. В конце концов было решено, что неудачная попытка выделения памяти с помощью оператора **new** по умолчанию будет возбуждать исключительную ситуацию, но по желанию в качестве опции можно возвращать нулевой указатель. Таким образом, оператор **new** реализовывался по-разному в разное время разными производителями компиляторов. Хотя в будущем все компиляторы должны быть выполнены в точном соответствии с требованиями стандарта Standard C++, сегодня это не так. Если представленные здесь примеры программ у вас не работают, проверьте по документации вашего компилятора то, как именно в нем реализован оператор new.

В соответствии со стандартом Standard C++, когда требование на выделение памяти не может быть выполнено, оператор new возбуждает исключительную ситуацию **bad_alloc**. При невозможности перехватить эту исключительную ситуацию программа завершается. Хотя для коротких программ такой алгоритм кажется вполне очевидным и понятным, в реальных приложениях вы должны не только перехватить, но и каким-то разумным образом обработать эту исключительную ситуацию. Для доступа к указанной исключительной ситуации в программу необходимо включить заголовок **<new>.**

Изначально описанная выше исключительная ситуация называлась ***xalloc*** *и во время написания данной книги это имя продолжало использоваться на многих компиляторах. Тем не менее в дальнейшем оно, несомненно, будет вытеснено определенным в стандарте Standard C++ именем* ***bad_alloc***.

Как уже упоминалось, в соответствии с требованиями стандарта Standard C++ при неудачной попытке вьщеления памяти допускается, чтобы оператор **new** возвращал нуль, а не возбуждал исключительную ситуацию. Такая форма оператора **new** может оказаться полезной при компиляции устаревших программ на современном компиляторе, а также при замене функций **malloc()** операторами **new.** Ниже представлена именно эта форма оператора **new:**

```
указатель = new(nothrow) тип;
```

Здесь ***указатель*** — это указатель на переменную типа ***тип.*** Форма оператора new с ключевым словом **nothrow** (без вызова исключительной ситуации) функционирует аналогично его прежней, изначальной версии. Поскольку в этом случае при неудачной попытке выделения памяти возвращается нуль, оператор **new** может быть легко "встроен" в старые программы и вам не нужно заботиться о формировании процедуры обработки какой бы то ни было исключительной ситуации. Тем не менее, в новых программах механизм исключительных ситуаций предоставляет вам гораздо более широкие возможности.

1. В представленном ниже примере с оператором **new** использование блока **try/catch** дает возможность проконтролировать неудачную попытку выделения памяти.

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

int main()
{
    int *p;

    try {
        p = new int; // выделение памяти для целого
    } catch (bad_alloc xa) {
      cout << "Ошибка вьщеления памяти\n";
```

```
    return 1;
    }

  for(*p = 0; *p < 10; (*p)++)
      cout << *p << " ";

  delete p; // освобождение памяти

  return 0;
}
```

В данном примере, если при выделении памяти случается ошибка, она перехватывается инструкцией catch.

2. Поскольку в предыдущей программе при работе в нормальных условиях ошибка выделения памяти чрезвычайно маловероятна, в представленном ниже примере для демонстрации возможности возбуждения исключительной ситуации оператором **new** ошибка выделения памяти достигается принудительно. Процесс выделения памяти длится до тех пор, пока не произойдет ошибка.

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

int main()
{
  double *p;

  // цикл будет продолжаться вплоть до исчерпания ресурса памяти
  do {
    try {
      p = new double[100000];
    } catch (bad_alloc xa) {
      cout << "Ошибка выделения памяти\n";
      return 1;
      }
      cout << "Выделение памяти идет нормально\n";
  } while(p);

  return 0;
}
```

3. В следующей программе показано, как использовать альтернативную форму оператора new — оператор **new(nothrow)**. Это переработанная версия предыдущей программы с принудительным возбуждением исключительной ситуации.

```
// Демонстрация работы оператора new(nothrow)
#include <iostream>
#include <new>
```

```
using namespace std;

int main()
{
    double *p;

    // цикл будет продолжаться вплоть до исчерпания ресурса памяти

    do {
        p = new(nothrow) double[100000];
        if(p) cout << "Выделение памяти идет нормально\n";
        else cout << "Ошибка выделения памяти\n";
    } while(p);

    return 0;
}
```

Как показано в этой программе, при использовании оператора new с ключевым словом **nothrow**, после каждого запроса на выделение памяти следует проверять возвращаемое оператором значение указателя.

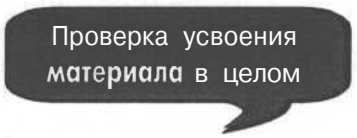
1. Объясните, в чем разница между функционированием операторов **new** и **new(nothrow)**, если при выделении памяти происходит ошибка.
2. Дан следующий фрагмент программы. Приведите два переделанных варианта этого фрагмента с учетом современных требований к программам на C++.

```
p = malloc(sizeof(int));

    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
```

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Создайте родовую функцию, возвращающую значение элемента, который чаще всего встречается в массиве.

2. Создайте родовую функцию, возвращающую сумму значений элементов массива.
3. Создайте родовой класс для "пузырьковой" сортировки (или используйте любой другой известный вам алгоритм сортировки).
4. Измените класс **stack** так, чтобы в стеке можно было хранить пары объектов разных типов.
5. Напишите обычные формы инструкций **try, catch и throw.** Опишите своими словами их функции.
6. Еще раз измените класс **stack** так, чтобы переполнение и, наоборот, опустошение стека обрабатывались как исключительные ситуации.
7. Просмотрите документацию на ваш компилятор. Проверьте, поддерживает ли он функции **terminate() и unexpected()**. Как правило, эти функции можно конфигурировать так, чтобы из них вы могли вызвать любую необходимую вам функцию. Если в случае с вашим компилятором это так, постарайтесь создать собственный набор функций завершения программы, который обеспечил бы возможность обработки необрабатываемых до этого исключительных ситуаций.
8. Философский вопрос: в чем, по вашему мнению, при неудачной попытке выделения памяти преимущество возбуждения исключительной ситуации оператором new по сравнению с возвращением нуля?



В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущей глав.

1. В главе 6, раздел 6.7, пример 3 был представлен класс с безопасным массивом. Переделайте его в родовой класс с безопасным массивом.
2. В главе 1 были созданы перегруженные версии функции **abs().** Усовершенствуйте решение, создав родовую функцию abs(), которая возвращала бы абсолютную величину любого численного объекта.

Глава 12

Динамическая идентификация и приведение типов

В этой главе рассказывается о двух сравнительно новых инструментах C++: динамической идентификации типа (Run-Time Type Identification, RTTI) и новых, более совершенных операторах приведения типов (casting operators). Динамическая идентификация типа дает возможность определить тип объекта во время выполнения программы. Новые операторы приведения типов предоставляют более безопасные и управляемые способы выполнения операций приведения типов, по сравнению с существовавшими ранее. Как вы увидите в дальнейшем, один из операторов приведения типов, а именно оператор **dynamic_cast**, относится непосредственно к RTTI, поэтому имело смысл объединить эти две темы в одной главе.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Что такое родовая функция и какова ее основная форма?
2. Что такое родовой класс и какова его основная форма?
3. Напишите родовую функцию **gexp(),** возвращающую значение одного из своих аргументов, возведенного в степень другого.
4. В главе 9, раздел 9.7, пример 1 был создан класс **coord** для хранения целочисленных координат. Создайте родовую версию этого класса, чтобы можно было хранить координаты любого типа. Продемонстрируйте программу решения этой задачи.
5. Кратко объясните, как совместная работа инструкций **try, catch и throw** обеспечивает в C++ обработку исключительных ситуаций.
6. Можно ли использовать инструкцию **throw,** если ход выполнения программы не затрагивает инструкции, расположенные в блоке **try?**
7. Для чего служат функции **terminate() и unexpected()**?
8. Какая форма инструкции **catch** будет обрабатывать все типы исключительных ситуаций?

12.1. Понятие о динамической идентификации типа

Поскольку динамическая идентификация типа не характерна для языков программирования, в которых не поддерживается полиморфизм (например, C), это понятие может оказаться для вас неизвестным. В языках, в которых не поддерживается полиморфизм, информация о типе объекта во время выполнения программы просто не нужна, так как тип каждого объекта известен уже на этапе компиляции программы (вернее даже тогда, когда программа еще пишется). С другой стороны, в языках, поддерживающих полиморфизм (таких, как C++), возможны ситуации, в которых тип объекта на этапе компиляции неизвестен, поскольку до выполнения программы не определена точная природа объекта. Как вы знаете, в C++ полиморфизм реализуется через иерархии классов, виртуальные функции и указатели базовых классов. При таком подходе указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на *объект любого класса, производного от этого базового.* Следовательно, не всегда есть возможность заранее узнать тип объекта, на который будет указывать указатель базового класса в каждый данный момент времени. В таких случаях определение типа объекта должно происходить во время выполнения программы, а для этого служит механизм динамической идентификации типа.

Информацию о типе объекта получают с помощью оператора typeid. Для использования оператора **typeid** в программу следует включить заголовок **<typeinfo>**. Ниже представлена основная форма оператора **typeid:**

```
typeid(объект)
```

Здесь ***объект*** — этот тот объект, информацию о типе которого необходимо получить. Оператор **typeid** возвращает ссылку на объект типа **type_info**, который и описывает тип объекта ***объект.*** В классе **type_info** определены следующие открытые члены:

```
bool operator==(const type_info &объект);
bool operator!=(const type_info &объект);
bool before(const type_info &объект);
const char *name();
```

Сравнение типов обеспечивают перегруженные операторы == и !=. Функция **before()** возвращает истину, если вызывающий объект в порядке сортировки расположен раньше объекта, заданного в качестве параметра. (Эта функция обычно предназначена только для внутреннего использования. Ее возвращаемое значение вряд ли может пригодиться при операциях с наследованием или иерархиями классов.) Функция **name()** возвращает указатель на имя типа.

Хотя оператор **typeid** позволяет получать типы разных объектов, наиболее полезен он будет, если в качестве его аргумента задать указатель полиморфного базового класса. В этом случае оператор автоматически возвращает тип реального объекта, на который указывает указатель. Этим объектом может быть как объект базового класса, так и объект любого класса, производного от этого базового. (Вспомните, указатель базового класса может указывать либо на объект базового класса, либо на объект любого класса, производного от этого базового.) Таким образом, с помощью оператора **typeid** во время выполнения программы можно определить тип объекта, на который указывает указатель базового класса. То же самое относится и к ссылкам. Когда в качестве аргумента оператора **typeid** указана ссылка на объект полиморфного класса, оператор возвращает тип реального объекта, на который имеется ссылка. Этим объектом, так же как и в случае с указателем, может быть объект производного класса. Когда оператор **typeid** применяют к неполиморфному классу, получают указатель или ссылку базового типа.

Ниже представлена вторая форма оператора **typeid,** в которой в качестве аргумента указывают имя типа:

```
typeid(имя_типа)
```

Обычно с помощью данной формы оператора **typeid** получают объект типа **type_info,** который можно использовать в- инструкции сравнения типов.

Поскольку оператор **typeid** чаще всего применяют к разыменованному указателю (т. е. указателю, к которому уже был применен оператор *), для обработки положения, когда разыменованный указатель равен нулю, была придумана специальная исключительная ситуация **bad_typeid,** которую в этом случае возбуждает оператор **typeid.**

Динамическая идентификация типа используется далеко не в каждой программе. Тем не менее, если вы работаете с полиморфными типами данных, она позволяет в самых разнообразных ситуациях определять типы обрабатываемых объектов.

1. В следующей программе демонстрируется использование оператора **typeid.** Сначала с помощью этого оператора мы получаем информацию об одном из встроенных типов данных C++ — типе **int.** Затем оператор **typeid** дает нам возможность вывести на экран типы объектов, на которые указывает указатель **p,** являющийся указателем базового класса **BaseClass.**

```
// Пример использования оператора typeid
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
class BaseClass {
    virtual void f(). {}; // делаем класс BaseClass полиморфным
    // ...
};

class Derived1: public BaseClass {
    // ...
};

class Derived2: public BaseClass {
    // ...
};

int main()
{
    int i;
    BaseClass *p, baseob;
    Derived1 ob1;
    Derived2 ob2;

    // Вывод на экран встроенного типа данных
    cout << "Тип переменной i — это ";
    cout << typeid(i) .name() << endl;

    // Обработка полиморфных типов
    p = &baseob;
    cout << "Указатель p указывает на объект типа " ;
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "Указатель p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "Указатель p указывает на объект типа ";
    cout << typeid(*p).name() << endl;

    return 0;
}
```

Программа выводит на экран следующую информацию:

```
Тип переменной i — это int
Указатель p указывает на объект типа BaseClass
Указатель p указывает на объект типа Derived1
Указатель p указывает на объект типа Derived2
```

Как уже отмечалось, когда в качестве аргумента оператора **typeid** задан указатель полиморфного базового класса, реальный тип объекта, на который указывает указатель определяется во время выполнения программы, что очевидно по выводимой на экран информации. В качестве эксперимента за-

комментируйте виртуальную функцию f() в определении базового класса BaseClass и посмотрите, что получится.

2. Ранее уже говорилось, что когда в качестве аргумента оператора typeid указана ссылка полиморфного базового класса, возвращаемым типом будет тип реального объекта, на который дана ссылка. Чаще всего это свойство используется в ситуациях, когда объекты передаются функциям по ссылке. Например, в следующей программе в объявлении функции **WhatType()** объект типа BaseClass задан параметром-ссылкой. Это означает, что функции **WhatType()** можно передавать ссылки на объекты типа BaseClass или типов, производных от класса BaseClass. Если в операторе typeid задать такой параметр, то он возвратит тип реального объекта.

```
// Использование оператора typeid со ссылкой в качестве аргумента
#include <iostream>
ttinclude <typeinfo>
using namespace std;

class BaseClass {
    virtual void f() {}; // делаем класс BaseClass полиморфным
    // ...
};

class Derivedl: public BaseClass {
    // ...
};

class Derived2: public BaseClass {
    // ...
};

// Задание ссылки в качестве параметра функции
void WhatType (BaseClass &ob)
{
    cout << "ob — это ссылка на объект типа " ;
    cout << typeid(ob) .name() << endl;
}

int main()
{
    int i;
    BaseClass baseob;
    Derivedl ob1;
    Derived2 ob2;

    WhatType (baseob)
    WhatType (ob1)
    WhatType (ob2)

    return 0;
}
```

Программа выводит на экран следующую информацию:

```
ob — это ссылка на объект типа BaseClass
ob — это ссылка на объект типа Derived1
ob — это ссылка на объект типа Derived2
```

3. **Хотя получение имени типа объекта в некоторых ситуациях оказывается весьма полезным, часто бывает необходимо узнать, соответствуют ли друг другу типы нескольких объектов. Это легко сделать, зная что объект типа type_info, возвращаемый оператором typeid, перегружает операторы == и !=. В представленной ниже программе показано использование этих операторов.**

```
// Использование операторов == и != с оператором typeid
#include <iostream>
#include <typeinfo>
using namespace std;

class X {
    virtual void f() {}
};

class Y {
    virtual void f() {}
};

int main()
{
    X x1, x2;
    Y y1;

    if(typeid(x1) == (typeid(x2))
        cout << "Тип объектов x1 и x2 одинаков\n";
    else
        cout << "Тип объектов x1 и x2 не одинаков\n";

    if(typeid(x1) != (typeid(y1))
        cout << "Тип объектов x1 и y1 не одинаков\n";
    else
        cout << "Тип объектов x1 и y1 одинаков\n";

    return 0;
}
```

Программа выводит на экран следующую информацию:

```
Тип объектов x1 и x2 одинаков
Тип объектов x1 и y1 не одинаков
```

4. **Хотя в предыдущих примерах и были показаны некоторые приемы работы с оператором type_info, главных его достоинств мы не увидели, поскольку типы объектов были известны уже на этапе компиляции программы. В** сле-

дующем примере этот пробел восполнен. В программе определена простая иерархия классов, предназначенных для рисования на экране разного рода геометрических фигур. На вершине иерархии находится абстрактный класс Shape. Его наследуют четыре класса: Line, Square, Rectangle и **NullShape**. Функция **generator()** генерирует объект и возвращает указатель на него. (Функцию, предназначенную для создания объектов, иногда называют *фабрикой объектов.)* То, какой именно объект создается, определяет генератор случайных чисел **rand()**. В функции **main()** реализован вывод получающихся объектов разных типов на экран, исключая объекты типа NullShape, у которых нет какой бы то ни было формы. Поскольку объекты возникают Случайно, заранее неизвестно, какой объект будет создан следующим. Следовательно, для определения типа создаваемых объектов требуется динамическая идентификация типа.

```
// Использование операторов == и != с оператором typeid
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

class Shape {
public:
    virtual void example() = 0;
};

class Rectangle: public Shape {
public:
    void example() {
        cout << "*****\n*   *\n*   *\n*****\n";
    }
};

class Triangle: public Shape {
public:
    void example() {
        cout << "*\n* *\n*  *\n*****\n";
    }
};

class Line: public Shape {
public:
    void example() {
        cout << "*****\n";
    }
};

class NullShape: public Shape {
public:
```

```
    void example() {
}
};

// Фабрика производных от класса Shape объектов
Shape *generator()
{
    switch(rand() % 4) {
        case 0:
            return new Line;
        case 1:
            return new Rectangle;
        case 2:
            return new Triangle;
        case 3:
            return new NullShape;
    }
    return NULL;
}

int main()
{
    int i;
    Shape *p;

    for(i=0; i<10; i++) {
        p = generator(); // создание следующего объекта

        cout << typeid(*p).name() << endl;

        // рисует объект, если он не типа NullShape
        if(typeid(*p) != typeid(NullShape))
            p->example();
    }

    return 0;
}
```

Программа выводит на экран следующее:

```
class Rectangle
*****
*   *
*   *
*****
class NullShape
class Triangle
*
* *
*  *
*****
```

```
class Line
*****
class Rectangle
*****
*   *
*   *
*****
class Line
*****
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Triangle
*
* *
*  *
*****
class Line
*****
```

5. Оператор **typeid** может работать с классами-шаблонами. Например, рассмотрим следующую программу. В ней для хранения некоторых значений создается иерархия классов-шаблонов. Виртуальная функция **get_val()** возвращает определенное в каждом классе значение. Для класса **Num** это значение соответствует самому числу. Для класса **Square** — это квадрат числа. Для класса **Sqr_root** — это квадратный корень числа. Объекты, производные от класса **Num,** генерирует функция **genegator()**. С помощью оператора **typeid** определяется тип генерируемых объектов.

```
// Использование оператора typeid с шаблонами
finclude  <iostream>
#include <cstdlib>
#include <cmath
#include <typeinfo>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i)  { x = i; }
```

```
    virtual T get_val() { return x; }
};

template <class T>
class Squary: public Num<T> {
public:
    Squary(T i) : Num<T>(i) {}
    T get_val() { return x*x; }
};

template <class T>
class Sqr_root: public Num<T> {
public:
    Sqr_root(T i) : Num<T>(i) {}
    T get_val() { return sqrt( (double) x) ; }
};

// Фабрика производных от класса Num объектов
Num<double> *generator()
{
    switch(rand() % 2) {
        case 0: return new Squary<double> (rand() % 100);
        case 1: return new Sqr_root<double> (rand() % 100);
    }
    return NULL;
}

int main()
{
    Num<double> ob1 (10), *p1;
    Squary<double> ob2 (100.0);
    Sqr_root<double> ob3 (999.2);
    int i;

    cout << typeid(ob1).name() << endl;
    cout << typeid(ob2).name() << endl;
    cout << typeid(ob3).name() << endl;

    if(typeid(ob2) == typeid( Squary<double>))
        cout << "is Squary<double>\n";

    p1 = &ob2;

    if(typeid(*p1) != typeid(ob1))
        cout << "Значение равно: " << p1->get_val() ;
    cout << "\n\n";

    cout << "Теперь генерируем объекты\n";
    for(i=0; i<10; i++) {
        p1 = generator(); // получение следующего объекта
```

```
        if(typeid(*p1) == typeid(Squary<double>))
            cout << "Квадрат объекта: ";
        if(typeid(*p1) == typeid(Sqr_root<double>))
            cout << "Квадратный корень объекта: ";

        cout << "Значение равно: " << p1->get_val();
        cout << endl;
    }

    return 0;
}
```

Программа выводит на экран следующее:

```
class Num<double>
class Squary<double>
class Sqr_root<double>
is Squary<double>

Значение равно: 10000
Теперь генерируем объекты
Квадратный корень объекта : Значение равно: 8.18535
Квадрат объекта: Значение равно: 0
Квадратный корень объекта : Значение равно: 4.89898
Квадрат объекта: Значение равно: 3364
Квадрат объекта: Значение равно: 4096
Квадратный корень объекта : Значение равно: 6.7082
Квадратный корень объекта : Значение равно: 5.19616
Квадратный корень объекта : Значение равно: 9.53939
Квадратный корень объекта : Значение равно: 6.48074
Квадратный корень объекта : Значение равно: 6
```

1. Зачем нужна динамическая идентификация типа?
2. Проведите эксперимент, о котором говорилось в примере 1. Что вы увидите на экране?
3. Правилен ли следующий фрагмент программы?

```
cout << typeid(float).name();
```

4. Дан фрагмент программы. Как определить, является ли p указателем на объект типа D2?

```
class B {
    virtual void f() {}
};
```

```
class D1: public. B {
    void f() {}
};

class D2: public B {
    void f() {}
};

int main()
{
    B *p;
```

5. По отношению к классу **Num** из примера 5 следующее выражение является истинным или ложным?

```
typeid(Num<int>) == typeid(Num<double>)
```

6. Поэкспериментируйте с RTTI. Хотя польза от динамической идентификации типа в контексте приведенных здесь простых примеров может показаться не слишком очевидной, тем не менее это мощный инструмент управления объектами во время работы программы.

12.2. Оператор *dynamic_cast*

Хотя в C++ продолжают поддерживаться характерные для языка C операторы приведения типов, имеется и несколько новых. Это операторы **dynamic_cast, const_cast, reinterpret_cast и static_cast**. Поскольку оператор **dynamic_cast** имеет непосредственное отношение к динамической идентификации типа, мы рассмотрим его первым. Остальные операторы приведения типов рассматриваются в следующем разделе.

Оператор **dynamic_cast** реализует приведение типов в динамическом режиме, что позволяет контролировать правильность этой операции во время работы программы. Если при выполнении оператора **dynamic_cast** приведения типов не произошло, будет выдана ошибка приведения типов. Ниже представлена основная форма оператора **dynamic_cast**:

```
dynamic_cast<целевой_тип> (выражение)
```

Здесь ***целевой_тип*** — это тип, которым должен стать тип параметра ***выражение*** после выполнения операции приведения типов. Целевой тип должен быть типом указателя или ссылки и результат выполнения параметра ***выражение*** тоже должен стать указателем или ссылкой. Таким образом, оператор **dynamic_cast** используется для приведения типа одного указателя к типу другого или типа одной ссылки к типу другой.

Основное назначение оператора **dynamic_cast** заключается в реализации операции приведения полиморфных типов. Например, пусть дано два поли-

морфных класса B и D, причем класс D является производным от класса B, тогда оператор **dynamic_cast** всегда может привести тип указателя D* к типу указателя B*. Это возможно потому, что указатель базового класса всегда может указывать на объект производного класса. Оператор dynamic_cast может также привести тип указателя B* к типу указателя **D***, но только в том случае, если объект, на который указывает указатель, действительно является объектом типа D. Как правило, оператор **dynamic_cast** выполняется успешно, когда указатель (или ссылка) после приведения типов становится указателем (или ссылкой) либо на объект целевого типа, либо на объект производного от целевого типа. В противном случае приведения типов не происходит. При неудачной попытке приведения типов результатом выполнения оператора dynamic_cast является нуль, если в операции использовались указатели. Если же в операции использовались ссылки, возбуждается исключительная ситуация bad_cast.

Рассмотрим простой пример. Предположим, что **Base** — это базовый класс, а **Derived** — это класс, производный от класса Base.

```
Base *bp, b_ob;
Derived *dp, d_ob;

bp = &d_ob; // указатель базового класса
            // указывает на объект производного класса
dp = dynamic_cast<Derived *> (bp)
if(!dp) cout "Приведение типов прошло успешно";
```

Здесь приведение типа указателя **bp** базового класса к типу указателя dp производного класса прошло успешно, поскольку указатель bp на самом деле указывает на объект производного класса **Derived.** Таким образом, после выполнения этого фрагмента программы на экране появится сообщение **Приведение типов прошло успешно.** Однако в следующем фрагменте операция приведения типов заканчивается неудачей, поскольку указатель bp указывает на объект базового класса **Base,** а приводить тип объекта базового класса к типу объекта производного класса неправильно.

```
bp = &b_ob; // указатель базового класса
            // указывает на объект базового класса
dp = dynamic_cast<Derived *> (bp)
if(!dp) cout "Приведения типов не произошло";
```

Поскольку попытка приведения типов закончилась неудачей, на экран будет выведено сообщение **Приведения типов не произошло.**

Оператор **dynamic_cast** в некоторых случаях можно использовать вместо оператора typeid. Например, опять предположим, что **Base** — это базовый класс, а **Derived** — это класс, производный от класса **Base.** В следующем фрагменте указателю dp присваивается адрес объекта, на который указывает указатель **bp,** но только в том случае, если это действительно объект класса **Derived.**

```
Base *bp;
Derived *dp;
// ...
if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

В данном примере для фактического выполнения операции приведения типов используется стиль языка С. Это безопасней, поскольку инструкция if с помощью оператора **typeid** проверяет правильность выполнения операции еще до того, как она действительно происходит. Тем не менее для этого есть более короткий путь. Оператор **typeid** с инструкцией **if** можно заменить оператором **dynamic_**cast:

```
dp = dynamic_cast<Derived *> (bp)
```

Поскольку оператор **dynamic_cast** заканчивается успешно только в том случае, если объект, к которому применяется операция приведения типов, является либо объектом целевого типа, либо объектом производного от целевого типа, то после выполнения приведенной выше инструкции указатель **dp** будет либо нулевым, либо указателем на объект типа **Derived.** Таким образом, оператор **dynamic_cast** успешно завершается только при правильном приведении типов, а это значит, что в определенных ситуациях он может упростить логику программы.

1. В следующей программе продемонстрировано использование оператора **dynamic_cast**:

```
// Использование оператора dynamic_cast
#include <iostream>
using namespace std;

class Base {
public:
    virtual void f() { cout << "Внутри класса Base"\n; }
};

class Derived: public Base {
public:
    void f() { cout << "Внутри класса Derived"\n; }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;
```

```
dp = dynamic_cast<Derived *> (&d_ob);
if(dp) {
    cout << "Тип Derived * к типу Derived * приведен успешно\n";
    dp->f();
} else
    cout << "Ошибка\n";

cout << endl;

bp = dynamic_cast<Base *> (&d_ob);
if(bp) {
    cout << "Тип Derived * к типу Base * приведен успешно\n";
    bp->f();
} else
    cout << "Ошибка\n";

cout << endl;

bp = dynamic_cast<Base *> (&b_ob);
if(bp) {
    cout << "Тип Base * к типу Base * приведен успешно\n";
    bp->f();
} else
    cout << "Ошибка\n";

cout << endl;

dp = dynamic_cast<Derived *> (&b_ob);
if(dp) {
    cout << "Ошибка\n";
} else
    cout << "Тип Base * к типу Derived * не приведен\n";

cout << endl;

bp = &d_ob; // bp указывает на объект типа Derived
dp = dynamic_cast<Derived *> (bp) ;
if(dp) {
   cout << "Указатель bp к типу Derived * приведен успешно\n" <<
           "поскольку bp в действительности указывает\n" <<
           "на объект типа Derived\n";
    dp->f();
} else
    cout << "Ошибка\n";

cout << endl;

bp = &b_ob; // bp указывает на объект типа Base
dp = dynamic_cast<Derived *> (bp);
if(dp)
   cout << "Ошибка\n";
```

```
    else {
    cout << "Указатель bp к типу Derived * не приведен\n" <<
            "поскольку bp в действительности указывает\n" <<
            "на объект типа Base\n";
    }

    cout << endl;

    dp = &d_ob; // dp указывает на объект типа Derived
    bp = dynamic_cast<Base *> (dp);
    if(bp) {
       cout << "Указатель dp к типу Base * приведен успешно\n" <<
       bp->f();
    } else
        cout << "Ошибка\n";

    return 0;
}
```

Программа выводит на экран следующую информацию:

```
Тип Derived * к типу Derived * приведен успешно
Внутри класса Derived

Тип Derived * к типу Base * приведен успешно
Внутри класса Derived

Тип Base * к типу Base * приведен успешно
Внутри класса Base

Тип Base * к типу Derived * не приведен

Указатель bp к типу Derived * приведен успешно
поскольку bp в действительности указывает
на объект типа Derived
Внутри класса Derived

Указатель bp к типу Derived * не приведен
поскольку bp в действительности указывает
на объект типа Base

Указатель dp к типу Base * приведен успешно
Внутри класса Derived
```

2. В следующем примере показано, как оператор **typeid** можно заменить оператором **dynamic_cast**.

```
/* Использование оператора dynamic_cast для замены оператора typeid
*/
ttinclude<iostream>
```

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

class Base {
public:
    virtual void f() {}
};
class Derived: public Base {
public:
    void derivedOnly() {
        cout << "Это объект класса Derived"\n;
    }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;

    // ************************************************************
    // использование оператора typeid
    // ************************************************************
    bp = &b_ob;
    if(typeid(*bp) == typeid(Derived)) {
        dp = (Derived *} bp;
        dp->derivedOnly();
    } else
        cout << "Тип Base к типу Derived не приведен\n";

    bp = &d_ob;
    if(typeid(*bp) == typeid(Derived)) {
        dp = (Derived *) bp;
        dp->derivedOnly();
    } else
        cout << "Ошибка, приведение типов должно работать!\n";


    // ************************************************************
    // использование оператора dynamic_cast
    // ************************************************************
    bp = &b_ob;
    dp = dynamic_cast<Derived *> (bp);
    if(dp)dp->derivedOnly();
    else
        cout << "Тип Base к типу Derived не приведен\n";

    bp = &d_ob;
    dp = dynamic_cast<Derived *> (bp);
```

```
    if(dp)dp->derivedOnly();
    else
        cout << "Ошибка, приведение типов должно работать!\n";
    return 0;
}
```

Как видите, использование оператора **dynamic_cast** делает проще логику приведения типа указателя базового класса к типу указателя производного класса. После выполнения программы на экран будет выведена следующая информация:

```
Тип Base к типу Derived не приведен
Это объект класса Derived
Тип Base к типу Derived не приведен
Это объект класса Derived
```

3. Оператор **dynamic_cast**, как и оператор **typeid**, можно использовать с классами-шаблонами. Например, в следующем примере представлен переработанный класс-шаблон из примера 5 раздела 12.1. Здесь тип объекта, возвращаемый функцией **genegator()**, определяется с помощью оператора **dynamic_cast**.

```
// Использование оператора dynamic_cast с шаблонами
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <typeinfo>
using namespace std;

template <class T> class Num {
public:
    T x;
    Num(T i) { x = i; }
    virtual T get_val() { return x; }
};

template <class T>
class Squary: public Num<T> {
public:
    Squary(T i) : Num<T>(i) {}
    T get_val() { return x*x; }
};

template <class T>
class Sqr_root: public Num<T> {
public:
    Sqr_root(T i) : Num<T>(i) {}
    T get_val() { return sqrt((double)x); }
};
```

```
// Фабрика производных от класса Num объектов
Num<double> *generator()
{
    switch(rand() % 2) {
        case 0: return new Squary<double> (rand() % 100);
        case 1: return new Sqr_root<double> (rand() % 100);
    }
    return NULL;
}

int main()
{
    Num<double> ob1(10), *p1;
    Squary<double> ob2(100.0), *p2;
    Sqr_root<double> ob3(999.2), *p3;
    int i;

    cout << "Генерируем несколько объектов\n";
    for(i=0; i<10; i++) {
        p1 = generator();

        p2 = dynamic_cast<Squary<double> *> (p1);
        if(p2) cout << "Квадрат объекта: ";

        p3 = dynamic_cast<Sqr_root<double> *> (p1);
        if(p3) cout << "Квадратный корень объекта: ";

        cout << "Значение равно: " << p1->get_val();
        cout << endl;
    }

    return 0;
}
```

1. Своими словами объясните назначение оператора **dynamic_cast**.
2. Дан следующий фрагмент программы. Покажите, как с помощью оператора **dynamic_cast** сделать так, чтобы указатель p указывал на некоторый объект ob только в том случае, если объект ob является объектом типа D2.

```
class B {
    virtual void f() {}
};

class D1: public B {
    void f() {}
};
```

```
class D2: public B {
    void f() {}
};

B *p;
```

3. Переделайте функцию **main()** из раздела 12.1, упражнение 4 так, чтобы для запрещения вывода на экран объектов типа NullShape использовался не оператор **typeid,** а оператор **dynamic_cast**.
4. Будет ли работоспособен следующий фрагмент программы в иерархии классов с базовым классом Num из примера 3 этого раздела?

```
Num<int> *Bp;
Square<double> *Dp;
// ...
Dp = dynamic_cast<Num<int>> (Bp);
```

12.3. Операторы *const_cast*, *reinterpret_cast* и *static_cast*

Хотя оператор **dynamic_cast** самый полезный из новых операторов приведения типов, кроме него программистам доступны еще три. Ниже представлены их основные формы:

const_cast<*целевой_тип*> *(выражение)*

reinterpret_cast<*целевой_тип*> (выражение)

static_cast<*целевой_тип*> (выражение)

Здесь ***целевой_тип*** — это тип, которым должен стать тип параметра ***выражение*** после выполнения операции приведения типов. Как правило, указанные операторы обеспечивают более безопасный и интуитивно понятный способ выполнения некоторых видов операций преобразования, чем оператор приведения типов, более характерный для языка C.

Оператор **const_cast** при выполнении операции приведения типов используется для явной подмены атрибутов **const** (постоянный) и/или **volatile** (переменный). Целевой тип должен совпадать с исходным типом, за исключением изменения его атрибутов **const** или **volatile.** Обычно с помощью оператора **const_cast** значение лишают атрибута **const.**

Оператор **static_cast** предназначен для выполнения операций приведения типов над объектами неполиморфных классов. Например, его можно использовать для приведения типа указателя базового класса к типу указателя производного класса. Кроме этого, он подойдет и для выполнения любой

стандартной операции преобразования, но только не в динамическом режиме (т. е. не во время выполнения программы).

Оператор **reinterpret_cast** дает возможность преобразовать указатель одного типа в указатель совершенно другого типа. Он также позволяет приводить указатель к типу целого и целое к типу указателя. Оператор **reinterpret_cast** следует использовать для выполнения операции приведения внутренне несовместимых типов указателей.

Атрибута **const** объект можно лишить только с помощью оператора **const_cast. С** помощью оператора **dynamic_cast**, **static_cast** или **reinterpret_cast** этого сделать нельзя.

1. В следующей программе демонстрируется использование оператора **reinterpret_cast**.

```
// Пример использования оператора reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
    int i;
    char *p = "Это строка";

    // приведение типа указателя к типу целого
    i = reinterpret_cast<int> (p);

    cout << i;

    return 0;
}
```

В данной программе с помощью оператора **reinterpret_cast** указатель на строку превращен в целое. Это фундаментальное преобразование типа и оно хорошо отражает возможности оператора **reinterpret_cast**.

2. В следующей программе демонстрируется оператор **const_cast.**

```
// Пример использования оператора const_cast
#include <iostream>
using namespace std;

void f(const int *p)
{
    int *v;
```

```
    // преобразование типа лишает указатель p атрибута const
    v = const_cast<int *> (p);
    *v = 100; // теперь указатель v может изменить объект
}

int main()
{
    int x = 99;
    cout << "Объект x перед вызовом функции равен: " << x << endl;
    f(&x);
    cout << "Объект x после вызова функции равен: " << x << endl;
    return 0;
}
```

Ниже представлен результат выполнения программы:

```
Объект x перед вызовом функции равен: 99
Объект x после вызова функции равен: 100
```

Как видите, несмотря на то что параметром функции f() задан постоянный указатель, вызов этой функции с объектом x в качестве параметра изменил значение объекта.

*Возможность снятия оператором **const_cast** атрибута **const** при выполнении операции приведения типов потенциально очень опасна. Пользоваться этой возможностью следует с величайшей осторожностью.*

3. Оператор **static_cast,** по существу, предназначен для замены прежнего оператора приведения типов. Он просто выполняет операцию приведения типов над объектами неполиморфных классов. Например, в следующей программе тип **float** приводится к типу **int.**

```
// Пример использования оператора static_cast
#include <iostream>
using namespace std;

int main()
{
    int i;
    float f;
    f = 199.22;
    i = static_cast<int> (f);
```

```
    cout << i;

    return 0;
}
```

1. Объясните, зачем нужны операторы **const_cast, reinterpret_cast и static_cast.**
2. В следующей программе имеется ошибка. Исправьте ее с помощью оператора **const_cast**.

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

void f(const double &i)
{
    i = 100; // Ошибка! Исправьте ее с помощью оператора const_cast
}

int main()
{
    double x = 98.6;

    cout << x << endl;
    f(x);
    cout << x << endl;

    return 0;
}
```

3. Объясните, почему оператор **const_cast** следует использовать только в самых крайних случаях.

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Опишите операции, которые можно выполнить с помощью оператора **typeid**.
2. Какой заголовок нужно включить в вашу программу для использования оператора **typeid?**

3. Помимо стандартного приведения типов, в C++ определено для этой цели еще четыре оператора. Что это за операторы и зачем они нужны?

4. Допишите следующую программу, чтобы на экран выводилась информация о выбранном пользователем типе объекта.

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

class A {
    virtual void f() {}
};

class B: public A {
};

class C: public B {
};

int main()
{
    A *p, a_ob;
    B b_ob;
    C c_ob;
    int i;

    cout << "Введите 0 для объектов типа A,";
    cout << "1 для объектов типа B или";
    cout << "2 для объектов типа C.\n";

    cin>>i;

    if(i==1) p = &b_ob;
    else if(i==2) p = &c_ob;
    else p = &a_ob;

    // выведите на экран сообщение о типе
    // выбранного пользователем объекта

    return 0;
}
```

5. Объясните, каким образом оператор **typeid** можно иногда заменить оператором **dynamic_cast**.

6. Тип какого объекта можно определить с помощью оператора typeid?

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Переделайте программу из раздела 12.1, пример 4, чтобы возможную ошибку выделения памяти внутри функции **generator()** отслеживать с помощью механизма обработки исключительных ситуаций.
2. Измените функцию **generator()** из вопроса 1, чтобы в ней использовать версию оператора new с ключевым словом **nothrow**.
3. Особо сложное задание: попытайтесь создать иерархию классов с абстрактным классом **DataStruct** на ее вершине. В основании иерархии создайте два производных класса. В одном должен быть реализован стек, в другом — очередь. Создайте также функцию **DataStructFactory()** со следующим прототипом:

```
DataStruct *DataStructFactory(char метка);
```

Функция **DataStructFactory()** должна создавать стек, если параметр ***метка*** равен s, и очередь, если параметр ***метка*** равен q. Возвращаемым значением функции должен быть указатель на созданный объект. Покажите, что ваша "фабрика объектов" работает.

Глава 13

Пространстваимен и другие темы

В этой главе рассказывается о пространствах имен (namespaces), функциях преобразования (conversion functions), статических (static) и постоянных (const) членах класса, а также о других необычных инструментах C++.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Что собой представляют новые операторы приведения типов и для чего они используются?
2. Что собой представляет класс **type_info**?
3. С помощью какого оператора определяют тип объекта?
4. Дан следующий фрагмент программы. Покажите, как определить, объектом какого типа является указатель p: объектом типа **Base** или объектом типа **Derived.**

```
class Base {
    virtual void f() {}
};

class Derived: public Base {
};

int raain()
{
    Base *p, b_ob;
    Derived d_ob;

    // ...
```

5. Оператор **dynamic_cast** успешно завершается только в том случае, если объект, над которым выполняется операция приведения типов, является

либо объектом целевого типа, либо объектом _____ от целевого типа. (Вставьте нужное слово.)

6. Можно ли с помощью оператора **dynamic_cast** снять атрибут const?

13.1. Пространства имен

О пространствах имен коротко было рассказано в главе 1, сейчас мы рассмотрим это понятие более подробно. Пространства имен появились в C++ относительно недавно. Они предназначены для локализации имен идентификаторов во избежание конфликтов имен (name collisions). В среде программирования C++ сосуществует невероятное количество имен переменных, функций и классов. До введения понятия пространств имен все эти имена находились в одном глобальном пространстве имен и возникало множество конфликтов. Например, если вы в своей программе определяли функцию **toupper()**, то она могла бы (в зависимости от списка своих параметров) подменить стандартную библиотечную функцию **toupper()**, поскольку имена обеих функций хранились бы в одном глобальном пространстве имен. Конфликты имен имеют место, когда в одной и той же программе используются библиотеки функций и классов разных производителей. В этом случае вполне возможно — и даже очень вероятно, — что имена, определенные в одной библиотеке, будут конфликтовать с теми же именами, но определенными в другой библиотеке.

Все проблемы решило введение понятия пространств имен и ключевого слова **namespace.** Это ключевое слово позволяет локализовать область видимости имен, объявленных в данном пространстве имен. Пространство имен дает возможность использовать одно и то же имя в разных контекстах и при этом почвы для конфликтов не возникает. Вероятно, больше всего выиграла от введения пространств имен стандартная библиотека C++ (C++ standard library). В ранних версиях языка вся библиотека C++ определялась в глобальном пространстве имен. Теперь библиотека C++ определяется в собственном пространстве имен **std,** что значительно снижает вероятность конфликтов имен. Вы также можете создавать в программе свои собственные пространства имен, чтобы локализовать область видимости тех имен, которые, по вашему мнению, могли бы вызывать конфликты. Это особенно важно при создании собственных библиотек классов или функций.

Ключевое слово **namespace** путем объявления именованных областей дает возможность разделить глобальное пространство имен. По существу, пространство имен определяет область видимости. Ниже приведена основная форма использования ключевого слова **namespace:**

```
namespace имя {
   // объявления
}
```

Все, что определено внутри инструкции **namespace,** находится внутри области видимости данного пространства имен.

Ниже приведен пример объявления пространства имен **MyNameSpace:**

```
namespace MyNameSpace {
    int i, k;
    void myfunc(int j) { cout << j; }

    class myclass {
    public:
        void seti(int x) { i = x; }
        int geti() { return i; }
    };
}
```

Здесь имена переменных **i** и **k**, функции **myfunc()**, а также класса **myclass** находятся в области видимости, определенной пространством имен **MyNameSpace.**

К идентификаторам, объявленным в пространстве имен, внутри этого пространства можно обращаться напрямую. Например, в пространстве имен **MyNameSpace** в инструкции **return i** переменная i указана явно. Однако поскольку ключевое слово **namespace** определяет некоторую область видимости, то при обращении извне пространства имен к объектам, объявленным внутри этого пространства, следует указывать оператор расширения области видимости. Например, чтобы присвоить значение 10 переменной i в той части программы, которая не входит в пространство имен **MyNameSpace,** необходимо использовать следующую инструкцию:

```
MyNameSpace::i = 10;
```

А чтобы объявить объект типа **myclass** в той части программы, которая не входит в пространство имен **MyNameSpace,** нужна такая инструкция:

```
MyNameSpace: :myclass ob;
```

Таким образом, для доступа к члену пространства имен извне этого пространства перед именем члена следует указать имя пространства имен с оператором расширения области видимости.

Можно себе вообразить, что если в вашей программе обращения к членам пространства имен происходят достаточно часто, необходимость каждый раз указывать имя пространства имен и оператор расширения области видимости может быстро надоесть. Для решения этой проблемы была разработана инструкция **using.** У этой инструкции имеются две основные формы:

```
using namespace имя;
using имя: : член;
```

В первой форме параметр ***имя*** задает имя пространства имен, доступ к которому вы хотите получить. При использовании этой формы инструкции **using** все члены, определенные в указанном пространстве имен, становятся доступными в текущем пространстве имен и с ними можно работать напрямую, без необходимости каждый раз указывать имя пространства и оператор расширения области видимости. При использовании второй формы инструкции **using** видимым делается только указанный в инструкции член пространства имен. Например, пусть у нас имеется описанное выше пространство имен MyNameSpace, тогда правильны все представленные ниже инструкции **using** и операторы присваивания:

```
using MyNameSpace: :k; // видимой делается только переменная k
k = 10; // инструкция правильна, поскольку переменная k видима

using namespace MyNameSpace; // видимыми делаются все члены
                             // пространства имен MyNameSpace
i = 10; // инструкция правильна, поскольку видимы все члены
        // пространства имен MyNameSpace
```

Имеется возможность объявить более одного пространства имен с одним и тем же именем. Это позволяет разделить пространство имен на несколько файлов или даже разделить пространство имен внутри одного файла. Рассмотрим следующий пример:

```
namespace NS {
    int i;
}

// ...

namespace NS {
    int j;
}
```

Здесь пространство имен NS разделено на две части. Несмотря на это, содержимое каждой части по-прежнему остается в одном и том же пространстве имен — пространстве NS.

Пространство имен должно объявляться вне всех остальных областей видимости за исключением того, что одно пространство имен может быть вложено в другое. То есть вложенным пространство имен может быть только в другое пространство имен, но не в какую бы то ни было иную область видимости. Это означает, что нельзя объявлять пространства имен, например, внутри функции.

Имеется пространство имен особого типа — *безымянное пространство имен (unnamed namespace)*. Безымянное пространство имен позволяет создавать идентификаторы, являющиеся уникальными внутри некоторого файла. Ниже представлена основная форма безымянного пространства имен:

```
namespace {
    // объявления
}
```

Безымянные пространства имен дают возможность задавать уникальные идентификаторы, известные только внутри области видимости одного файла. Таким образом, внутри файла, содержащего безымянное пространство имен, к членам этого пространства можно обращаться напрямую, без необходимости каких бы то ни было уточнений. Но вне этого файла такие идентификаторы неизвестны.

Вам вряд ли понадобится создавать свои пространства имен для небольших и средних по объему программ. Однако, если вы собираетесь создавать библиотеки функций или классов, предназначенных для многократного использования, или хотите гарантировать своим программам широкую переносимость, вам следует рассмотреть возможность размещения своих кодов внутри некоторого пространства имен.

1. В представленном ниже примере иллюстрируются атрибуты демонстрационных пространств имен.

```
// Демонстрационные пространства имен
#include <iostream>
using namespace std;

// определение первого пространства имен
namespace firstNS {
    class demo {
        int i;
    public:
        demo(int x) { i = x; }
        void seti(int x) { i = x; }
        int geti() { return i; }
    }

    char str[] = "Иллюстрация пространств имен\n";
    int counter;
}

// определение второго пространства имен
namespace secondNS {
    int x, y;
}

int main()
```

```
{
    // расширяем область видимости
    firstNS::demo ob(10);
    /* После объявления объекта ob, его функции-члены могут
использоваться прямо, без какого бы то ни было уточнения
пространства имен
*/
    cout << "Значение объекта ob равно: " << ob.geti();
    cout << endl;
    ob.seti(99);
    cout << "Теперь значение объекта ob равно: " << ob.geti();
    cout << endl;

    // вводим строку str в текущую область видимости
    using firstNS::str;
    cout << str;

    // вводим все пространство имен firstNS
    // в текущую область видимости
    using namespace firstNS;
    for(counter=10; counter; counter--)
        cout << counter << " ";
    cout << endl;

    // используем пространство имен secondNS
    secondNS::x = 10;
    secondNS::y = 20;

    cout << "Переменные x, y равны: " << secondNS::x;
    cout << ", " << secondNS::y << endl;

    // вводим все пространство имен secondNS
    // в текущую область видимости
    using namespace secondNS;
    demo xob(x), yob(y);

    cout << "Значения объектов xob, yob равны: " << xob.geti();
    cout << ", " << yob.geti() << endl;

    return 0;
}
```

После выполнения программы на экран будет выведено следующее:

```
Значение объекта ob равно: 10
Теперь значение объекта ob равно: 99
Иллюстрация пространств имен
10 9 8 7 6 5 4 3 2 1
Переменные x, y равны: 10, 20
Значения объектов xob, yob равны: 10, 20
```

Программа иллюстрирует одно важное положение: при совместном использовании нескольких пространств имен одно пространство не подменяет другое. Когда вы вводите некоторое пространство имен в текущую область видимости, его имена просто добавляются в эту область, независимо от того, находятся ли в ней в это время имена из других пространств имен. Таким образом, ко времени завершения программы в глобальное пространство имен были добавлены пространства имен **std, firstNS и secondNs.**

2. Как уже упоминалось, пространство имен можно разделить либо между файлами, либо внутри одного файла, тогда содержимое этого пространства имен объединяется. Рассмотрим пример объединения разделенного пространства имен.

```
// Объединение пространства имен
#include <iostream>
using namespace std;

namespace Demo {
    int a; // объявление переменной a в пространстве имен Demo
}

    int x; // объявление переменной x в глобальном пространстве имен

namespace Demo {
    int b; // объявление переменной b в пространстве имен Demo
}

int main()
{
    using namespace Demo;

    a = b = x = 100;

    cout << a << " " << b << " " << x;

    return 0;
}
```

В данном примере обе переменные, a и b, оказались в одном пространстве имен — пространстве имен **Demo,** а переменная x осталась в глобальном пространстве имен.

3. Как уже упоминалось, стандарт Standard C++ определяет целую библиотеку в собственном пространстве имен **std.** Именно по этой причине во всех программах этой книги имеется следующая инструкция:

```
using namespace std;
```

Эта инструкция делает пространство имен **std** текущим, что позволяет получить прямой доступ к именам функций и классов, определенных в библиотеке языка Standard C++, без необходимости каждый раз с помощью оператора

расширения области видимости уточнять, что используется пространство имен std.

Тем не менее, если пожелаете, можете перед каждым идентификатором ставить имя пространства имен std и оператор расширения области видимости — ошибки не будет. Например, в следующей программе библиотека языка Standard C++ не введена в глобальную область видимости.

```
// Явное задание используемого пространства имен
#include <iostream>

int main()
{
    double val;

    std::cout << "Введите число: ";

    std::cin >> val;

    std::cout << "Вот ваше число: ";
    std::cout << val;

    return 0;
}
```

Как показано в данной программе, чтобы воспользоваться стандартными потоками ввода и вывода **cin** и cout, перед именами этих потоков необходимо явно указывать их пространство имен.

Если в вашей программе не предусмотрено широкое использование библиотеки языка Standard C++, вы можете не вводить пространство имен std в глобальную область видимости. Однако, если в вашей программе содержатся тысячи ссылок на стандартные библиотечные имена, включить в программу идентификатор std гораздо проще, чем добавлять его чуть ли не к каждой инструкции.

4. Если в своей программе вы используете только несколько имен из стандартной библиотеки, может оказаться удобней с помощью инструкции using отдельно задать **эти** несколько имен. Преимущество такого подхода в том, что указанные имена можно вносить в программу без необходимости их уточнения, и в то же время не нужно вводить всю стандартную библиотеку в глобальное пространство имен. Рассмотрим пример:

```
// Введение в глобальное пространство только нескольких имен
linclude <iostream>

// обеспечение доступа к потокам cin и cout
using std::cout;
using std::cin;

int main()
{
    double val;
```

```
    cout << "Введите число: ";
    cin >> val;
    cout << "Вот ваше число: ";
    cout << val;
    return 0;
}
```

Здесь стандартными потоками ввода и вывода **cin** и **cout** можно пользоваться напрямую, но в то же время остальные имена из пространства имен std оставлены вне текущей области видимости.

5. Ранее уже говорилось, что библиотека исходного языка C++ была определена в глобальном пространстве имен. Если вам придется модернизировать старую программу на C++, то вам понадобится либо включить в нее инструкцию **using namespace std,** либо перед каждой ссылкой на члена библиотеки дописывать имя пространства имен с оператором расширения области видимости **std::**. Это особенно важно, если вы замените прежние заголовочные файлы заголовками нового стиля (без расширения .h). Помните, прежние заголовочные файлы размещают свое содержимое в глобальном пространстве имен, а заголовки нового стиля — в пространстве имен **std.**

6. В языке C, если вы хотите ограничить область видимости глобального имени только тем файлом, в котором это имя объявлено, вы должны объявить его как статическое, т. е. с идентификатором **static.** Например, предположим, что следующие два файла являются частью одной программы:

Первый файл

```
static int counter;
void f1() {
counter = 99; // OK
}
```

Второй файл

```
extern int counter;
void f2() {
counter = 10; // Ошибка
}
```

Поскольку переменная **counter** определена в первом файле, то и использовать ее можно в первом файле. Во втором файле, несмотря на указание в инструкции с переменной **counter** идентификатора **extern,** попытка использования этой переменной ведет к ошибке. Объявляя в первом файле переменную **counter** статической, мы ограничиваем ее область видимости этим файлом.

Хотя объявления глобальных переменных с идентификатором **static** в C++ по-прежнему доступны, для достижения той же цели здесь имеется лучший путь — использовать, как показано в следующем примере, безымянное пространство имен.

Первый файл

```
namespace {
    int counter;
}
void f1() {
counter = 99; // OK
}
```

Второй файл

```
extern int counter;
void f2() {
counter = 10; // Ошибка
}
```

Здесь область видимости переменной **counter** также ограничена первым файлом. Использовать безымянное пространство имен вместо ключевого слова **static** рекомендует новый стандарт Standard C++.

1. Переделайте представленную ниже программу из главы 9 так, чтобы в ней отсутствовала инструкция **using namespace std.**

```
// Превращение пробелов в вертикальные линии |
ttinclude <iostream>
ttinclude <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Преобразование <файл_ввода> <файл_вывода>\n";
        return 1;
    }

    ifstream fin(argv[1]); // открытие файла для ввода
    ofstream fout(argv[2]); // создание файла для вывода

    if(!fout) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    if(!fin) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char ch;

    fin.unsetf(ios::skipws); // не пропускать пробелы
    while(!fin.eof()) {
        fin>>ch;
        if(ch==' ')ch= ' | ';
        if(!fin.eof()) fout << ch;
    }

    fin.close();
    fout.close();

    return 0;
}
```

2. Объясните назначение безымянного пространства имен.
3. Опишите различия между двумя формами инструкции **using.**
4. Объясните, почему в подавляющее большинство программ данной книги введена инструкция **using.** Приведите альтернативу ее использованию.
5. Объясните почему, если вы создаете многократно используемый код, его желательно размещать в собственном пространстве имен.

13.2. Функции преобразования

Иногда бывает полезно преобразовать объект одного типа в объект другого типа. Хотя для этого можно воспользоваться перегруженной оператор-функцией, часто более легким (и лучшим) способом такого преобразования является функция преобразования. *Функция преобразования (conversion function)* преобразует объект в значение, совместимое с другим типом данных, который часто является одним из встроенных типов данных C++. Уточним, функция преобразования автоматически преобразует объект в значение, совместимое с типом выражения, в котором этот объект используется.

Здесь показана основная форма функции преобразования:

```
operator тип() { return значение; }
```

Здесь *тип* — это целевой тип преобразования, а ***значение*** — это значение объекта после выполнения преобразования. Функции преобразования возвращают значение типа *тип*. *У* функции преобразования не должно быть параметров, и она должна быть членом класса, для которого выполняется преобразование.

Как будет показано в примерах, обычно функция преобразования обеспечивает более ясный подход к преобразованию значения объекта в другой тип, чем любой другой метод C++, поскольку она позволяет явно включить объект в выражение, содержащее переменные целевого типа.

1. В следующей программе класс **coord** содержит функцию преобразования, которая преобразует объект в целое. В данном случае функция возвращает произведение двух координат; однако в каждом конкретном случае допускается любое необходимое действие.

```
// Простой пример функции преобразования
#include <iostream>
using namespace std;
```

```
class coord {
    int x, y;
public:
    coord(int i, int j) { x = i; y = j; }
    operator int() { return x*y; } // функция преобразования
};

int main()
{
    coord o1(2, 3), o2(4, 3) ;
    int i;

    i = o1; // объект o1 автоматически преобразуется в целое
    cout << i << '\n';

    i = 100 + o2; // объект o2 преобразуется в целое
    cout << i << '\n';

    return 0;
}
```

В результате выполнения этой программы на экран будут выведены значения 6 и 112.

Обратите внимание, что функция преобразования вызывается тогда, когда объект **o1** присваивается целому, а также когда объект o2 используется, как часть большого выражения, оперирующего с целыми. Как уже установлено, с помощью функции преобразования вы разрешаете вставлять объекты созданных вами классов в "обычные" выражения, без образования сложных цепочек перегруженных оператор-функций.

2. Следующим представлен другой пример функции преобразования. Здесь преобразуется строка типа **strtype** в символьный указатель на **str**.

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

class strtype {
    char str[80];
    int len;
public:
    strtype(char *s) { strcpy(str, s); len = strlen(s); }
    operator char *() { return str; } // преобразование в тип char *
};

int main()
{
    strtype s("Это проверка\n");
    char *p, s2[80];
```

```
    p = s; /У преобразование в тип char *
    cout << "Это строка: " << p << '\n';

    // при вызове функции преобразуется в тип char *
    strcpy(s2, s);
    cout << "Это копия строки: " << s2 << '\n';

    return 0;
}
```

Эта программа выводит на экран следующее:

```
Это строка: Это проверка
Это копия строки: Это проверка
```

Как можно заметить, функция преобразования вызывается не только при присваивании объекта s объекту p (который имеет тип **char** *), но она также используется как параметр для функции **strcpy()**. Вспомните, функция **strcpy()** имеет следующий прототип:

```
char *strcpy(char *s1, const char *s2);
```

Поскольку прототип определяет, что объект s2 имеет тип **char** *, функция преобразования объекта в тип **char** * вызывается автоматически. Этот пример показывает, как функция преобразования может помочь интегрировать ваши классы в библиотеку стандартных функций C++.

Упражнения

1. Используя класс **strtype** из примера 2, создайте функцию преобразования для превращения строки в целое. Функция преобразования должна возвращать длину строки, хранящейся в str. Покажите, что ваша функция преобразования работает.
2. Пусть дан класс:

```
class pwr {
    int base;
    int exp;
public:
    pwr(int b, int e) { base = b; exp = e; }
    // здесь создайте функцию преобразования в целое
};
```

3. Создайте функцию преобразования для превращения объекта типа **pwr** в целое. Функция должна возвращать результат возведения в степень $\mathbf{base}^{\mathbf{exp}}$.

13.3. Статические члены класса

Переменные — члены класса можно объявлять как статические (**static**). Используя статические переменные-члены, можно решить несколько непростых проблем. Если вы объявляете переменную статической, то может существовать только одна копия этой переменной — независимо от того, сколько объектов данного класса создается. Каждый объект просто использует (совместно с другими) эту одну переменную. Запомните, для обычных переменных-членов при создании каждого объекта создается их новая копия, и доступ к каждой копии возможен только через этот объект. (Таким образом, для обычных переменных каждый объект обладает собственными копиями переменных.) С другой стороны, имеется только одна копия статической переменной — члена класса, и все объекты класса используют ее совместно. Кроме этого, одна и та же статическая переменная будет использоваться всеми классами, производными от класса, в котором эта статическая переменная содержится.

Может показаться необычным то, что статическая переменная — член класса создается еще *до* того, как создан объект этого класса. По сути, статический член класса — это просто глобальная переменная, область видимости которой ограничена классом, в котором она объявлена. В результате, как вы узнаете из следующих примеров, доступ к статической переменной-члену возможен без всякой связи с каким бы то ни было объектом.

Когда вы объявляете статические данные-члены внутри класса, вы не определяете их. Определить их вы должны где-нибудь вне класса. Для того чтобы указать, к какому классу статическая переменная принадлежит, вам необходимо переобъявить (redeclare) ее, используя операцию расширения области видимости.

Все статические переменные-члены по умолчанию инициализируются нулем. Однако, если это необходимо, статической переменной класса можно дать любое выбранное начальное значение.

Запомните, что основной смысл поддержки в C++ статических переменных-членов состоит в том, что благодаря им отпадает необходимость в использовании глобальных переменных. Как можно предположить, если при работе с классами задавать глобальные переменные, то это почти всегда нарушает принцип инкапсуляции, являющийся фундаментальным принципом ООР и C++.

Помимо переменных-членов статическими можно объявлять и функции-члены, но обычно это не делается. Доступ к объявленной статической функции — члену класса возможен только для других статических членов этого класса. (Естественно, что доступ к объявленной статической функции — члену класса возможен также и для глобальных данных и функций.) У статической функции-члена нет указателя **this.** Статические функции-члены не могут быть виртуальными. Статические функции-члены не могут

объявляться с идентификаторами **const** (постоянный) и **volatile** (переменный). И наконец, статические функции — члены класса могут вызываться любым объектом этого класса, а также через имя класса и оператор расширения области видимости без всякой связи с каким бы то ни было объектом этого класса.

1. Простой пример использования статической переменной-члена.

```
// Пример статической переменной-члена
#include <iostream>
using namespace std;

class myclass {
    static int i;
public:
    void seti(int n) { i = n; }
    int geti() { return i; }
};

// Определение myclass::i. Переменная i по-прежнему
// остается закрытым членом класса myclass
int myclass::i;

int main()
{
    myclass o1, o2;

    o1.seti(10);

    cout << "o1.i: " << o1.geti() << '\n'; // выводит значение 10
    cout << "o2.i: " << o2.geti() << '\n'; // также выводит 10

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
o1.i: 10
o2.i: 10
```

Глядя на программу, можно заметить, что фактически только для объекта **ol** устанавливается значение статической переменной-члена i. Однако поскольку переменная i совместно используется объектами ol и o2 (и, на самом деле, всеми объектами типа **myclass),** оба вызова функции **geti()** дают один и тот же результат.

Обратите внимание, что переменная i объявляется внутри класса **myclass,** но определяется вне его. Этот второй шаг гарантирует, что для переменной i будет выделена память. С технической точки зрения, определение класса является всего лишь определением типа и не более. Память для статической переменной при этом не выделяется. Поэтому для выделения памяти статической переменной-члену требуется ее явное определение.

2. Поскольку статическая переменная — член класса существует еще до создания объекта этого класса, доступ к ней в программе может быть реализован без всякого объекта. Например, в следующем варианте предыдущей программы для статической переменной i устанавливается значение 100 без всякой ссылки на конкретный объект. Обратите внимание на использование оператора расширения области видимости для доступа к переменной i.

```
// Для обращения к статической переменной объект не нужен
#include <iostream>
using namespace std;

class myclass {
public:
    static int i;
    void seti(int n) { i = n; }
    int geti() { return i; }
};

int myclass::i;

int main()
{
    myclass o1, o2;

    // непосредственное задание значения переменной i
    myclass::i = 100; // объекты не упоминаются

    cout << "o1.i: " << o1.geti() << '\n'; // выводит 100
    cout << "o2.i: " << o2.geti() << '\n'; // также выводит 100

    return 0;
}
```

Так как статической переменной i присвоено значение 100, программа выводит на экран следующее:

```
o1.i: 100
o2.i: 100
```

3. Традиционным использованием статических переменных класса является координация доступа к разделяемым ресурсам, таким как дисковая память, принтер или сетевой сервер. Как вы уже, вероятно, знаете из своего опыта, координация доступа к разделяемым ресурсам требует некоторых знаний о

последовательности событий. Чтобы понять, как с помощью статических переменных-членов можно управлять доступом к разделяемым ресурсам, изучите следующую программу. В ней создается класс **output,** который поддерживает общий выходной буфер **outbuf,** являющийся статическим символьным массивом. Этот буфер используется для получения выходной информации, передаваемой функцией-членом **outbuf ()**. Эта функция посимвольно передает содержимое строки **str.** Сначала функция запрашивает доступ к буферу, а затем передает в него все символы **str.** Функция предотвращает доступ в буфер другим объектам до окончания вывода. Понять работу программы можно по комментариям к ней.

```
// Пример разделения ресурсов
#include <iostream>
#include <cstring>
using namespace std;

class output {
    static char outbuf[255]; // это разделяемые ресурсы
    static int inuse; // если переменная inuse равна 0,
                      // буфер доступен; иначе — нет
    static int oindex; // индекс буфера
    char str[80];
    int i; // индекс следующего символа строки
    int who; // идентификатор объекта должен быть положительным
public:
    output(int w, char *s) {
        strcpy(str, s); i = 0; who = w;
    }

/* Эта функция возвращает -1 при ожидании буфера; 0 — при завершении
вывода; who — если буфер все еще используется.
*/
    int putbuf()
    {
        if(!str[i]) { // вывод завершен
            inuse = 0; // освобождение буфера
            return 0; // признак завершения
        }
        if(!inuse) inuse = who; // захват буфера
        if(inuse != who) return -1; // буфер кто-то использует
        if(str[i]) { // символы еще остались
            outbuf[oindex] = str[i];
            i++; oindex++;
            outbuf[oindex] = '\0'; // последним всегда идет нуль
            return 1;
        }
        return 0;
    }
```

```
    void show() { cout << outbuf << '\n';}
};

char output::outbuf[255]; // это разделяемые ресурсы
int output::inuse = 0; // если переменная inuse равна 0,
                       // буфер доступен; иначе – нет
int output::oindex = 0; // индекс буфера

int main()
{
    output o1(1, "Это проверка "), o2(2, "статических переменных");

    while (o1.putbuf() I o2.putbuf()); // вывод символов

    o1.show();

    return 0;
}
```

4. Статические функции-члены применяются достаточно редко, но для предварительной (до создания реального объекта) инициализации закрытых статических данных-членов они могут оказаться очень удобными. Например, ниже представлена совершенно правильная программа.

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

class static_func_demo {
    static int i;
public:
    static void init(int x) { i = x; }
    void show() { cout << i; }
};

int static_func_demo: : i; // определение переменной i

int main()
{
    // инициализация статических данных еще до создания объекта
    static_func_demo::init(100);
    static_func_demo x;
    x.show(); // вывод на экран значения 100

    return 0;
}
```

Здесь вызов функции **init()** инициализирует переменную i еще до создания объекта типа **static_func_demo.**

Упражнения

1. Переделайте пример 3 так, чтобы на экране отображался тот объект, который осуществляет вывод символов, и тот объект или те объекты, для которых из-за занятости буфера вывод запрещен.
2. Одним из интересных применений статических переменных-членов является хранение информации о количестве объектов класса, существующих в каждый конкретный момент времени. Для этого необходимо увеличивать на единицу статическую переменную-член каждый раз, когда вызывается конструктор класса, и уменьшать на единицу, когда вызывается деструктор. Реализуйте эту схему и продемонстрируйте ее работу.

13.4. Постоянные и модифицируемые члены класса

Функции — члены класса могут объявляться постоянными (с идентификатором **const).** Если функция объявлена постоянной, она не может изменить вызывающий ее объект. Кроме этого, постоянный объект не может вызвать непостоянную функцию-член. Тем не менее, постоянная функция-член может вызываться как постоянными, так и непостоянными объектами.

Для задания постоянной функции-члена используйте ее форму, представленную в следующем примере:

```
class X {
    int some_var;
public:
    int f1() const; // постоянная функция-член
};
```

Обратите внимание, что ключевое слово **const** указывают следом за списком параметров функции, а не перед именем функции.

Возможна ситуация, когда вам понадобится, чтобы функция-член, оставаясь постоянной, тем не менее была способна изменить один или несколько членов класса. Это достигается заданием модифицируемых членов класса (ключевое слово **mutable).** Модифицируемый член класса можно изменить с помощью постоянной функции-члена.

Примеры

1. Функция-член объявляется постоянной, чтобы предотвратить возможность изменения вызвавшего ее объекта. Для примера рассмотрим следующую программу.

```
/* Пример объявления постоянных функций-членов. Данная программа
содержит ошибку и компилироваться не будет
*/
#include <iostream>
using namespace std;

class Demo {
    int i;
public:
    int geti() const {
        return i; // здесь все правильно
    }

    void seti(int x) const {
        i = x; // Ошибка!!!
    }
};

int main()
{
    Demo ob;

    ob.seti(1900);
    cout << ob.geti();

    return 0;
}
```

Данная программа не будет компилироваться, поскольку функция-член seti() объявлена постоянной, что означает невозможность изменения вызывающего ее объекта. Таким образом, попытка изменения функцией переменной i ведет к ошибке. С другой стороны, поскольку функция **geti()** не меняет переменной i, она совершенно правильна.

2. Чтобы допустить изменение избранных членов класса постоянной функцией-членом, они задаются модифицируемыми. Ниже представлен пример.

```
// Пример задания модифицируемого члена класса
#include <iostream>
using namespace std;

class Demo {
    mutable int i;
    int j;
public:
    int geti() const {
        return i; // здесь все правильно
    }

    void seti(int x) const {
        i = x; // теперь все правильно
    }
```

```
/* Если убрать комментарии вокруг этой функции, то программа
компилироваться не будет
    void setj(int x) const {
        j = x; // здесь прежняя ошибка
    }
*/
};

int main()
{
    Demo ob;

    ob.seti(1900);
    cout << ob.geti();

    return 0;
}
```

Здесь переменная i задана модифицируемой, поэтому ее может изменить функция-член **seti()**. Тем не менее, поскольку переменная j по-прежнему остается не модифицируемой, постоянная функция-член seti() не может изменить ее значение.

Упражнения

1. В следующей программе сделана попытка создать простой таймер для измерения временных интервалов. По истечении каждого такого интервала таймер должен подавать звуковой сигнал. К сожалению, в том виде, в котором программа представлена, она компилироваться не будет. Найдите и исправьте ошибку.

```
// В этой программе имеется ошибка
#include <iostream>
using namespace std;

class CountDown {
    int incr;
    int target;
    int current;
public:
    CountDown(int delay, int i = 1) {
        target = delay;
        incr = i;
        current = 0;
    }

    bool counting() const {
        current += incr;
```

```
        if(current >= target) {
            cout << "\a";
            return false;
        }
        cout << current <<" ";
        return true;
    }
};

int main()
{
    CountDown ob(100, 2);

    while(ob.counting());

    return 0;
}
```

2. Может ли постоянная функция-член вызвать непостоянную функцию? Если нет, то почему?

13.5. Заключительный обзор конструкторов

Хотя тема конструкторов в этой книге уже обсуждалась, некоторые аспекты их применения остались нераскрытыми. Рассмотрим следующую программу:

```
ftinclude<iostream>
using namespace std;

class myclass {
    int a;
public:
    myclass(int x) { a = x; }
    int geta() { return a; }
};

int main()
{
    myclass ob(4);

    cout << ob.geta();

    return 0;
}
```

Здесь у конструктора класса myclass имеется один параметр. Обратите особое внимание на то, как в функции **main()** объявлен объект ob. Значение 4, заданное в скобках сразу за объектом ob, — это аргумент, который передается параметру x конструктора **myclass()** и с помощью которого инициализируется переменная a. Именно такая форма инициализации использовалась в примерах программ, начиная с первых глав этой книги. Однако это не единственный способ инициализации. Рассмотрим, к примеру, следующую инструкцию:

```
myclass ob = 4; // эта инструкция автоматически преобразуется
                // в инструкцию myclass ob(4);
```

Как показано в комментариях, эта форма инициализации автоматически преобразуется в вызов конструктора **myclass()** со значением 4 в качестве аргумента. Таким образом, предыдущая инструкция обрабатывается компилятором так, как будто на ее месте находится инструкция:

```
myclass ob(4);
```

Как правило, всегда, когда у конструктора имеется только один аргумент, можно использовать любую из представленных выше двух форм инициализации объекта. Смысл второй формы инициализации в том, что для конструктора с одним аргументом она позволяет организовать неявное преобразование типа этого аргумента в тип класса, к которому относится конструктор.

Неявное преобразование можно запретить с помощью спецификатора **explicit** (явный). Спецификатор **explicit** применим только к конструкторам. Для конструкторов, заданных со спецификатором **explicit**, допустим только обычный синтаксис. Автоматического преобразования для таких конструкторов не выполняется. Например, если в предыдущем примере конструктор класса myclass объявить со спецификатором **explicit,** то для такого конструктора автоматического преобразования поддерживаться не будет. В представленном ниже классе конструктор **myclass()** объявлен со спецификатором **explicit.**

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

class myclass {
    int a;
public:
    explicit myclass(int x) { a = x; }
    int geta() { return a; }
};
```

Для такого класса допустима только одна форма конструкторов:

```
myclass ob(4);
```

Примеры

1. В классе может быть более одного преобразующего конструктора. Например, рассмотрим следующую версию класса **myclass.**

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

class myclass {
    int a;
public:
    myclass(int x) { a = x; }
    myclass(char *str) { a = atoi(str); }
    int geta() { return a; }
};

int main()
{
    // преобразование в вызов конструктора myclass ob(4)
    myclass ob1 = 4;

    // преобразование в вызов конструктора myclass ob("123")
    myclass ob2 = "123";

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    return 0;
}
```

Поскольку типы аргументов обоих конструкторов различны (как это и должно быть) каждая инструкция инициализации автоматически преобразуется в соответствующий вызов конструктора.

2. Автоматическое преобразование на основе типа первого аргумента конструктора в вызов самого конструктора имеет интересное применение. Например, для класса **myclass** из примера 1, чтобы присвоить объектам **ob1** и **ob2** новые значения, функция **main()** выполняет преобразования из типа **int** в тип **char *.**

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

class myclass {
    int a;
public:
    myclass(int x) { a = x; }
    myclass(char *str) { a = atoi(str); }
```

```
    int geta() { return a; }
};

int main()
{
    // преобразование в вызов конструктора myclass ob(4)
    myclass ob1 = 4;

    // преобразование в вызов конструктора myclass ob("123")
    myclass ob2 = "123";

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    /* использование автоматического преобразования для присваивания
новых значений
*/
    // преобразование в вызов конструктора myclass ob("1776")
    myclass ob1 = "1776";

    // преобразование в вызов конструктора myclass ob(2001)
    myclass ob1 = 2001;

    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    return 0;
}
```

3. Чтобы запретить показанные в предыдущих примерах преобразования, для конструкторов можно задать спецификатор **explicit:**

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

class myclass {
    int a;
public:
    explicit myclass(int x) { a = x; }
    explicit myclass(char *str) { a = atoi(str); }
    int geta() { return a; }
};

int main()
{
    // преобразование в вызов конструктора myclass ob(4)
    myclass ob1 = 4;

    // преобразование в вызов конструктора myclass ob("123")
    myclass ob1 = "123";
```

```
    cout << "ob1: " << ob1.geta() << endl;
    cout << "ob2: " << ob2.geta() << endl;

    return 0;
}
```

1. В примере 3, если спецификатор **explicit** указать только для конструктора **myclass(int)**, можно ли будет выполнить неявное преобразование также и для конструктора **myclass(char** *)? (Подсказка: попробуйте и посмотрите, что получится.)
2. Работоспособен ли следующий фрагмент программы?

```
class Demo {
    double x;
public:
    Demo(double i) { x = i; }
    // ...
};

// ...
Demo counter = 10;
```

3. Попытайтесь оправдать введение ключевого слова **explicit.** (Другими словами, объясните, почему неявное преобразование конструкторов в некоторых случаях может оказаться нежелательным.)

13.6. Спецификаторы сборки и ключевое слово asm

В C++ поддерживаются два важных механизма для облегчения связи C++ с другими языками программирования. Первым является *спецификатор сборки (linkage specifier),* который сообщает компилятору, что одна или более функций вашей программы на C++ будет компоноваться с другим языком программирования, который может иметь другие соглашения о передаче параметров процедуре заполнения стека и т. д. Вторым является ключевое слово **asm,** которое позволяет вставлять в исходную программу команды ассемблера. Оба этих механизма рассматриваются в этом разделе.

По умолчанию все функции программы на C++ компилируются и компонуются как функции C++. Однако компилятору C++ можно сообщить, что функция будет компоноваться как функция, написанная на другом языке

программирования. Все компиляторы C++ допускают компоновку функций либо как функций C, либо как функций C++. Некоторые также допускают компоновку функций для таких языков, как Pascal, Ada или FORTRAN. Чтобы компоновать функции для других языков программирования, используется следующая основная форма спецификатора сборки:

```
extern "язык" прототип_функции;
```

Здесь *язык* — это название языка программирования, как функцию которого вы хотите компоновать вашу функцию. Если необходимо использовать спецификатор сборки более чем для одной функции, используется такая его основная форма:

```
extern "язык" {
прототипы_функций;
}
```

Все спецификаторы сборки должны быть глобальными; их нельзя задавать внутри функций.

Чаще всего в программы на C++ приходится вставлять фрагменты программ на C. Путем задания сборки с "C" вы предотвращаете *искажения (mangling)* имен функций информацией о типе. Поскольку в C++ имеется возможность перегружать функции и создавать функции-члены, то каждому имени функции обычно сопутствует некоторая информация о ее типе. С другой стороны, так как язык C не поддерживает ни перегрузки функций, ни функций-членов, он не может распознавать имена функций, искаженные информацией об их типе. Указание компилятору необходимости сборки с "C" позволяет решить проблему.

Хотя вполне возможно совместно компоновать ассемблерные подпрограммы с программами на C++, часто легче использовать язык ассемблера в процессе написания программ на C++. В C++ поддерживается специальное ключевое слово **asm,** позволяющее вставлять ассемблерные инструкции в функции C++. Преимущество встроенного ассемблера в том, что ваша программа полностью компилируется как программа на C++, и нет необходимости раздельно с ней компилировать, а затем совместно компоновать ассемблерные файлы. Здесь показана основная форма ключевого слова **asm:**

```
asm ("ас_инструкция");
```

где *ас_инструкция* — это ассемблерная инструкция, которая будет встроена в вашу программу.

Важно отметить, что в некоторых компиляторах поддерживаются следующие три, несколько иные формы инструкции **asm:**

```
asm ас_инструкция;
asm ас_инструкция физический_конец_строки
```

```
asm {
    последовательность ассемблерных инструкций
}
```

Здесь ассемблерные инструкции не выделяются кавычками. Для правильного встраивания ассемблерных инструкций вам понадобится изучить техническую документацию на ваш компилятор.

В среде программирования Microsoft Visual C++ для встраивания ассемблерного кода используется инструкция __asm. *Во всем остальном эта инструкция аналогична только что описанной инструкции* ***asm.***

1. В этой программе функция func() компонуется не как функция C++, а как функция C:

```
// Демонстрация спецификатора сборки
#include <iostream>
using namespace std;

extern "C" int func(int x); // компонуется как функция C

// Теперь функция компонуется как функция C.
int func(int x)
{
    return x/3;
}
```

Теперь эта функция может компоноваться с программой, которая компилировалась компилятором C.

2. В представленной ниже программе компилятору сообщается, что функции **f1()**, **f2()** и **f3()** должны компоноваться как функции C:

```
extern "C" {
    void f1();
    int f2(int x);
    double f3(double x, int *p);
}
```

3. В этом фрагменте в функцию **func()** вставляется несколько ассемблерных инструкций:

```
// Не пытайтесь выполнить эту функцию!
void func()
{
    asm ("mov bp, sp");
    asm ("push ax");
    asm ("mov cl, 4");
    // ...
}
```

Для успешного использования встроенного ассемблера вы должны быть опытным программистом на языке ассемблера. Кроме этого, нужно тщательно изучить техническую документацию на ваш компилятор.

Изучите те разделы технической документации на ваш компилятор, которые относятся к спецификаторам сборки и интерфейсу с ассемблером.

13.7. Массивы в качестве объектов ввода/вывода

Помимо ввода/вывода на экран и в файл, в C++ поддерживается целый ряд функций, в которых в качестве устройств для ввода и вывода используются массивы. Хотя ввод/вывод с использованием массивов *(array-based I/O)* в C++ концептуально перекликается с аналогичным вводом/выводом в C (в особенности это касается функций **sscanf()** и **sprintf()** языка C), ввод/вывод с использованием массивов в C++ более гибкий и полезный, поскольку он позволяет интегрировать в него определенные пользователем типы данных. Хотя охватить все аспекты массивов в качестве объектов ввода/вывода невозможно, здесь рассматриваются их наиболее важные и часто используемые свойства.

Важно понимать, что для реализации ввода/вывода с использованием массивов тоже нужны потоки. Все, что вы узнали о вводе/выводе C++ из глав 8 и 9 применимо и к вводу/выводу с использованием массивов. При этом, чтобы узнать о всех достоинствах массивов в качестве объектов ввода/вывода, вам следует познакомиться с несколькими новыми функциями. Эти функции предназначены для связывания нужного потока с некоторой

областью памяти. После того как эта операция выполнена, весь ввод/вывод производится посредством тех самых функций для ввода и вывода, о которых вы уже знаете.

Перед тем как начать пользоваться массивами в качестве объектов ввода/вывода, необходимо удостовериться в том, что в вашу программу включен заголовок **<strstream>**. В этом заголовке определяются классы **istrstream**, **ostrstream** и **strstream**. Эти классы образуют, соответственно, основанные на использовании массивов потоки для ввода, вывода и ввода/вывода. Базовым для этих классов является класс **ios,** поэтому все функции и манипуляторы классов **istream**, **ostream** и **iostream** имеются также и в классах **istrstream, ostrstream** и **strstream.**

Для вывода в символьный массив используйте следующую основную форму конструктора класса **ostrstream:**

```
ostrstream поток_вывода (char *буфер, streamsize размер,
                         openmode режим = ios::out);
```

Здесь ***поток_вывода*** — это поток, который связывается с массивом, заданным через указатель ***буфер.*** Параметр ***размер*** определяет размер массива. Параметр ***режим*** по умолчанию обычно настроен на обычный режим вывода, но вы можете задать любой, определенный в классе **ios,** флаг режима вывода. (За подробностями обращайтесь к главе 9.)

После того как массив открыт для вывода, символы будут выводиться в массив вплоть до его заполнения. Переполнения массива произойти не может. Любая попытка переполнения массива приведет к ошибке ввода/вывода. Для определения количества записанных в массив символов используйте приведенную ниже функцию-член **pcount()**:

```
int pcount();
```

Функция должна вызываться только в связи с потоком и возвращает она число символов, записанных в массив, включая нулевой символ завершения.

Чтобы открыть массив для ввода из него, используйте следующую форму конструктора класса **istrstream:**

```
istrstream поток_ввода (const char *буфер);
```

Здесь ***буфер*** — это указатель на массив, из которого будут вводиться символы. Поток ввода обозначен через ***поток_ввода.*** **При** считывании входной информации из массива, функция **eof()** возвращает истину при достижении конца массива.

Чтобы открыть массив для ввода/вывода, используйте следующую форму конструктора класса **strstream:**

```
strstream поток_ввод_вывод (char *буфер, streamsize размер,
                            openmode режим = ios::in | ios::out);
```

Здесь ***поток_ввод_вывод*** — это поток ввода/вывода, для которого в качестве объекта ввода/вывода через указатель ***буфер*** задан массив длиной в ***размер*** символов.

Важно помнить, что все описанные ранее функции ввода/вывода работают и с массивами, включая функции ввода/вывода двоичных файлов и функции произвольного доступа.

Применение потоковых классов для символьных массивов резко осуждается стандартом Standard C++. Это означает, что хотя сейчас потоковые классы для символьных массивов достаточно широко распространены, в будущих версиях языка C++ они могут не поддерживаться. Для решения тех же задач, для которых предназначены символьные массивы, стандартом Standard C++ рекомендуются классы-контейнеры, описанные в главе 14.

1. В этом простом примере показано, как открыть массив для вывода и записать в него данные:

```
// Короткий пример вывода в массив
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    char buf[255]; // буфер для вывода

    ostrstream ostr(buf, sizeof buf); // открытие массива для вывода

    ostr << "ввод/вывод через массивы работает с потоками\n";
    ostr << "точно так же, как при обычном вводе/выводе\n " << 100;
    ostr << ' ' << 123.23 << '\n';

    // можно также использовать манипуляторы
    ostr << hex << 100 << ' ';
    // или флаги формата
    ostr.setf(ios::scientific);
    ostr << 123.23 << '\n';
    ostr << ends;

    // вывод на экран полученной строки
    cout << buf;

    return 0;
}
```

В результате выполнения программы на экран выводится следующее:

```
ввод/вывод через массивы работает с потоками
точно так же, как при обычном вводе/выводе
100 123.23
64 01.2323e+02
```

Как вы могли заметить, перегруженные операторы ввода/вывода, встроенные манипуляторы ввода/вывода, функции-члены и флаги формата полностью доступны и для ввода/вывода с использованием массивов. (Это также относится ко всем манипуляторам и перегруженным операторам ввода/вывода, которые вы создаете для своих классов.)

В представленной выше программе с помощью манипулятора **ends** в массив специально добавляется завершающий нулевой символ. Будет ли нулевой символ занесен в массив автоматически или нет, зависит от реализации. Поэтому, если это важно для вашей программы, лучше специально записать в массив завершающий нулевой символ.

2. Пример ввода данных из массива:

```
// Пример ввода из массива
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    char buf[] = "Привет 100 123.125 a";

    istrstream istr(buf); // открытие массива для ввода из него

    int i;
    char str[80];
    float f;
    char c;

    istr >> str >> i >> f >> c;

    cout << str << ' ' << i << ' ' << f;
    cout << ' ' << c << '\n';

    return 0;
}
```

Эта программа считывает и воспроизводит данные, содержащиеся в массиве, заданном указателем **buf**.

3. Запомните, что массив для ввода, после того как он связан с потоком, становится похожим на файл. Например, в следующей программе для считывания содержимого массива по адресу **buf** используются функции **eof() и get():**

```
/* Демонстрация того факта, что функции eof() и get() работают
с вводом/выводом, основанным на использовании массивов
*/
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    char buf[] = "Привет 100 123.125 a";

    istrstream istr(buf);
    char c;

    while(!istr.eof()) {
        istr.get(c);
        if (!istr.eof())cout << c;
    }

    return 0;
}
```

4. В следующей программе выполняется ввод данных из массива и вывод данных в массив:

```
// Демонстрация ввода/вывода с использованием массивов
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    char iobuf[255];

    strstream iostr (iobuf, sizeof iobuf);

    iostr << "Это проверка\n";
    iostr << 100 << hex << ' ' << 100 << ends;

    char str[80];
    int i;

    iostr.getline(str, 79); // считывает строку вплоть до \n
    iostr >> dec >> i; // считывает 100

    cout << str << ' ' << i << ' ';
                    t
    iostr >> hex >> i;
    cout << hex<< i;

    return 0;
}
```

Эта программа сначала записывает информацию в массив по адресу **iobuf**, а затем считывает ее обратно. Сначала с помощью функции **getline()** строка **"Это проверка"** целиком считывается в массив, далее считывается десятичное значение 100, а затем шестнадцатеричное 0x64.

Упражнения

1. Модифицируйте пример 1 так, чтобы перед завершением программы на экран было выведено число символов, записанных в массив по адресу **buf.**
2. Используя массивы в качестве объектов ввода/вывода, напишите программу для копирования содержимого одного массива в другой. (Конечно это не самый эффективный способ решения подобной задачи.)
3. Используя массивы в качестве объектов ввода/вывода, напишите программу для преобразования строки, содержащей значение с плавающей точкой, в число с плавающей точкой.

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Что отличает статическую переменную-член от других переменных-членов?
2. Какой заголовок необходимо включить в вашу программу, чтобы в качестве объектов ввода/вывода можно было использовать массивы?
3. Не считая того, что при вводе/выводе через массивы память используется в качестве устройства ввода и/или вывода, имеется ли еще какое либо отличие между таким и "обычным" вводом/выводом?
4. Для заданной функции **counter()** напишите инструкцию, позволяющую компилировать эту функцию для сборки с языком C.
5. Для чего нужна функция преобразования?
6. Объясните назначение спецификатора **explicit**.
7. Какое имеется принципиальное ограничение на использование постоянных функций-членов?
8. Объясните понятие пространств имен.
9. Для чего нужно ключевое слово **mutable**?

Проверка усвоения материала в целом

В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Поскольку для конструктора с одним аргументом преобразование типа этого аргумента в тип класса, в котором определен конструктор, происходит автоматически, исчезает ли в такой ситуации необходимость в использовании перегруженного оператора присваивания?
2. Можно ли в постоянной функции-члене использовать оператор **const_cast,** чтобы разрешить этой функции-члену модифицировать вызвавший ее объект?
3. Философский вопрос: поскольку библиотека исходного C++ содержится в глобальном пространстве имен и для старых программ на C++ это уже свершившийся факт, какая польза от размещения указанной библиотеки в пространстве имен **std** "задним числом"?
4. Вернитесь к примерам первых двенадцати глав. Подумайте о том, в каких из них функции-члены можно было бы сделать постоянными или статическими. Не те ли это примеры, в которых определение пространств имен наиболее предпочтительно?

Глава 14

Библиотека стандартных шаблонов

Поздравляем! Если при изучении предыдущих глав этой книги вы действительно работали, то теперь можете с полным правом называть себя состоявшимся программистом на C++. В этой последней главе мы расскажем об одном из наиболее увлекательных и совершенных инструментов языка программирования C++ — библиотеке стандартных шаблонов (Standard Template Library, STL).

Библиотека стандартных шаблонов не являлась частью исходной спецификации C++, а была добавлена к ней позже, в процессе стандартизации, на что и были направлены основные усилия разработчиков. Библиотека стандартных шаблонов обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. Например, в библиотеке стандартных шаблонов поддерживаются уже известные нам векторы (vectors), списки (lists), очереди (queues) и стеки (stacks). В ней также определены различные процедуры доступа к этим структурам данных. Поскольку библиотека стандартных шаблонов строится на основе классов-шаблонов, входящие в нее алгоритмы и структуры применимы почти ко всем типам данных.

Рассказ о библиотеке стандартных шаблонов необходимо начать с признания того факта, что она представляет собой вершину искусства программирования, и в ней используются самые изощренные свойства C++. Чтобы научиться понимать и применять библиотеку стандартных шаблонов, вам следует досконально освоить материал предыдущих глав и уметь свободно оперировать полученными знаниями. В особенности это касается шаблонов. Синтаксис шаблонов, на котором написана библиотека стандартных шаблонов, может показаться совершенно устрашающим, но не надо бояться, он выглядит сложнее, чем это есть на самом деле. Помните, в этой главе нет ничего более сложного, чем то, с чем вы уже познакомились в предыдущих главах книги, поэтому не надо расстраиваться или пугаться, если на первых порах библиотека стандартных шаблонов покажется вам непонятной. Немного терпения, усидчивости, экспериментов и, главное, не позволяйте незнакомому синтаксису заслонить от вас исходную простоту библиотеки стандартных шаблонов.

Библиотека стандартных шаблонов достаточно велика, поэтому вы узнаете здесь далеко не обо всех ее свойствах. Фактически, полного описания биб-

лиотеки, всех ее свойств, нюансов и приемов программирования хватило бы на большую отдельную книгу. Представленный в этой главе обзор предназначен для того, чтобы познакомить вас с ее базовыми операциями, философией, основами программирования. После усвоения этого материала вы, несомненно, легко сможете проделать оставшуюся часть пути самостоятельно.

Помимо библиотеки стандартных шаблонов в этой главе описан один из наиболее важных новых классов C++ — *строковый класс (string class)*. Строковый класс определяет строковый тип данных, что позволяет с помощью операторов работать с символьными строками почти так же, как это делается с данными других типов.

Повторение пройденного

Перед тем как продолжить, необходимо правильно ответить на следующие вопросы и сделать упражнения.

1. Объясните, зачем в C++ были добавлены пространства имен.
2. Как задать постоянную функцию-член?
3. Модификатор mutable (модифицируемый) позволяет пользователю вашей программы изменить библиотечную функцию. Так ли это?
4. Дан следующий класс:

```
class X (
    int a, b;
public:
    X(int i, int j) { a = i, b = j; }
    // создайте здесь функцию преобразования в целое
};
```

   Создайте функцию преобразования, возвращаемым значением которой была бы сумма переменных a и **b**.

5. Статическая переменная — член класса может использоваться еще до создания объекта этого класса. Так ли это?
6. Дан следующий класс:

```
class Demo {
    int a;
public:
    explicit Demo(int i) { a = i; }
    int geta() { return a,- }
};
```

   Допустимо ли следующее объявление:

```
Demo o = 10;
```

14.1. Знакомство с библиотекой стандартных шаблонов

Хотя библиотека стандартных шаблонов достаточно велика, а ее синтаксис иногда пугающе сложен, с ней гораздо проще работать, если понять, как она образована и из каких элементов состоит. Поэтому перед изучением примеров программ вполне оправдано дать ее краткий обзор.

Ядро библиотеки стандартных шаблонов образуют три основополагающих элемента: контейнеры, алгоритмы и итераторы. Эти элементы функционируют в тесной взаимосвязи друг с другом, обеспечивая искомые решения проблем программирования.

Контейнеры (containers) — это объекты, предназначенные для хранения других объектов. Контейнеры бывают различных типов. Например, в классе **vector** (вектор) определяется динамический массив, в классе **queue** (очередь) — очередь, в классе **list** (список) — линейный список. Помимо базовых контейнеров, в библиотеке стандартных шаблонов определены также *ассоциативные контейнеры (associative containers),* позволяющие с помощью ключей (keys) быстро получать хранящиеся в них значения. Например, в классе **tap** (ассоциативный список) определяется ассоциативный список, обеспечивающий доступ к значениям по уникальным ключам. То есть, в ассоциативных списках хранятся пары величин ключ/значение, что позволяет при наличии ключа получить соответствующее ключу значение.

В каждом классе-контейнере определяется набор функций для работы с этим контейнером. Например, список содержит функции для вставки, удаления и слияния (merge) элементов. В стеке имеются функции для размещения элемента в стеке и извлечения его из стека.

Алгоритмы (algorithms) выполняют операции над содержимым контейнеров. Существуют алгоритмы для инициализации, сортировки, поиска или замены содержимого контейнеров. Многие алгоритмы предназначены для работы с *последовательностью (sequence),* которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) — это объекты, которые по отношению к контейнерам играют роль указателей. Они позволяют получать доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива. Имеется пять типов итераторов, которые описаны ниже:

Итератор	Описание
Произвольного доступа (random access)	Используется для считывания и записи значений. Доступ к элементам произвольный
Двунаправленный (bidirectional)	Используется для считывания и записи значений. Может проходить контейнер в обоих направлениях

(продолжение)

Итератор	Описание
Однонаправленный (forward)	Используется для считывания и записи значений. Может проходить контейнер только в одном направлении
Ввода (input)	Используется только для считывания значений. Может проходить контейнер только в одном направлении
Вывода (output)	Используется только для записи значений. Может проходить контейнер только в одном направлении

(Не запутайтесь. По аналогии с потоковым вводом/выводом под *вводом* понимается ввод информации *из* контейнера, т. е. считывание, а под *выводом* — вывод информации *в* контейнер, т. е. запись, — *примеч. пер.)*

Как правило, итератор с большими возможностями доступа к содержимому контейнера может использоваться вместо итератора с меньшими возможностями. Например, однонаправленным итератором можно заменить итератор ввода.

С итераторами можно работать точно так же, как с указателями. Над ними можно выполнять операции инкремента и декремента. К ним можно применить оператор *. Типом итераторов объявляется тип **iterator,** который определен в различных контейнерах.

В библиотеке стандартных шаблонов также поддерживаются *обратные итераторы (reverse iterators)*. Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении. То есть, если обратный итератор указывает на последний элемент последовательности, то инкремент этого итератора приведет к тому, что он будет указывать на элемент перед последним.

При упоминании различных типов итераторов в описаниях шаблонов, в данной книге будут использоваться следующие термины:

Термин	Тип итератора
RandIter	Произвольного доступа (random access)
BiIter	Двунаправленный (bidirectional)
ForIter	Однонаправленный (forward)
InIter	Ввода (input)
OutIter	Вывода (output)

Вдобавок к контейнерам, алгоритмам и итераторам, в библиотеке стандартных шаблонов поддерживается еще несколько стандартных компонентов. Главными среди них являются распределители памяти, предикаты и функции сравнения.

У каждого контейнера имеется определенный для него *распределитель памяти (allocator),* который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса **allocator,** но вы можете определить собственный распределитель памяти, если хотите возложить на него какие-нибудь необычные функции. В большинстве случаев достаточно распределителя памяти, заданного по умолчанию.

В некоторых алгоритмах и контейнерах используется функция особого типа, называемая *предикатом (predicate).* Предикат может быть бинарным или унарным. У унарного предиката один аргумент, а у бинарного — два. Возвращаемым значением этих функций является значения истина либо ложь. Точные условия получения того или иного значения определяются программистом. Все унарные предикаты, которые будут упомянуты в этой главе, имеют тип **UnPred,** а все бинарные — **BinPred.** Аргументы бинарного предиката всегда расположены по порядку: *первый, второй.* Тип аргументов как унарного, так и бинарного предиката соответствует типу хранящихся в контейнере объектов.

В некоторых алгоритмах и классах используется специальный тип бинарного предиката, предназначенный для сравнения двух элементов. Такой предикат называется *функцией сравнения (comparison function).* Функция сравнения возвращает истину, если ее первый аргумент меньше второго. Типом функции сравнения является тип **Сотр.**

Помимо заголовков для разнообразных классов-контейнеров, входящих в библиотеку стандартных шаблонов, стандартная библиотека C++ включает также заголовки **<utility> и <functional>,** предназначенные для поддержки классов-шаблонов. Например, заголовочный файл **<utility>** содержит определение класса-шаблона **pair** (пара), в котором могут храниться пары значений. Позднее в этой главе мы еще воспользуемся шаблоном **pair**.

Шаблоны из заголовочного файла **<functional>** помогают создавать объекты, определяющие оператор-функцию **operator()**. Эти объекты называются *объектами-функциями (function objects)* и во многих случаях могут использоваться вместо указателей на функцию. В заголовочном файле **<functional>** объявлено несколько встроенных объектов-функций, некоторые из которых перечислены ниже:

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

Вероятно, чаще других применяется объект-функция **less** (меньше), которая позволяет определить, является ли значение одного объекта меньше, чем значение другого. В описываемых далее алгоритмах библиотеки стандартных шаблонов объектами-функциями можно заменять указатели на реальные функции. Если использовать объекты-функции вместо указателей на функцию, библиотека стандартных шаблонов будет генерировать более эффек-

тивный код. Тем не менее для целей данной главы (обзор библиотеки стандартных шаблонов) объекты-функции не нужны и непосредственно применяться не будут. Хотя сами по себе объекты-функции не представляют особой сложности, их подробное обсуждение достаточно продолжительно и выходит за рамки нашей книги. Этот материал вам следует освоить самостоятельно, если в будущем вы захотите использовать библиотеку стандартных шаблонов с максимальной эффективностью.

1. Что представляют собой контейнеры, алгоритмы и итераторы, входящие в библиотеку стандартных шаблонов?
2. Какие вы знаете два типа предикатов?
3. Какие существуют пять типов итераторов?

14.2. Классы-контейнеры

Ранее уже объяснялось, что контейнерами называются объекты библиотеки стандартных шаблонов, непосредственно предназначенные для хранения данных. В табл. 14.1 перечислены контейнеры, определенные в библиотеке стандартных шаблонов, а также заголовки, которые следует включить в программу, чтобы использовать тот или иной контейнер. Хотя строковый класс, который управляет символьными строками, также является контейнером, ему будет посвящен отдельный раздел.

Таблица 14.1. *Контейнеры, определенные в библиотеке стандартных шаблонов*

Контейнер	Описание	Заголовок
bitset	Множество битов	<bitset>
deque	Двусторонняя очередь	<deque>
list	Линейный список	<list>
map	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано только одно значение	<map>
multimap	Ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано два или более значений	<map>
multiset	Множество, в котором каждый элемент не обязательно уникален	<set>
priority_queue	Очередь с приоритетом	<queue>
queue	Очередь	<queue>

Таблица 14.1 (продолжение)

Контейнер	Описание	Заголовок
set	Множество, в котором каждый элемент уникален	<set>
stack	Стек	<stack>
vector	Динамический массив	<vector>

Поскольку имена типов элементов, входящих в объявление класса-шаблона, могут быть самыми разными, в классах-контейнерах с помощью ключевого слова **typedef** объявляются некоторые согласованные версии этих типов. Эта операция позволяет конкретизировать имена типов. Ниже представлены имена типов, конкретизированные с помощью ключевого слова **typedef,** которые можно встретить чаще других:

Согласованное имя типа	Описание
size_type	Интегральный тип, эквивалентный типу **size_t**
reference	Ссылка на элемент
const_reference	Постоянная ссылка на элемент
iterator	Итератор
const_iterator	Постоянный итератор
reverse_iterator	Обратный итератор
const_reverse_iterator	Постоянный обратный итератор
value_type	Тип хранящегося в контейнере значения
allocator_type	Тип распределителя памяти
key_type	Тип ключа
key_compare	Тип функции, которая сравнивает два ключа
value_compare	Тип функции, которая сравнивает два значения

Хотя изучить все контейнеры в рамках одной главы невозможно, в следующих разделах рассказывается о трех из них: векторе, списке и ассоциативном списке. Если вы поймете, как работают эти три контейнера, то с другими классами библиотеки стандартных шаблонов у вас проблем не будет.

14.3. Векторы

Вероятно, самым популярным контейнером является вектор. В классе **vector** поддерживаются динамические массивы. Динамическим массивом называется массив, размеры которого могут увеличиваться по мере необходимости. Как известно, в C++ в процессе компиляции размеры массива фиксируют-

ся. Хотя это наиболее эффективный способ реализации массивов, одновременно он и самый ограниченный, поскольку не позволяет адаптировать размер массива к изменяющимся в процессе выполнения программы условиям. Решает проблему вектор, который выделяет память для массива по мере возникновения потребности в этой памяти. Несмотря на то, что вектор является, по сути, динамическим массивом, для доступа к его элементам подходит обычная индексная нотация, которая используется для доступа к элементам стандартного массива.

Ниже представлена спецификация шаблона для класса **vector:**

```
template<class T, class Allocator = allocator<T>>class vector
```

Здесь T — это тип предназначенных для хранения в контейнере данных, а ключевое слово **Allocator** задает распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе **vector** определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type число, const T &значение = T(),
                const Allocator fia = Allocator());

vector (const vector<T, Allocator>&объект);

template<class InIter>vector(InIter начало, InIter колец,
                const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов — это ***число,*** а каждый элемент равен значению ***значение.*** Параметр ***значение*** может быть значением по умолчанию. В третьей форме конструктора вектор предназначен для одинаковых элементов, каждый из которых — это ***объект.*** Четвертая форма — это конструктор вектора, содержащего диапазон элементов, заданный итераторами ***начало*** **и** ***конец.***

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме этого, для объекта должны быть определены операторы < и ==. Для некоторых компиляторов может потребоваться определить и другие операторы сравнения. (Для получения более точной информации обратитесь к документации на ваш компилятор.) Для встроенных типов данных все указанные требования выполняются автоматически.

Хотя синтаксис шаблона выглядит довольно сложно, в объявлении вектора ничего сложного нет. Ниже представлено несколько примеров такого объявления:

```
vector<int>  iv;     // создание вектора нулевой длины для целых

vector<char> cv(5); // создание пятиэлементного вектора для символов
```

```
vector<char> cv(5, 'x'); // создание и инициализация
                         // пятиэлементного вектора для символов

vector<int> iv2(iv);     // создание вектора для целых
                         // из вектора для целых
```

Для класса **vector** определяются следующие операторы сравнения:

==, <, <=, **!=**, >, **>=**

Кроме этого для класса **vector** определяется оператор индекса **[],** что обеспечивает доступ к элементам вектора посредством обычной индексной нотации, которая используется для доступа к элементам стандартного массива.

В табл. 14.2 представлены функции — члены класса **vector.** (Повторяем, не нужно пугаться необычного синтаксиса.) Наиболее важными функциями-членами являются функции **size(), begin(), end(), push_back(),** insert() и **erase()**. Функция **size()** возвращает текущий размер вектора. Эта функция особенно полезна, поскольку позволяет узнать размер вектора во время выполнения программы. Помните, вектор может расти по мере необходимости, поэтому размер вектора необходимо определять не в процессе компиляции, а в процессе выполнения программы.

Функция **begin()** возвращает итератор начала вектора. Функция end() возвращает итератор конца вектора. Как уже говорилось, итераторы очень похожи на указатели и с помощью функций **begin()** и **end()** можно получить итераторы (читай: указатели) начала и конца вектора.

Функция **push_back()** помещает значение в конец вектора. Если это необходимо для размещения нового элемента, вектор удлиняется. В середину вектора элемент можно добавить с помощью функции **insert()**. Вектор можно инициализировать. В любом случае, если в векторе хранятся элементы, то с помощью оператора индекса массива к этим элементам можно получить доступ и их изменить. Удалить элементы из вектора можно с помощью функции **erase()**.

Таблица 14.2. *Функции — члены класса vector*

Функция-член	Описание
template<class InIter> void assign(InIter *начало*, InIter *конец*);	Присваивает вектору последовательность, определенную итераторами ***начало*** и конец
template<class Size, class T> void assign(Size *число*, const T &*значение* = T());	Присваивает вектору ***число*** элементов, причем значение каждого элемента равно параметру значение
reference at(size_type *i*); **const_reference at(size_type *i*) const;**	Возвращает ссылку на элемент, заданный параметром *i*

Таблица 14.2 *(продолжение)*

Функция-член	Описание
reference back(); **const_reference back() const;**	Возвращает ссылку на последний элемент вектора
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента вектора
size_type capacity() const;	Возвращает текущую емкость вектора, т. е. то число элементов, которое можно разместить в векторе без необходимости выделения дополнительной области памяти
void clear();	Удаляет все элементы вектора
bool empty() const;	Возвращает истину, если вызывающий вектор пуст, в противном случае возвращает ложь
iterator end(); **const_iterator end() const;**	Возвращает итератор конца вектора
iterator erase(iterator *i*);	Удаляет элемент, на который указывает итератор /. Возвращает итератор элемента, который расположен следующим за удаленным
iterator erase(iterator *начало*, **iterator *конец*);**	Удаляет элементы, заданные между итераторами ***начало* и *конец*.** Возвращает итератор элемента, который расположен следующим за последним удаленным
reference front(); **const_reference front() const;**	Возвращает ссылку на первый элемент вектора
allocator_type **get_allocator() const;**	Возвращает распределитель памяти вектора
iterator insert(iterator *i*, **const T &*значение* = T());**	Вставляет параметр ***значение*** перед элементом, заданным итератором *i*. Возвращает итератор элемента
void insert(iterator *i*, **size_type *число*,** **const T &*значение*);**	Вставляет ***число*** копий параметра ***значение*** перед элементом, заданным итератором *i*
template<class InIter> **void insert(iterator *i*,** **InIter *начало*,** **InIter *конец*);**	Вставляет последовательность, определенную между итераторами ***начало* и *конец*,** перед элементом, заданным итератором *i*
size_type max_size() const;	Возвращает максимальное число элементов, которое может храниться в векторе
reference operator[] **(size..type *i*) const;** **const_reference operator[]** **(size_type *i*) const;**	Возвращает ссылку на элемент, заданный параметром *i*
void pop_back();	Удаляет последний элемент вектора
void push_back(const **T &*значение*);**	Добавляет в конец вектора элемент, значение которого равно параметру ***значение***

Таблица 14.2 (продолжение)

Функция-член	Описание
reverse_iterator rbegin(); **const_reverse_iterator rbegin() const;**	Возвращает обратный итератор конца вектора
reverse_iterator rend(); **const_reverse_iterator rend() const;**	Возвращает обратный итератор начала вектора
void reserve(size_type *число*);	Устанавливает емкость вектора равной, по меньшей мере, параметру ***число*** элементов
void resize(size_type *число*, **T *значение* = T());**	Изменяет размер вектора в соответствии с параметром ***число.*** Если при этом вектор удлиняется, то добавляемые в конец вектора элементы получают значение, заданное параметром ***значение***
size_type size() const;	Возвращает хранящееся на данный момент в векторе число элементов
void swap(vector<T, **Allocator> &*объект*);**	Обменивает элементы, хранящиеся в вызывающем векторе, с элементами в объекте ***объект***

Примеры

1. В представленном ниже коротком примере показаны основные операции, которые можно выполнять при работе с вектором.

```
// Основные операции вектора
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v; // создание вектора нулевой длины
    int i;

    // вывод на экран размера исходного вектора v
    cout << "Размер = " << v.size() << endl;

    // помещение значений в конец вектора,
    // по мере необходимости вектор будет расти
    for(i=0; i<10; i++) v.push_back(i);

    // вывод на экран текущего размера вектора v
    cout << "Новый размер = " << v.size() << endl;
```

```
    // вывод на экран содержимого вектора v
    cout << "Текущее содержимое:\n";
    for (i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    // помещение новых значений в конец вектора,
    // и опять по мере необходимости вектор будет расти
    for(i=0; i<10; i++) v.push_back(i+10);

    // вывод на экран текущего размера вектора
    cout << "Новый размер = " << v.size() << endl;

    // вывод на экран содержимого вектора
    cout << "Текущее содержимое:\n";
    for (i=0; Kv.sizeO; i++) cout << v[i] << " ";
    cout << endl;

    // изменение содержимого вектора
    for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];

    // вывод на экран содержимого вектора
    cout << "Удвоенное содержимое:\n";
    for (i=0; Kv.sizeO; i++) cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер = О
Новый размер = 10
Текущее содержимое:
0 1 2 3 4 5 6 7 8 9
Новый размер = 20
Текущее содержимое:
О 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Удвоенное содержимое:
О 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Тщательно проанализируйте программу. В функции **main()** создается вектор v для хранения целых. Поскольку не используется никакой инициализации, это пустой вектор с равной нулю начальной емкостью, то есть это вектор нулевой длины. Этот факт подтверждается вызовом функции-члена **size()**. Далее с помощью функции-члена **push_back()** к концу вектора v добавляется десять элементов. Чтобы разместить эти новые элементы, вектор v вынужден увеличиться. Как показывает выводимая на экран информация, его размер стал равным 10. После этого выводится содержимое вектора v. Обратите внимание, что для этого используется обычный оператор индекса массива. Далее к вектору добав-

ляется еще десять элементов и, чтобы их разместить, вектор снова автоматически увеличивается. В конце концов, с помощью стандартного оператора индекса массива меняются значения элементов вектора.

В программе есть еще кое-что интересное. Отметьте, что функция **v.size()** указана прямо в инструкции организации цикла вывода на экран содержимого вектора v. Одним из преимуществ векторов по сравнению с массивами является то, что вы всегда имеете возможность определить текущий размер вектора. Очевидно, что такая возможность может оказаться полезной в самых разных ситуациях.

2. Как вы знаете, в C++ массивы и указатели очень тесно связаны. Доступ к массиву можно получить либо через оператор индекса, либо через указатель. По аналогии с этим в библиотеке стандартных шаблонов имеется тесная связь между векторами и итераторами. Доступ к членам вектора можно получить либо через оператор индекса, либо через итератор. В следующем примере показаны оба этих подхода.

```
// Организация доступа к вектору с помощью итератора
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v; // создание вектора нулевой длины
    int i;

    // помещение значений в вектор
    for(i=0; i<10; i++) v.push_back(i);

    // доступ к содержимому вектора
    // с использованием оператора индекса
    for(i=0; i<10; i++) cout << v[i] << " ";
    cout << endl;

    // доступ к вектору через итератор
    vector<int>::iterator p = v.begin();
    while(p != v.end()) {
        cout << *p << " ";
        p++;
    }

    return 0;
}
```

После выполнения программы на экране появится следующее:

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

В этой программе тоже сначала создается вектор v нулевой длины. Далее с помощью функции-члена push_back() к концу вектора v добавляются некоторые значения и размер вектора v увеличивается.

Обратите внимание на объявление итератора **p.** Тип **iterator** определяется с помощью класса-контейнера. То есть, чтобы получить итератор для выбранного контейнера, объявить его нужно именно так, как показано в примере: просто укажите перед типом **iterator** имя контейнера. С помощью функции-члена begin() итератор инициализируется, указывая на начало вектора. Возвращаемым значением этой функции как раз и является итератор начала вектора. Теперь, применяя к итератору оператор инкремента, можно получить доступ к любому выбранному элементу вектора. Этот процесс совершенно аналогичен использованию указателя для доступа к элементам массива. С помощью функции-члена **end()** определяется факт достижения конца вектора. Возвращаемым значением этой функции является итератор того места, которое находится сразу за последним элементом вектора. Таким образом, если итератор p равен возвращаемому значению функции **v.end()**, значит, конец вектора был достигнут.

3. Помимо возможности размещения элементов в конце вектора, с помощью функции-члена **insert()** их можно вставлять в его середину. Удалять элементы из вектора можно с помощью функции-члена **erase().**

```
// Демонстрация функций insert() и erase()
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v(5, 1); // создание пятиэлементного вектора
                         // из единиц
    int i;

    // вывод на экран исходных размера и содержимого вектора
    cout << "Размер = " << v.size() << endl;
    cout << "Исходное содержимое:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    vector<int>::iterator p = v.begin();
    p += 2; // указывает на третий элемент

    // вставка в вектор на то место,
    // куда указывает итератор p десяти новых элементов,
    // каждый из которых равен 9
    v.insert(p, 10, 9);

    // вывод на экран размера
    // и содержимого вектора после вставки
    cout << "Размер после вставки = " << v.size() << endl;
```

```
    cout << "Содержимое после вставки:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    // удаление вставленных элементов
    p = v.begin();
    p += 2; // указывает на третий элемент
    v.erase(p, p+10); // удаление следующих десяти элементов
                      // за элементом, на который указывает
                      // итератор p

    // вывод на экран размера
    // и содержимого вектора после удаления
    cout << "Размер после удаления = " << v.size() << endl;
    cout << "Содержимое после удаления:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер = 5
Исходное содержимое:
1 1 1 1 1

Размер после вставки = 15
Содержимое после вставки:
1 1 9 9 9 9 9 9 9 9 9 1 1 1

Размер после удаления = 5
Содержимое после удаления:
1 1 1 1 1
```

4. В следующем примере вектор используется для хранения объектов класса, определенного программистом. Обратите внимание, что в классе определяются конструктор по умолчанию и перегруженные версии операторов < и ==. Помните, в зависимости от того, как реализована библиотека стандартных шаблонов для вашего компилятора, вам может понадобиться определить и другие операторы сравнения.

```
// Хранение в векторе объектов пользовательского класса
#include <iostream>
#include <vector>
using namespace std;

class Demo {
    double d;
```

```
public:
    Demo () { d = 0.0; }
    Demo (double x) { d = x; }

    Demo &operator=(double x) {
        d = x; return *this;
    }
    double getd() {return d; }
};

boll operator< (Demo a, Demo b)
{
    return a.getd() < b.getd();
}

boll operator==(Demo a, Demo b)
{
    return a.getd() == b.getd();
}

int main()
{
    vector<Demo> v;
    int i;

    for(i=0; i<10; i++)
        v.push_back(Demo(i/3.0));

    for(i=0; i<v.size(); i++)
        cout << v[i].getd() << " ";

    cout << endl;

    for(i=0; i<v.size(); i++)
        v[i] = v[i].getd() * 2.1;

    for(i=0; i<v.size(); i++)
        cout << v[i].getd() << " ";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
O 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
O 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3
```

1. Поэкспериментируйте с представленными примерами. Попытайтесь делать небольшие изменения в программах и исследуйте результаты.

2. В примере 4 для класса **Demo** были определены два конструктора — конструктор по умолчанию (конструктор без параметров) и конструктор с параметрами. Сможете ли вы объяснить, почему это так важно?
3. Ниже представлен пример класса **Coord.** Напишите программу для хранения объектов типа **Coord** в векторе. (Подсказка: не забудьте для класса **Coord** определить операторы < и ==.)

```
class Coord {
public:
    int x, y;
    Coord() { x = y = 0; }
    Coord(int a, int b) { x = a; y = b; }
};
```

14.4. Списки

Класс **list** поддерживает двунаправленный линейный список. В отличии от вектора, в котором реализован произвольный доступ, к элементам списка доступ может быть только последовательным. Поскольку списки являются двунаправленными, доступ к элементам списка возможен с обеих его сторон.

Ниже представлена спецификация шаблона для класса **list:**

```
template<class T, class Allocator = allocator<T>>class list
```

Здесь T — это тип данных, предназначенных для хранения в списке, а ключевое слово **Allocator** задает распределитель памяти, который по умолчанию является стандартным распределителем памяти. В классе **list** определены следующие конструкторы:

```
explicit list(const Allocator fia - Allocator());

explicit list(size_type число, const T &значение = T(),
              const Allocator Sa = Allocator());

list (const list<T, Allocator>&объект);

template<class InIter>list(InIter начало, InIter конец,
              const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого списка. Вторая форма — конструктор списка, число элементов которого — это ***число,*** а каждый элемент равен значению ***значение,*** которое может быть значением по умолчанию. Третья форма конструктора предназначена для списка из одинаковых элементов, каждый из которых — это ***объект.*** Четвертая форма — это конструктор списка, содержащего диапазон элементов, заданный итераторами ***начало*** и ***конец.***

Для класса **list** определяются следующие операторы сравнения:

```
==, <, <=, !=, >, >=
```

В табл. 14.3 представлены функции — члены класса **list.** Размещать элементы в конце списка можно с помощью функции **push_back()** (как и в случае с вектором), в начале — с помощью функции **push_front()**, а в середине — с помощью функции **insert()**. Для *соединения (join)* двух списков нужна функция **splice(),** а для *слияния (merge)* — функция **merge()**.

Для любого типа данных, которые вы собираетесь хранить в списке, должен быть определен конструктор по умолчанию. Кроме этого, необходимо определить различные операторы сравнения. К моменту написания этой книги точные требования к объектам, предназначенным для хранения в списке, у разных компиляторов были разными, поэтому перед использованием списка тщательно изучите техническую документацию на ваш компилятор.

Таблица 14.3. *Функции — члены класса* ***list***

Функция-член	Описание
`template<classInIter> void assign(InIter начало, InIter конец);`	Присваивает списку последовательность, определенную итераторами ***начало*** и ***конец***
`template<class Size, class T> void assign(Size число, const T &значение = T());`	Присваивает списку ***число*** элементов, причем значение каждого элемента равно параметру **значение**
`reference back();` `const_reference back() const;`	Возвращает ссылку на последний элемент списка
`iterator begin();` `const_iterator begin() const;`	Возвращает итератор первого элемента списка
`void clear();`	Удаляет все элементы списка
`bool empty() const;`	Возвращает истину, если вызывающий список пуст, в противном случае возвращает ложь
`iterator end();` `const_iterator end() const;`	Возвращает итератор конца списка
`iterator erase(iterator i);`	Удаляет элемент, на который указывает итератор ***i***. Возвращает итератор элемента, который расположен следующим за удаленным
`iterator erase(iterator начало, iterator конец);`	Удаляет элементы, заданные между итераторами ***начало*** и ***конец.*** Возвращает итератор элемента, который расположен следующим за последним удаленным
`reference front();` `const_reference front() const;`	Возвращает ссылку на первый элемент списка
`allocator_type get_allocator() const;`	Возвращает распределитель памяти списка

Таблица 14.3 (продолжение)

Функция-член	Описание
`iterator insert(iterator i, const T &значение = T());`	Вставляет параметр ***значение*** перед элементом, заданным итератором /. Возвращает итератор элемента
`void insert(iterator i, size_type число, const T &значение);`	Вставляет ***число*** копий параметра ***значение*** перед элементом, заданным итератором *i*
`template<classInIter> void insert(iterator i, InIter начало, InIter конец);`	Вставляет последовательность, определенную между итераторами ***начало*** и ***конец,*** перед элементом, заданным итератором *i*
`size_type max_size() const;`	Возвращает максимальное число элементов, которое может храниться в списке
`void merge(list<T, Allocator> &объект);` `template<class Comp> void merge(list<T, Allocator> &объект, Comp ф_сравн);`	Выполняет слияние упорядоченного списка, хранящегося в объекте ***объект,*** с вызывающим упорядоченным списком. Результат упорядочивается. После слияния список, хранящийся в объекте ***объект*** становится пустым. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения ***ф_сравн***
`void pop_back();`	Удаляет последний элемент списка
`void pop_front();`	Удаляет первый элемент списка
`void push_back(const T &значение);`	Добавляет в конец списка элемент, значение которого равно параметру ***значение***
`void push_front(const T &значение);`	Добавляет в начало списка элемент, значение которого равно параметру ***значение***
`reverse_iterator rbegin();` `const_reverse_iterator rbegin() const;`	Возвращает обратный итератор конца списка
`void remove(const T &значение);`	Удаляет из списка элементы, значения которых равны параметру ***значение***
`template<class UnPred> void remove_if(UnPred пред);`	Удаляет из списка значения, для которых истинно значение унарного предиката ***пред***
`reversejterator rend();` `const_reverse_iterator rend() const;`	Возвращает обратный итератор начала списка
`void resize(size_type число, T значение = T());`	Изменяет размер списка в соответствии с параметром ***число.*** Если при этом список удлиняется, то добавляемые в конец списка элементы получают значение, заданное параметром ***значение***
`void reverse();`	Выполняет реверс (т. е. реализует обратный порядок расположения элементов) вызывающего списка

Таблица 14.3 (продолжение)

Функция-член	Описание
slze_type size() const;	Возвращает хранящееся на данный момент в списке число элементов
void sort(); **template<class Comp>** **void sort Comp *ф_сравн*);**	Сортирует список. Во второй форме для определения того, является ли значение одного элемента меньшим, чем значение другого, может задаваться функция сравнения ***ф_сравн***
void splice(iterator *i*, **list<T, Allocator> &*объект*);**	Вставляет содержимое объекта ***объект*** в вызывающий список. Место вставки определяется итератором *I*. После выполнения операции ***объект*** становится пустым
void splice(iterator /, **list<T, Allocator> &*объект*,** **iterator *элемент*);**	Удаляет элемент, на который указывает итератор ***элемент,*** из списка, хранящегося в объекте ***объект,*** и сохраняет его в вызывающем списке. Место вставки определяется итератором ***i***
void splice(iterator /, **lisKT, Allocator> &*объект*,** **iterator *начало*,** **iterator *конец*);**	Удаляет диапазон элементов, обозначенный итераторами ***начало*** **и** ***конец,*** из списка, хранящегося в объекте ***объект,*** и сохраняет его в вызывающем списке. Место вставки определяется итератором /'
void swap(list<T, **Allocator> &*объект*);**	Обменивает элементы из вызывающего списка с элементами из объекта ***объект***
void unique(); **template<class BinPred>** **void unique(BinPred *пред*);**	Удаляет из вызывающего списка парные элементы. Во второй форме для выяснения уникальности элементов используется предикат ***пред***

Примеры

1. Ниже представлен пример простого списка.

```
// Основные операции списка
ttinclude <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> lst; // создание пустого списка
    int i;

    for(i=0; i<10; i++) lst.push_back('A' + i);

    cout << "Размер = " << lst.size() << endl;
```

```
    list<char>::iterator p;

    cout << "Содержимое: ";
    while(!lst.empty()) {
        p = lst.begin();
        cout << *p;
        lst.pop_front();
    }

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер = 10
Содержимое: ABCDEFGHIJ
```

В этой программе создается список символов. Сначала создается пустой список. Затем туда помещается десять символов (буквы от A до J включительно). Эта операция выполняется с помощью функции **push_back(),** которая помещает каждое следующее значение в конец существующего списка. Далее размер списка выводится на экран. После этого организуется вывод на экран содержимого списка, для чего каждый раз последовательно извлекают, выводят на экран и удаляют очередной первый элемент списка. Этот процесс продолжается, пока список не опустеет.

2. В предыдущем примере, пройдя список от начала до конца, мы его опустошили. Это, конечно, не обязательно. Ниже представлена переработанная версия программы.

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

int. main()
{
    list<char> lst;
    int i;

    for(i=0; i<10; i++) lst.push_back('A' + i) ;

    cout << "Размер = " << lst.size() << endl;

    list<char>: : iterator p = lst.begin();

    cout << "Содержимое: ";
    while(p != lst.end()) (
        cout << *p;
        P++;
    }
```

```
    return 0;
}
```

В данной программе итератор p инициализируется таким образом, чтобы он указывал на начало списка. Затем при каждом проходе цикла итератор p инкрементируется, что заставляет его указывать на следующий элемент списка. Цикл завершится, когда итератор p укажет на конец списка.

3. Поскольку список является двунаправленным, размещать элементы в нем можно как с начала списка, так и с его конца. В следующей программе создается два списка, причем во втором списке организуется обратный первому порядок расположения элементов.

```
// Элементы можно размещать не только начиная с начала списка,
// но также и начиная с его конца
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> lst;
    list<char> revlst;
    int i;

    for(i=0; i<10; i++) lst.push_back('A' + i);

    cout << "Размер прямого списка = " << lst.size() << endl;
    cout << "Содержимое прямого списка: ";

    list<char>::iterator p;

    // Удаление элементов из первого списка
    // и размещение их в обратном порядке во втором списке
    while(!lst.empty()) {
        p = lst.begin();
        cout << *p;
        lst.pop_front();
        revlst.push_front(*p);
    }
    cout << endl;

    cout << "Размер обратного списка = ";
    cout << revlst.size() << endl;
    cout << "Содержимое обратного списка: ";
    p = revlst.begin();
    while(p != revlst.end()) {
        cout << *p;
        p++;
    }
```

```
    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер прямого списка = 10
Содержимое прямого списка: ABCDEFGHIJ

Размер обратного списка = 10
Содержимое обратного списка: JIHGFEDCBA
```

В данной программе реверс списка **lst** достигается следующим образом: элементы поочередно извлекаются из начала списка **lst** и размещаются в начале списка **revlst.** Таким образом в списке **revlst** реализуется обратный порядок расположения элементов.

4. Вызвав функцию-член **sort()**, вы можете отсортировать список. В следующей программе создается список случайных символов, а затем эти символы сортируются.

```
// Сортировка списка
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
    list<char> lst;
    int i;

    // заполнение списка случайными символами
    for(i=0; i<10; i++) lst.push_back('A' + (rand()%26));

    cout << "Исходное содержимое: ";
    list<char>::iterator p = lst.begin();
    while(p != lst.end()) {
        cout << *p;
        p++;
    }
    cout << endl;

    // сортировка списка
    lst.sort();

    cout << "Отсортированное содержимое: ";
    p = lst.begin();
    while(p != lst.end()) {
        cout << *p;
        p++;
    }
```

```
    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходное содержимое: PHQGHUMEAY
Отсортированное содержимое: AEGHHMPQUY
```

5. Отсортированный список можно слить с другим. В результате будет получен новый отсортированный список с содержимым, состоящим из содержимого обоих исходных списков. Новый список остается в вызывающем списке, а второй список оказывается пустым. Ниже представлен пример слияния двух списков. В первом находятся символы **ACEGI,** а во втором — **BDFHJ.** После слияния мы получим последовательность **ABCDEFGHIJ.**

```
// Слияние двух списков
#include <iostream>
tfinclude<list>
using namespace std;

int main()
{
    list<char> lst1, lst2;
    int i;
    for(i=0; i<10; i+=2) lst1.push_back( 'A' + i);
    for(i=1; i<11; i+=2) lst2.push_back( 'A' + i);

    cout << "Содержимое первого списка: ";
    list<char>::iterator p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p;
        p++;
    }
    cout << endl;

    cout << "Содержимое второго списка: ";
    p = lst2.begin();
    while(p != lst2.end()) {
        cout << *p;
        p++;
    }
    cout << endl;

    // Слияние двух списков
    lst1.merge(lst2);
    if(lst2.empty())
        cout << "Теперь второй список пуст\n";
```

```
    cout << "Содержимое первого списка после слияния:\n";
    p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p;
        P++;
    }

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Содержимое первого списка: ACEGI
Содержимое второго списка: BDFHJ
Теперь второй список пуст
Содержимое первого списка после слияния:
ABCDEFGHIJ
```

6. В следующем примере список используется для хранения объектов типа **Project. Project** — это класс, с помощью которого организуется управление программными проектами. Обратите внимание, что для объектов типа **Project** перегружаются операторы <, >, != и ==. Перегрузки этих операторов требует компилятор Microsoft Visual C++ 5. (Именно этот компилятор использовался при отладке примеров данной главы.) Для других компиляторов может потребоваться перегрузить какие-либо дополнительные операторы. В библиотеке стандартных шаблонов с помощью указанных оператор-функций сравниваются объекты, хранящиеся в контейнере. Хотя список не является контейнером с упорядоченным хранением элементов, тем не менее и здесь при поиске, сортировке или слиянии элементы приходится сравнивать.

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

class Project {
public:
    char name[40];
    int days_to_completion;
    Project() {
        strcpy(name, " ");
        days_to_completion = 0;
    }
    Project(char *n, int d) {
        strcpy(name, n);
        days_to_completion = d;
    }
```

```
    void add_days(int i) {
        days_to_completion += i;
    }

    void sub_days(int i) {
        days_to_completion -= 1;
    }

    bool completed() { return !days_to_completion; }

    void report() {
        cout << name << " : ";
        cout << days_to_completion;
        cout << " дней до завершения\n";
    }
};

bool operator<(const Project &a, const Project &b)
{
    return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project &a, const Project &b)
{
    return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
    return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
    return a.days_to_completion != b.days_to_completion;
}

int main()
{
    list<Project> proj;

  proj.push_back(Project("Разработка компилятора", 35));
  proj.push_back(Project("Разработка электронной таблицы", 190));
  proj.push_back(Project("Разработка STL", 1000));

    list<Project>::iterator p = proj.begin();

    // вывод проектов на экран
    while(p != proj.end()) {
        p->report();
        p++;
    }
```

```
    // увеличение сроков выполнения первого проекта на 10 дней
    p = proj.begin();
    p->add_days(10);

    // последовательное завершение первого проекта
    do {
        p->sub_days(5);
        p->report();
    } while(!p->completed());

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Разработка компилятора: 35 дней до завершения
Разработка электронной таблицы: 190 дней до завершения
Разработка STL: 1000 дней до завершения
Разработка компилятора: 40 дней до завершения
Разработка компилятора: 35 дней до завершения
Разработка компилятора: 30 дней до завершения
Разработка компилятора: 25 дней до завершения
Разработка компилятора: 20 дней до завершения
Разработка компилятора: 15 дней до завершения
Разработка компилятора: 10 дней до завершения
Разработка компилятора: 5 дней до завершения
Разработка компилятора: 0 дней до завершения
```

1. Поэкспериментируйте с представленными примерами. Попытайтесь делать небольшие изменения в программах и исследуйте результаты.
2. В примере 1 после вывода информации на экран список опустел. В примере 2 вы узнали об одном из способов исследования содержимого списка, при котором он остается неповрежденным. Можете ли вы придумать другой способ просмотреть список, не опустошая его при этом? Продемонстрируйте ваше решение, заменив необходимые инструкции в программе из примера 1.
3. Отталкиваясь от программы из примера 6, создайте еще один список, в котором представьте следующие проекты:

Проект	Срок завершения
Разработка базы данных	780
Разработка стандартных писем	50
Разработка объектов COM	300

После создания второго списка выполните сортировку и затем слияние обоих списков. Выведите на экран итоговый результат.

14.5. Ассоциативные списки

Класс **map** поддерживает ассоциативный контейнер, в котором каждому значению соответствует уникальный ключ. По существу, ключ — это просто имя, которое вы присваиваете значению. После того как значение помещено в контейнер, извлечь его оттуда можно с помощью ключа. Таким образом, в самом общем смысле можно сказать, что ассоциативный список представляет собой список пар ключ/значение. Преимущество ассоциативных списков состоит в возможности получения значения по данному ключу. Например, используя ассоциативный список, можно хранить имена телефонных абонентов в качестве ключей, а номера телефонов в качестве значений. Ассоциативные контейнеры в программировании становятся все более и более популярными.

Как уже упоминалось, в ассоциативном списке можно хранить только уникальные ключи. Дублирования ключей не допускается. Для создания ассоциативного списка с неуникальными ключами используется класс-контейнер **multimap**.

Ниже представлена спецификация шаблона для класса **map:**

```
template<class Key, class T, class Comp = less<Key>,
                    class Allocator = allocator<T>>class map
```

Здесь Key — это данные типа ключ, T — тип данных, предназначенных для хранения (в карте), а **Comp** - функция для сравнения двух ключей, которой по умолчанию является стандартная объект-функция **less().** Ключевое слово **Allocator** задает распределитель памяти (которым по умолчанию является **allocator).**

В классе **map** определены следующие конструкторы:

```
explicit map (const Comp &ф_сравн = Comp() ,
              const Allocator &a = Allocator()) ;

map (const map<Key, T, Comp, Allocator>&объект) ;

template<class InIter>map(InIter начало, InIter конец,
const Comp &ф_сравн = Comp() , const Allocator &a = Allocator()) ;
```

Первая форма представляет собой конструктор пустого ассоциативного списка. Вторая форма конструктора предназначена для ассоциативного списка из одинаковых элементов, каждый из которых — это ***объект.*** Третья форма — это конструктор ассоциативного списка, содержащего диапазон элементов, заданный итераторами ***начало*** **и** ***конец.*** Функция сравнения ***ф_сравн***,

если она присутствует, задает порядок сортировки элементов ассоциативного списка.

Как правило, для любого объекта, заданного в качестве ключа, должны быть определены конструктор по умолчанию и несколько операторов сравнения.

Для класса **tap** определяются следующие операторы сравнения:

==, <, <=, !=, >, >=

В табл. 14.4 представлены функции — члены класса tap. В данной таблице тип **key_type** — это тип ключа, а **key_value** — тип пары ключ/значение (тип **pair<Key, T>**).

Таблица 14.4. *Функции — члены класса tap*

Функция-член	Описание
iterator begin(); **const_iterator begin() const;**	Возвращает итератор первого элемента ассоциативного списка
void clear();	Удаляет все элементы ассоциативного списка
size_type count (const key_type &A) const;	Возвращает 1 или 0, в зависимости от того, встречается или нет в ассоциативном списке ключ *k*
bool empty() const;	Возвращает истину, если вызывающий ассоциативный список пуст, в противном случае возвращает ложь
iterator end(); **constjterator end() const;**	Возвращает итератор конца ассоциативного списка
pair<iterator, iterator> equal_range(const key_type &k); **pair<const_iterator, const_iterator> equal_range(const key_type &k) const;**	Возвращает пару итераторов, которые указывают на первый и последний элементы ассоциативного списка, содержащего указанный ключ *k*
void erase(iterator *i*);	Удаляет элемент, на который указывает итератор *i*
void erase (iterator *начало*, iterator *конец*);	Удаляет элементы, заданные между итераторами ***начало*** **и** ***конец***
size_type erase (const key_type &k);	Удаляет элементы, соответствующие значению ключа *k*
iterator find (const key_type &k); **constjterator find (const key_type &k) const;**	Возвращает итератор по заданному ключу *k*. Если ключ не обнаружен, возвращает итератор конца ассоциативного списка
allocator_type get_allocator() const;	Возвращает распределитель памяти ассоциативного списка

Таблица 14.4 *(продолжение)*

Функция-член	Описание
iterator insert(iterator *i*, const value_type &*значение*);	Вставляет параметр ***значение*** на место элемента или после элемента, заданного итератором ***i***. Возвращает итератор этого элемента
template<class InIter> void insert(InIter *начало*, InIter *конец*);	Вставляет последовательность элементов, заданную итераторами ***начало*** и ***конец***
pair<iterator, bool>insert (const value_type &*значение*);	Вставляет ***значение*** в вызывающий ассоциативный список. Возвращает итератор вставленного элемента. Элемент вставляется только в случае, если такого в ассоциативном списке еще нет. При удачной вставке элемента функция возвращает значение **pair<iterator, true>**, в противном случае — **pair<iterator, false>**
key_compare key_comp() const;	Возвращает объект-функцию сравнения ключей
iterator lower_bound (const key_type &*k*); const_iterator lower_bound (const key_type &*k*) const;	Возвращает итератор первого элемента ассоциативного списка, ключ которого равен или больше заданного ключа *k*
size_type max_size() const;	Возвращает максимальное число элементов, которое можно хранить в ассоциативном списке
reference operator[] (const key_type &*i*);	Возвращает ссылку на элемент, соответствующий ключу ***i***. Если такого элемента не существует, он вставляется в ассоциативный список
reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	Возвращает обратный итератор конца ассоциативного списка
reverse_iterator rend(); const_reverse_iterator rend() const;	Возвращает обратный итератор начала ассоциативного списка
size_type size() const;	Возвращает хранящееся на данный момент в ассоциативном списке число элементов
void swap(map<Key, T, Comp, Allocator> &*объект*);	Обменивает элементы из вызывающего ассоциативного списка с элементами из объекта ***объект***
iterator upper_bound (const key_type &*k*); constjterator upper_bound (const key_type &*k*) const;	Возвращает итератор первого элемента ассоциативного списка, ключ которого больше заданного ключа k
value_compare value_comp() const;	Возвращает объект-функцию сравнения значений

В ассоциативном списке хранятся пары ключ/значение в виде объектов типа **pair.** Шаблон объекта типа **pair** имеет следующую спецификацию:

```
template<class Ktype, class Vtype> struct pair {
    typedef Ktype первый_тип; // тип ключа
    typedef Vtype второй_тип; // тип значения
    Ktype первый; // содержит ключ
    Vtype второй; // содержит значение

    // конструкторы
    pair();
    pair(const Ktype &k, const Vtype &v);
    template<class A, class B> pair(const<A, B> &объект)
```

Ранее уже говорилось, что значение переменной ***первый*** содержит ключ и значение, а значение переменной ***второй*** — значение, соответствующее этому ключу.

Создавать пары ключ/значение можно не только с помощью конструкторов класса **pair**, но и с помощью функции **make_pair()**, которая создает объекты типа **pair,** используя типы данных в качестве параметров. Функция **make_pair()** — это родовая функция со следующим прототипом:

```
template<class Ktype, class Vtype>
    pair<Ktype, Vtype>make_pair(const Ktype &k, const Vtype &v);
```

Как видите, функция возвращает объект типа **pair,** содержащий заданные в качестве параметров функции значения типов ***Ktype*** **и** ***Vtype.*** Преимущество использования функции **make_pair()** состоит в том, что она дает возможность компилятору автоматически распознавать типы предназначенных для хранения объектов, и вам не нужно указывать их явно.

Пример

1. В следующей программе на примере ассоциативного списка, предназначенного для хранения десяти пар ключ/значение, иллюстрируются основы использования ассоциативных списков. Ключом здесь является символ, а значением — целое. Пары ключ/значение хранятся следующим образом:

л	О
в	1
с	2
. . .	
J	9

Поскольку пары хранятся именно таким образом, то, когда пользователь набирает на клавиатуре ключ (т. е. одну из букв от А до J), программа выводит на экран соответствующее этому ключу значение.

```
// Иллюстрация возможностей ассоциативного списка
ttinclude<iostream>
#include <map>
using namespace std;

int main()
{
    map<char, int> m;
    int i;

    // размещение пар в ассоциативном списке
    for(i=0; i<10; i++) {
        m.insert(pair<char, int>('A' + i, i));
    }

    char ch;
    cout << "Введите ключ: ";
    cin >> ch;

    map<char, int>::iterator p;

    // поиск значения по заданному ключу
    p = m.find(ch);
    if(p != m.end())
        cout << p->second;
    else
        cout << "Такого ключа в ассоциативном списке нет\n";

    return 0;
}
```

Обратите внимание на использование класса-шаблона **pair** для образования пар ключ/значение. Типы данных, указанные в классе-шаблоне **pair,** должны соответствовать типам данных, хранящимся в ассоциативном списке.

После того как ассоциативный список инициализирован парами ключ/значение, найти нужное значение по заданному ключу можно с помощью функции **find()**. Функция **find()** возвращает итератор соответствующего ключу элемента или итератор конца ассоциативного списка, если указанный ключ не найден. Когда соответствующее ключу значение найдено, оно сохраняется в качестве второго члена класса-шаблона **pair.**

2. В предыдущем примере типы пар ключ/значение были указаны явно в конструкции **pair<char, int>**. Хотя такой подход совершенно правилен, часто проще использовать функцию **make_pair()**, которая создает пары объектов на основе типов данных своих параметров.

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

```
int main ()
{
    map<char, int> m;
    int i;

    // размещение пар в ассоциативном списке
    for(i=0; i<10; i++) {
       m.insert(make_pair(char)('A' + i, i));
    }

    char ch;
    cout << "Введите ключ: ";
    cin >>ch;

    map<char, int>::iterator p;

    // поиск значения по заданному ключу
    p = m.find(ch);
    if (p != m.end())
        cout << p->second;
    else
        cout << "Такого ключа в ассоциативном списке нет\n";

    return 0;
}
```

Данный пример отличается от предыдущего только строкой

```
m.insert(make_pair(char) ('A' + i, i));
```

В данном случае, чтобы в операции сложения 'A' + i не допустить автоматического преобразования в тип **int,** используется приведение к типу **char**. Во всем остальном процесс определения типов объектов выполняется автоматически.

3. Так же как и в других контейнерах, в ассоциативных списках можно хранить создаваемые вами типы данных. Например, в представленной ниже программе создается ассоциативный список для хранения слов с соответствующими словам антонимами. С этой целью используются два класса: **word** (слово) и **opposite** (антоним). Поскольку для ассоциативных списков поддерживается отсортированный список ключей, в программе для объектов типа **word** определяется оператор <. Как правило, оператор < необходимо перегружать для всех классов, объекты которых предполагается использовать в качестве ключей. (Помимо оператора < в некоторых компиляторах может потребоваться определить дополнительные операторы сравнения.)

```
// Ассоциативный список слов и антонимов
#include <iostream>
#include <map>
ttinclude <cstring>
using namespace std;
```

```
class word {
    char str[20];
public:
    word() { strcpy(str, ""); }
    word(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

// для объектов типа word следует определить оператор < (меньше)
bool operator<(word a, word b)
{
    return strcmp(a.get(), b.get()) < 0;
}

class opposite {
    char str[20];
public:
    opposite() { strcmp(str, ""); }
    opposite(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

int main()
{
    map<word, opposite> m;

    // размещение в ассоциативном списке слов и антонимов
    m.insert(pair<word, opposite>
        (word("да"),opposite("нет")));
    m.insert(pair<word, opposite>
        (word("хорошо"),opposite("плохо")));
    m.insert(pair<word, opposite>
        (word("влево"),opposite("вправо")));
    га.insert(pair<word, opposite>
        (word("вверх"),opposite("вниз")));

    // поиск антонима по заданному слову
    char str[80];
    cout << "Введите слово: ";
    cin >> str;

    map<word, opposite>::iterator p;

    p = m.find(word(str));
    if(p != m.end())
        cout << "Антоним: " << p->second.get();
    else
        cout << "Такого слова в ассоциативном списке нет\n";
```

```
    return 0;
}
```

В данном примере любой объект, который вводится в ассоциативный список, представляет собой символьный массив для хранения заканчивающейся нулем строки. Далее в этой главе будет показано, как упростить эту программу, используя стандартный тип данных **string.**

1. Поэкспериментируйте с представленными примерами. Попытайтесь делать небольшие изменения в программах и исследуйте результаты.
2. Создайте ассоциативный список для хранения имен абонентов и их телефонных номеров. Имена и номера телефонов должны вводиться пользователем, а поиск нужного номера должен выполняться по введенному имени абонента. (Подсказка: в качестве модели воспользуйтесь примером 3.)
3. Нужно ли определять для объектов оператор <, если эти объекты используются в качестве ключей ассоциативного списка?

14.6. Алгоритмы

Как уже объяснялось, алгоритмы предназначены для разнообразной обработки контейнеров. Хотя в каждом контейнере поддерживается собственный базовый набор операций, стандартные алгоритмы обеспечивают более широкие и комплексные действия. Кроме этого, они позволяют одновременно работать с двумя контейнерами разных типов. Для доступа к алгоритмам библиотеки стандартных шаблонов в программу необходимо включить заголовок **<algorithm>**.

В библиотеке стандартных шаблонов определяется большое число алгоритмов, которые систематизированы в табл. 14.5. Все алгоритмы представляют собой функции-шаблоны. Это означает, что их можно использовать с контейнерами любых типов. Наиболее показательные варианты такого использования приведены в примерах данного раздела.

Таблица 14.5. *Алгоритмы библиотеки стандартных шаблонов*

Алгоритм	Назначение
adjacent_find	Выполняет поиск смежных парных элементов в последовательности. Возвращает итератор первой пары
binary_search	Выполняет бинарный поиск в упорядоченной последовательности

Таблица *14.5 (продолжение)*

Алгоритм	Назначение
copy	Копирует последовательность
copy_backward	Аналогична функции **copy()** за исключением того, что перемещает в начало последовательности элементы из ее конца
count	Возвращает число элементов в последовательности
count_if	Возвращает число элементов в последовательности, удовлетворяющих некоторому предикату
equal	Определяет идентичность двух диапазонов
equal_range	Возвращает диапазон, в который можно вставить элемент, не нарушив при этом порядок следования элементов в последовательности
fill **fill_n**	Заполняет диапазон заданным значением
find	Выполняет поиск диапазона для значения и возвращает первый найденный элемент
find_end	Выполняет поиск диапазона для подпоследовательности. Функция возвращает итератор конца подпоследовательности внутри диапазона
find_first_of	Находит первый элемент внутри последовательности, парный элементу внутри диапазона
find_if	Выполняет поиск диапазона для элемента, для которого определенный пользователем унарный предикат возвращает истину
for_each	Назначает функцию диапазону элементов
generate **generate_n**	Присваивает элементам в диапазоне значения, возвращаемые порождающей функцией
includes	Определяет, включает ли одна последовательность все элементы другой последовательности
inplace_merge	Выполняет слияние одного диапазона с другим. Оба диапазона должны быть отсортированы в порядке возрастания элементов. Результирующая последовательность сортируется
iter_swap	Меняет местами значения, на которые указывают два итератора, являющиеся аргументами функции
lexicographical_compare	Сравнивает две последовательности в алфавитном порядке
lower_bound	Обнаруживает первое значение в последовательности, которое не меньше заданного значения
make_heap	Выполняет пирамидальную сортировку последовательности (пирамида, на английском языке heap, — полное двоичное дерево, обладающее тем свойством, что значение каждого узла не меньше значения любого из его дочерних узлов. - *Примеч. пер.)*

Таблица 14.5 (продолжение)

Алгоритм	Назначение
max	Возвращает максимальное из двух значений
max_element	Возвращает итератор максимального элемента внутри диапазона
merge	Выполняет слияние двух упорядоченных последовательностей, а результат размещает в третьей последовательности
min	Возвращает минимальное из двух значений
min_element	Возвращает итератор минимального элемента внутри диапазона
mismatch	Обнаруживает первое несовпадение между элементами в двух последовательностях. Возвращает итераторы обоих несовпадающих элементов
next_permutation	Образует следующую перестановку (permutation) последовательности
nth_element	Упорядочивает последовательность таким образом, чтобы все элементы, меньшие заданного элемента *E,* располагались перед ним, а все элементы, большие заданного элемента E, — после него
partial_sort	Сортирует диапазон
partial_sort_copy	Сортирует диапазон, а затем копирует столько элементов, сколько войдет в результирующую последовательность
partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь
pop_heap	Меняет местами первый и предыдущий перед последним элементы, а затем восстанавливает пирамиду
prev_permutation	Образует предыдущую перестановку последовательности
push_heap	Размещает элемент на конце пирамиды
random_shuffle	Беспорядочно перемешивает последовательность
remove **remove_if** **remove_copy** **remove_copy_if**	Удаляет элементы из заданного диапазона
replace **replace_if** **replace_copy** **replace_copy_if**	Заменяет элементы внутри диапазона
reverse **reverse_copy**	Меняет порядок сортировки элементов диапазона на обратный

Таблица 14.5 (продолжение)

Алгоритм	Назначение
rotate **rotate_copy**	Выполняет циклический сдвиг влево элементов в диапазоне
search	Выполняет поиск подпоследовательности внутри последовательности
search_n	Выполняет поиск последовательности заданного числа одинаковых элементов
set_difference	Создает последовательность, которая содержит различающиеся участки двух упорядоченных наборов
set_intersection	Создает последовательность, которая содержит одинаковые участки двух упорядоченных наборов
set_symmetric_difference	Создает последовательность, которая содержит симметричные различающиеся участки двух упорядоченных наборов
set_union	Создает последовательность, которая содержит объединение (union) двух упорядоченных наборов
sort	Сортирует диапазон
sort_heap	Сортирует пирамиду внутри диапазона
stable_partition	Упорядочивает последовательность таким образом, чтобы все элементы, для которых предикат возвращает истину, располагались перед элементами, для которых предикат возвращает ложь. Разбиение на разделы остается постоянным; относительный порядок расположения элементов последовательности не меняется
stable_sort	Сортирует диапазон. Одинаковые элементы не переставляются
swap	Меняет местами два значения
swap_ranges	Меняет местами элементы в диапазоне
transform	Назначает функцию диапазону элементов и сохраняет результат в новой последовательности
unique **unique_copy**	Удаляет повторяющиеся элементы из диапазона
upper_bound	Обнаруживает последнее значение в последовательности, которое не больше некоторого значения

Примеры

1. Одними из самых простых алгоритмов являются алгоритмы **count()** и **count_if().** Ниже представлены их основные формы:

```
template<class InIter, class T>
    size_t count(InIter начало,
    Inlter окончание, const T &значение);
```

```
template<class InIter, class T>
    size_t count(InIter начало,
    Inlter окончание, UnPred &ф_предикат);
```

Алгоритм **count()** возвращает число элементов в последовательности, начиная с элемента, обозначенного итератором ***начало,*** и заканчивая элементом, обозначенным итератором ***окончание,*** значение которых равно параметру ***значение.*** Алгоритм **count_if()** возвращает число элементов в последовательности, начиная с элемента ***начало*** и заканчивая элементом ***окончание,*** для которых унарный предикат *ф_предикат* возвращает истину.

В следующей программе демонстрируются алгоритмы **count() и count_if()**.

```
// Демонстрация алгоритмов count и count_if
#include <iostream>
ttinclude <vector>
#include <algoritm>
using namespace std;

/* это унарный предикат, который определяет, является ли значение
четным
*/
bool even(int x)
{
    return !(x%2);
}
int main()
{
    vector<int> v;
    int i;

    for(i=0; i<20; i++) {
        if(i%2)v.push_back(1);
        else v.push_back(2);
    }

    cout << "Последовательность: ";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    int n;
    n = count(v.begin(), v.end(), 1);
    cout << n << " элементов равно 1\n";

    n = count_if(v.begin(), v.end(), even);
    cout << n << " четных элементов\n";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Последовательность:   2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 элементов равно 1
10 четных элементов
```

Программа начинается с создания 20-элементного вектора, содержащего чередующиеся значения 1 и 2. Для подсчета единиц используется алгоритм **count(),** а для подсчета четных элементов — алгоритм **count_if()**. Отметьте, как программируется унарный предикат **even()**. Все унарные предикаты получают в качестве параметра объект, тип которого тот же, что и тип объектов контейнера, для работы с которым предназначен предикат. В зависимости от значения этого объекта унарный предикат должен возвращать истину либо ложь.

2. Иногда полезно генерировать новую последовательность, состоящую только из определенных фрагментов исходной последовательности. Одним из предназначенных для этого алгоритмов является алгоритм **remove_copy()**, основная форма которого представлена ниже:

```
template<class InIter, class OutIter, class T>
    OutIter remove_copy (InIter начало,
    Inlter окончание, Outlter результат, const T &значение) ;
```

Алгоритм **remove_copy()** копирует элементы, равные параметру ***значение,*** из заданного итераторами ***начало*** **и** ***окончание*** диапазона и размещает результат в последовательности, обозначенный итератором ***результат.*** Алгоритм возвращает итератор конца новой последовательности. Результирующий контейнер должен быть достаточно велик для хранения новой последовательности.

В следующем примере показана работа алгоритма **remove_copy()**. Сначала в программе создается чередующаяся последовательность значений 1 и 2. Затем из последовательности удаляются все единицы.

```
// Демонстрация алгоритма remove_copy
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v, v2(20);
    int i;

    for(i=0; i<20; i++) {
        if(i%2) v.push_back(1);
        else v.push_back(2);
    }
```

```
    cout << "Последовательность: ";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    // удаление единиц
    remove_copy(v.begin(), v.end(), v2.begin(), 1);
    cout << "Результат: ";
    for(i=0; i<v2.size(); i++) cout << v2[i] << " ";
    cout << endl;

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Последовательность:  2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Результат: 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

3. Еще одним полезным алгоритмом является алгоритм **reverse(),** который меняет порядок расположения элементов последовательности на обратный. Ниже представлена основная форма этого алгоритма:

```
template<class BiIter>
        void reverse(BiIter начало, BiIter окончание);
```

Алгоритм **reverse()** меняет на обратный порядок расположения элементов в диапазоне, заданном итераторами ***начало*** **и** ***окончание.***

В следующем примере показана работа алгоритма **reverse().**

```
// Демонстрация алгоритма reverse
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v;
    int i;

    for(i=0; i<10; i++) v.push_back(i);

    cout << "Исходная последовательность: ";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;

    reverse(v.begin(), v.end());

    cout << "Обратная последовательность: ";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходная последовательность: 0 1 2 3 4 5 6 7 8 9
Обратная последовательность: 9 8 7 6 5 4 3 2 1 0
```

4. Одним из наиболее интересных алгоритмов является алгоритм **transform()**, который модифицирует каждый элемент некоторого диапазона в соответствии с заданной вами функцией. Алгоритм transform() имеет две основные формы:

```
template<class InIter, class OutIter, class Func>
        OutIter transform(Inlter начало, Inlter окончание,
                  Outlter результат, Func унарная_функция);

template<class InIter1, class InIter2, class Outlter, class Func>
    Outlter transform(InIter1 начало1, Inlterl окончание1,
      InIter2 начало2, Outlter результат, Func бинарная_функция);
```

Алгоритм **transform()** применяет функцию к диапазону элементов и сохраняет результат в месте, определенном итератором ***результат.*** В первой форме диапазон задается итераторами ***начало*** **и** ***окончание,*** **а** применяемой функцией является ***унарная_функция***. Эта функция в качестве параметра получает значение элемента и должна возвратить модифицированный элемент. Во второй форме модификация осуществляется с помощью бинарной оператор-функции ***бинарная_функция***, которая в качестве первого параметра получает значение элемента из предназначенной для модификации последовательности, а в качестве второго параметра — элемент из второй последовательности. Обе версии возвращают итератор конца итоговой последовательности.

В следующей программе для модификации используется функция **xform()**, которая возводит в квадрат элементы списка. Обратите внимание, что итоговая последовательность хранится в том же списке, что и исходная последовательность.

```
// Пример использования алгоритма transform
#include <iostream>
#include <list>
ttinclude <algorithm>
using namespace std;

// Простая функция модификации
int xform(int i) {
    return i * i; // квадрат исходного значения
}

int main()
{
    list<int> x1;
    int i;
```

```
    // размещение значений в списке
    for(i=0; i<10; i++) x1.push_back(i);

    cout << "Исходное содержимое списка x1: ";
    list<int>::iterator p = x1.begin();
    while(p != x1.end()) {
        cout << *p << " ";
        p++;
    }

    cout << endl;

    // модификация элементов списка xl
    p = transform(xl.begin(), x1.endO, x1.begin(), xform);

    cout << "Модифицированное содержимое списка x1: ";
    p = x1.begin();
    while(p != x1.end()) {
        cout << *p << " ";
        P++;
    }

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходное содержимое списка xl: 0 1 2 3 4 5 6 7 8 9
Модифицированное содержимое списка x1: 0 1 4 9 16 25 36 49 64 81
```

Как видите, возведен в квадрат каждый элемент списка **xl.**

Алгоритм **sort()** имеет следующие основные формы:

```
template<class RandIter>
          void sort(RandIter начало, RandIter окончание) ;
template<class RandIter, class Comp>
        void sort(RandIter начало, Randlter окончание,
                                   Comp функция_сравнения);
```

Алгоритм сортирует элементы диапазона, заданного итераторами ***начало*** **и** ***окончание.*** Во второй форме имеется возможность задать функцию сравнения, которая определяет, является ли один элемент меньше, чем другой. Напишите программу, демонстрирующую работу алгоритма **sort()**. (Выберите ту его форму, которая вам больше нравится.)

2. Алгоритм **merge()** выполняет слияние двух упорядоченных последовательностей, а результат слияния размещает в третьей последовательности. Ниже показана одна из основных форм этого алгоритма:

```
template<class InIter1, class InIter2, class OutIter>
    OutIter merge(InIter1 начало1, InIter1 окончание1,
       InIter2 начало2, InIter2 окончание2, OutIter результат);
```

Последовательности, предназначенные для слияния, обозначены итераторами ***начало1***, ***окончание1*** и ***начало2***, ***окончание2***. Место размещения итоговой последовательности обозначено итератором ***результат***. Алгоритм возвращает итератор конца итоговой последовательности. Продемонстрируйте работу этого алгоритма.

14.7. Строковый класс

Как известно, в C++ встроенный строковый тип данных сам по себе не поддерживается. Тем не менее для обработки строк здесь имеется две возможности. Во-первых, можно использовать хорошо вам знакомый оканчивающийся нулем символьный массив. О таком массиве иногда говорят как о *строке в стиле C (C string)*. Второй метод, который и рассматривается в этом разделе, подразумевает использование объектов типа **string.**

Фактически, класс **string** является конкретизацией более общего класса-шаблона **basic_string**. На самом деле у класса **basic_string** имеется два производных класса: класс **string,** который поддерживает строки 8-разрядных символов, и **wstring**, который поддерживает строки широких символов. Поскольку при обычном программировании чаще всего имеют дело именно с 8-разрядными символами, мы рассмотрим только версию string базового класса **basic_string**.

Перед тем как начать изучение класса **string,** важно понять, почему он включен в библиотеку классов C++. Стандартные классы появились в C++ не случайно. Фактически, включению в библиотеку каждого нового класса сопутствовало множество споров и дискуссий. Добавление в C++ класса **string** на первый взгляд кажется исключением из этого правила, поскольку в C++ в качестве строк уже поддерживаются оканчивающиеся нулем массивы. Тем не менее это далеко не так, и вот почему: оканчивающиеся нулем символьные массивы нельзя обрабатывать посредством стандартных операторов C++ и они не могут быть частью обычных выражений C++. Например, рассмотрим следующий фрагмент программы:

```
char s1[80], s2[80], s3[80];

s1 = "раз"; // не допускается
s2 = "два"; // не допускается
s3 = s1 + s2; // ошибка, снова не допускается
```

Как показано в комментариях, в C++ нельзя использовать оператор присваивания, чтобы дать символьному массиву новое значение (за исключением инициализации), а для конкатенации двух строк нельзя использовать оператор сложения. Эти операции приходится выполнять с помощью показанных ниже библиотечных функций:

```
strcpy(s1, "раз");
strcpy(s2, "два");
strcpy(s3, s1);
strcpy(s3, s2);
```

Поскольку оканчивающиеся нулем символьные массивы по своей сути технически не являются типами данных, к ним нельзя применять операторы C++. Это приводит к тому, что даже самые элементарные операции со строками становятся чрезвычайно запутанными. Невозможность использования стандартных операторов C++ для работы с оканчивающимися нулем символьными массивами и стала основной причиной разработки стандартного строкового класса. Вспомните, когда вы в C++ определяете класс, вы определяете новый тип данных, который может быть полностью интегрирован в среду программирования C++. Само собой это означает, что относительно нового класса можно перегружать операторы. Таким образом, благодаря добавлению в C++ стандартного класса **string,** становится возможным обрабатывать строки точно таким же образом, каким обрабатываются данные других типов, а именно с помощью операторов.

Имеется, однако, и еще один довод в пользу использования стандартного класса **string** — это обеспечение безопасности. Неопытный или неосторожный программист может очень легко выйти за границы массива, в котором хранится оканчивающаяся нулем строка. Например, рассмотрим стандартную функцию копирования строк **strcpy()**. В этой функции совершенно отсутствуют какие бы то ни было атрибуты, предназначенные для контроля границ целевого массива. Если в исходном массиве оказывается больше символов, чем может поместиться в целевом массиве, вполне возможна (и даже весьма вероятна) программная или даже системная ошибка. Как вы в дальнейшем увидите, стандартный класс **string** предотвращает саму возможность возникновения подобных ошибок.

Итак, для включения в C++ стандартного класса **string** имеется три довода: совместимость (теперь строка становится типом данных), удобство (можно использовать стандартные операторы C++) и безопасность (границы массива не нарушаются). Запомните, что это не доводы в пользу отказа от обычных, оканчивающихся нулем массивов. Они остаются наиболее эффективным способом реализации символьных строк. Тем не менее, если при создании приложения скорость выполнения программы не является доминирующим фактором, новый класс **string** предоставляет вам безопасный и полностью интегрированный в среду программирования C++ способ обработки строк.

Хотя строковый класс традиционно не считают частью библиотеки стандартных шаблонов, тем не менее, это один из определенных в C++ классов-контейнеров. Это, в частности, означает, что он поддерживает описанные в предыдущем разделе алгоритмы. Кроме этого, для обработки строк имеются дополнительные возможности. Чтобы получить доступ к классу string, в программу следует включить заголовок **<string>**.

Класс **string** очень велик, в нем имеется множество конструкторов и функций-членов. Помимо этого многие функции-члены имеют массу перегруженных форм. По этой причине в одной главе невозможно рассказать о всех членах класса **string.** Вместо этого мы исследуем только некоторые, основные его возможности. После получения базовых знаний о работе класса **string** в целом, все остальное вы сможете легко понять самостоятельно.

В классе **string** поддерживается несколько конструкторов. Ниже представлены прототипы трех из них, которые чаще всего используются:

```
string();
string(const char *строка);
string(const string &строка);
```

В первой форме создается пустой объект типа **string.** Во второй форме — объект типа **string** из оканчивающейся нулем строки, обозначенной указателем *строка*. Эта форма обеспечивает преобразование из оканчивающейся нулем строки в объект типа **string.** В третьей форме объект типа **string** создается из другого объекта типа **string.**

Ниже перечислена часть операторов, допустимых при работе с объектами типа **string:**

Оператор	Значение
=	Присваивание
+	Конкатенация
+=	Присваивание с конкатенацией
==	Равенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
[]	Индекс
<<	Вывод
>>	Ввод

Указанные операторы позволяют использовать объекты типа string в обычных выражениях и отказаться от вызовов специализированных функций, например, функций **strcpy()** или **strcat()**. Как правило, объекты типа string в выражениях можно записывать вместе с обычными, оканчивающимися нулем строками. Например, объект типа string можно присвоить оканчивающейся нулем строке.

Оператор + можно использовать для конкатенации объекта типа string с другим объектом типа string или для конкатенации объекта типа string со строкой в стиле С. Поддерживаются следующие варианты:

```
string + string
string + C-string
C-string + string
```

Кроме этого, оператор + можно использовать для присоединения одиночного символа к концу строки.

В классе string определена константа npos, обычно равная -1. Эта константа отражает максимально возможную длину строки.

Несмотря на то, что большая часть операций над строками может быть выполнена с помощью строковых операторов, для некоторых наиболее сложных или необычных операций нужны функции — члены класса string. Хотя их слишком много для одной главы, о некоторых наиболее полезных здесь будет рассказано. Чтобы присвоить одну строку другой используется функция **assign()**. Ниже представлены две основные формы этой функции:

```
string fiassign(const string &объект_строка,
                              size_type начало, size_type число);
string fiassign(const char *строка, size_type число);
```

В первой форме несколько символов, количество которых равно параметру *число* из объекта ***объект_строка***, начиная с индекса *начало,* присваиваются вызывающему объекту. Во второй форме вызывающему объекту присваивается первые несколько символов, количество которых равно параметру *число* из оканчивающейся нулем строки *строка*. В обоих случаях функция возвращает ссылку на вызывающий объект. Очевидно, что для присваивания одной целой строки другой гораздо проще использовать оператор =. Функция **assign()** может понадобиться только при присваивании части строки.

Присоединить часть одной строки к другой можно с помощью функции-члена **append()**. Ниже представлены две основные формы этой функции:

```
string &append(const string &объект_строка,
                              size_type начало, size_type число);
string &append(const char *строка, size_type число);
```

В первой форме несколько символов, количество которых равно параметру *число* из объекта ***объект_строка***, начиная с индекса *начало* присоединяются к вызывающему объекту. Во второй форме к вызывающему объекту присое-

диняются первые несколько символов, количество которых равно параметру ***число***, из оканчивающейся нулем строки *строка*. В обоих случаях функция возвращает ссылку на вызывающий объект. Очевидно, что для присоединения одной целой строки к другой гораздо проще использовать оператор +. Функция **append()** может понадобиться только при присоединении части строки.

С помощью функций **insert()** и **replace()** можно соответственно вставлять или заменять символы в строке. Ниже представлены прототипы основных форм этих функций:

```
string fiinsert(size_type начало, const string &объект_строка);

string fiinsert(size_type начало, const string &объект_строка,
                size_type начало_вставки, size_type число);

string fireplace(size_type начало, size_type число,
                                   const string &объект_строка);

string fireplace(size_type начало, size_type исх_номер,
         const string &объект_строка,
                 size_type начало_замены, size_type число_замены);
```

В первой форме функции **insert()** объект ***объект_строка*** вставляется в вызывающую строку по индексу ***начало***. Во второй форме функции **insert()** ***число*** символов из объекта ***объект_строка***, начиная с индекса ***начало_вставки***, вставляется в вызывающую строку по индексу ***начало***.

В первой форме функции **replace()** ***число*** символов, начиная с индекса ***начало***, заменяется в вызывающей строке объектом ***объект_строка***. Во второй форме функции **replace()** в вызывающей строке заменяется ***исх_число*** символов, начиная с индекса ***начало***, при этом из объекта ***объект_строка*** берутся ***число_замены*** символов, начиная с индекса ***начало_замены***. В обоих случаях функция возвращает ссылку на вызывающий объект.

Удалить символы из строки можно с помощью функции erase(). Ниже показана одна из форм этой функции:

```
string fierase(size_type начало = 0, size_type число = npos);
```

Функция удаляет *число* символов из вызывающей строки, начиная с индекса *начало*. Возвращаемым значением является ссылка на вызывающий объект.

В классе string поддерживается несколько функций-членов, предназначенных для поиска строк. Среди них имеются функции find() и **rfind()**. Ниже показаны прототипы основных версий этих функций:

```
size_type find(const string &объект_строка,
                                  size_type начало = 0) const;
size_type rfind(const string &объект_строка,
                                  size_type начало = npos) const;
```

Начиная с индекса ***начало*** функция **find()** ищет в вызывающей строке первое совпадение со строкой, содержащейся в объекте ***объект_строка***. Если искомая строка найдена, функция **find()** возвращает индекс вызывающей строки, соответствующий найденному совпадению. Если искомая строка не найдена, функция **find()** возвращает значение **npos**. **В** противоположность функции **find()**, функция **rfind()**, начиная с индекса ***начало,*** но в обратном направлении, ищет в вызывающей строке первое совпадение со строкой, содержащейся в объекте ***объект_строка***. (То есть ищет последнее совпадение со строкой, содержащейся в объекте ***объект_строка***.) Если искомая строка найдена, функция **rfind()** возвращает индекс вызывающей строки, соответствующий найденному совпадению. Если искомая строка не найдена, функция **rfind()** возвращает значение **npos.**

Для сравнения целых строковых объектов удобнее всего пользоваться описанными ранее перегруженными операторами отношения. Тем не менее, если вы захотите сравнить части строк, вам понадобится функция-член **compare().** Ниже представлен прототип этой функции:

```
int compare(size_type начало, size_type число,
                              const string &объект_строка) const;
```

Здесь с вызывающей строкой сравниваются ***число*** символов объекта ***объект_строка***, начиная с индекса ***начало.*** Если вызывающая строка меньше, чем ***объект_строка,*** функция **compare()** возвращает отрицательное значение. Если вызывающая строка больше, чем ***объект_строка***, функция **compare()** возвращает положительное значение. Если вызывающая строка равна объекту ***объект_строка***, возвращаемое значение функции **compare()** равно нулю.

Хотя объекты типа **string** сами по себе очень удобны, иногда у вас все же будет возникать необходимость в версии строки в виде массива символов, оканчивающихся нулем. Например, объект типа **string** можно использовать для образования имени файла. Однако при открытии файла вам придется задавать указатель на стандартную, оканчивающуюся нулем строку. Для решения проблемы в классе **string** имеется функция-член **c_str(),** прототип которой показан ниже:

```
const char *c_str() const;
```

Функция возвращает указатель на оканчивающуюся нулем версию строки, содержащуюся в вызывающем объекте типа **string.** Оканчивающаяся нулем строка не должна меняться. Кроме этого, если над объектом типа **string** выполнялись какие-либо другие операции, правильность выполнения функции **c_str()** не гарантируется.

Поскольку класс **string** является контейнером, в нем поддерживаются функции begin() и **end(),** возвращающие соответственно итератор начала и конца строки. Также поддерживается функция **size()**, возвращающая текущее число символов строки.

Примеры

1. Хотя мы уже привыкли к традиционным строкам в стиле C, в C++ класс **string** делает обработку строк существенно проще. Например, при работе с объектами типа **string** для присваивания строк можно использовать оператор =, для конкатенации строк — оператор +, а для сравнения строк — различные операторы сравнения. В следующей программе показаны эти операции.

```
// Короткий пример использования строкового класса
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1("Представление строк");
    string str2("Вторая строка");
    string str3;

    // присваивание строк
    str3 = str1;
    cout << str1 << "\n" << str3 << "\n";

    // конкатенация двух строк
    str3 = str1 + str2;
    cout << str3 << "\n";

    // сравнение строк
    if(str3 > str1) cout << "str3 > str1\n";
    if(str3 == str1+str2) cout << "str3 == str1+str2\n";

    // строковому объекту можно присвоить обычную строку
    str1 = "Это обычная строка\n";
    cout << str1;

    // создание строкового объекта
    // с помощью другого строкового объекта
    string str4(str1);
    cout << str4;

    // ввод строки
    cout << "Введите строку: ";
    cin >> str4;
    cout << str4;

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Представление строк
Представление строк
```

```
Представление строкВторая строка
str3 > strl
str3 == strl+str2
Это обычная строка
Это обычная строка
Введите строку: Привет
Привет
```

Как видите, с объектами типа **string** можно обращаться так же, как и со встроенными типами данных C++. Это, фактически, и есть главное достоинство строкового класса.

Отметьте простоту манипулирования со строками: для конкатенации строк используется обычный оператор +, а для их сравнения — обычный оператор >. Чтобы выполнить те же операции для оканчивающихся нулем строк в стиле C, вам пришлось бы вызывать функции **strcat()** и **strcmp()**, что, согласитесь, гораздо менее удобно. Поскольку объекты типа **string** можно совершенно свободно указывать в выражениях вместе с оканчивающимися нулем строками в стиле C, то никаких неприятностей от их использования в ваших программах быть не может, а выгоды, наоборот, — очевидны.

Имеется еще одна деталь, на которую следует обратить внимание в предыдущей программе: размеры строк не задаются. Объекты типа **string** автоматически настраиваются на хранение строк требуемой длины. Таким образом, когда вы выполняете присваивание или конкатенацию строк, размер целевой строки автоматически вырастает ровно настолько, насколько это нужно для размещения новой строки. Этот динамический аспект использования объектов типа **string** следует всегда принимать во внимание при выборе варианта представления строк в ваших программах. (Как уже отмечалось, стандартные оканчивающиеся нулем строки *являются* возможным источником нарушения границ массивов).

2. В следующей программе демонстрируются функции **insert()**, **erase()** и **replace()**.

```
// Использование функций insert(), erase() и replace()
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1("Это проверка");
    string str2("АБВГДЕЖ");

    cout << "Исходные строки:\n"
    cout << "strl: " << strl << endl;
    cout << "str2: " << str2 << "\n\n";

    // работа функции insert()
    cout << "Вставка строки str2 в строку str1:\n"
```

```
    str1.insert(4,str2);
    cout << str1 << "\n\n";

    // работа функции erase()
    cout << "Удаление семи символов из строки str1:\n"
    str1.erase(4, 7);
    cout << str1 << "\n\n";

    // работа функции replace()
    cout << "Замена восьми символов из str1 символами из str2:\n"
    str1.replace(4, 8, str2);
    cout << str1 << "\n\n";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Исходные строки:
str1: Это проверка
Str2: АБВГДЕЖ

Вставка строки str2 в строку str1:
Это АБВГДЕЖпроверка

Удаление семи символов из строки str1:
Это проверка

Замена восьми символов из str1 символами из str2:
Это АБВГДЕЖ
```

3. Поскольку класс **string** определяет тип данных, появляется возможность создавать контейнеры для хранения объектов типа **string.** Например, ниже представлена усовершенствованная версия программы создания ассоциативного списка для хранения слов и антонимов, впервые показанная в примере 3 раздела 14.5.

```
/* Ассоциативный список слов и антонимов для объектов типа string
*/
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
    map<string, string> m;
    int i;

    m.insert(pair<string, string>("да", "нет"));
    m.insert(pair<string,string>("хорошо", "плохо"));
    m.insert(pair<string, string>("влево"), "вправо"));
    m.insert(pair<string, string>("вверх"), "вниз"));
```

```
    string s;
    cout << "Введите слово: ";
    cin>>s;

    map<string, string>::iterator p;

    p = m.find(s);
    if (p != m.end())
        cout << "Антоним: " << p->second;
    else
        cout << "Такого слова в ассоциативном списке нет\n";

    return 0;
}
```

Упражнения

С помощью объектов типа **string** сохраните в списке следующие строки:

один	**два**	**три**	**четыре**	**пять**
шесть	**семь**	**восемь**	**девять**	**десять**

Затем отсортируйте список и выведите на экран содержимое отсортированного списка.

Поскольку класс **string** является контейнером, он может использоваться со стандартными алгоритмами. Создайте программу, в которой пользователь вводит строку. Затем с помощью функции **count()** сосчитайте в строке число символов "e" и выведите это значение на экран.

Модифицируйте решение упражнения 2 таким образом, чтобы подсчитывались только символы в нижнем регистре. (Подсказка: воспользуйтесь функцией **count_if()**.)

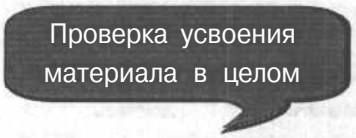
Класс **string** — это конкретизация некоторого класса-шаблона. Какого?

Проверка усвоения материала главы

Теперь вам необходимо выполнить следующие упражнения и ответить на вопросы.

1. Каким образом библиотека стандартных шаблонов позволяет упростить процесс создания более надежных программ?
2. Опишите контейнер, итератор и алгоритм в терминах библиотеки стандартных шаблонов.

3. Напишите программу создания 10-элементного вектора, содержащего числа от 1 до 10. Затем из полученного вектора скопируйте в список только четные элементы.
4. -В чем преимущество использования данных типа **string? В** чем их единственный недостаток?
5. Что такое предикат?
6. Переработайте пример 2 раздела 14.5 так, чтобы в нем использовались объекты типа **string.**
7. Начните изучение объектов-функций библиотеки стандартных шаблонов. Для начала познакомьтесь со стандартными классами **unary_function и binary_function,** которые помогут вам создавать объекты-функции.
8. Изучите техническую документацию на библиотеку стандартных шаблонов, поставляемую с вашим компилятором. Там вы обязательно обнаружите массу полезных инструментов и приемов программирования.



В этом разделе проверяется, хорошо ли вы усвоили материал этой и предыдущих глав.

1. Начиная с главы 1, вы проделали долгий путь. Потратьте еще немного времени для беглого просмотра книги. После этого подумайте, как можно было бы улучшить примеры (особенно в первых шести главах) с помощью всех известных вам теперь инструментов C++.
2. Программирование легче изучать в процессе работы. Пишите больше программ на C++. Пытайтесь экспериментировать с теми атрибутами языка, которые уникальны только для него.
3. Продолжайте изучение библиотеки стандартных шаблонов. В будущем большинство задач программирования будет решаться именно с помощью библиотеки стандартных шаблонов, поскольку, как правило, кажущаяся сложность работы с контейнерами может быть значительно снижена, благодаря алгоритмам.
4. И последнее. Запомните: C++ дает вам беспрецедентные возможности и важно научиться мудро ими пользоваться. Благодаря этим возможностям C++ позволит вам значительно расширить потенциальные области применения ваших приложений. Однако злоупотребление ими может привести к появлению программ, которые трудно понимать, которым почти невозможно следовать и которые слишком тяжело сопровождать. Язык программирования C++ — это мощнейший инструмент, но, как и любой другой инструмент, он хорош только тогда, когда находится в надежных руках.

Приложение А

Некоторые отличия языков программирования С и С++

Для подавляющего большинства задач программирования С++ можно считать надмножеством языка С (как он описан в стандарте ANSI по С), поэтому фактически все программы С являются также программами С++. Имеется несколько отличий, о части из которых было рассказано в главе 1. Ниже перечислены оставшиеся отличия, о которых вам следует знать:

- Незначительное, но потенциально важное отличие между С и С++ состоит в том, что в С символьная константа автоматически преобразуется в целое, а в С++ нет.
- В С можно объявить глобальную переменную несколько раз, что, хотя и считается плохим стилем программирования, но ошибкой не является. В С++ многократное объявление глобальной переменной считается ошибкой.
- В С по крайней мере первые 31 символ идентификатора являются значащими. В С++ значащими являются все символы идентификатора. Тем не менее с точки зрения практики столь длинные идентификаторы вряд ли могут где-нибудь пригодиться.
- В С вы можете вызвать функцию **main()** в любом месте программы, хотя какую-либо пользу от такого вызова представить себе довольно трудно. В С++ вызов функции **main()** из программы не допускается.
- В С нельзя получить адрес регистровой переменной (переменной типа **register),** а в С++ — можно.
- **В С тип wchar_t** определяется с помощью объявления **typedef. В С++ wchar_t** является ключевым словом.

Приложение В

Ответы на вопросы и решения упражнений

В этом приложении вы найдете ответы на вопросы и решения упражнений для подавляющего большинства (но не всех) приведенных в книге заданий. Часть вопросов и упражнений оставлены вам для самостоятельного творчества. Советуем вам тщательно и добросовестно выполнять все задания, независимо от того, есть в этом приложении ответы на них или нет.

ГЛАВА 1

Упражнения

1.3

1.
```
#include <iostream>
using namespace;

int main()
{
    double hours, wage;

    cout << "Введите количество проработанных часов: " ;
    cin >> hours;

    cout << "Введите почасовую оплату: ";
    cin >> wage;

    cout << "Зарплата равна: $" << wage * hours;

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;
```

```
int main()
{
    double feet;

    do {
        cout << "Введите число футов (0 для завершения): ";
        cin >> feet;

        cout << feet * 12 << " дюймов\n";
    } while (feet != 0.0);

    return 0;
}
```

3.
```
/* В этой программе рассчитывается наименьшее общее кратное
*/
ttinclude<iostream>
using namespace std;

int main()
{
    {
        int a, b, d, min;

        cout << "Введите два числа: ";
        cin>>a>>b;

        min = a>b ?b: a;

        for (d=2; d<min; d++)
            if(((a%d)==0) && ((b%d)==0)) break;
        if (d==min) {
            cout << "Нет общего кратного\n";
            return 0;
    },
    cout << "Наименьшее общее кратное равно " << d << "\n";

    return 0;
}
```

1.4

1. Этот комментарий, хотя и выглядит довольно странно, вполне допустим.

1.5

2.
```
#include <iostream>
#include <cstring>
using namespace std;
```

```
class card {
    char title[80]; // заглавие книги
    char author[40]; // автор
    int number; // количество имеющихся экземпляров
public:
    void store(char *t, char *name, int num);
    void show();
};

void card::store(char *t, char *name, int num)
{
    strcpy(title, t);
    strcpy(author, name);
    number = num;
}

void card::show()
{
    cout << "Заглавие: " << title << "\n";
    cout << "Автор: " << author << "\n";
    cout << "Количество экземпляров: " << number << "\n";
}

int main()
{
    card book1, book2, book3;

    book1.store("Dune", "Frank Herbert", 2);
    book2.store("The Foundation Trilogy", "Isaac Asimov", 2);
    book3.store("The Rainbow", "D. H. Lawrence", 1);

    book1.show();
    book2.show();
    book3.show();

    return 0;
}
```

3.
```
ttinclude <iostream>
using namespace std;

#define SIZE 100

class q_type {
    int queue[SIZE]; // содержит очередь
    int head, tail; // индекс вершины и хвоста
public:
    void init(); // инициализация
    void q(int num); // запоминание
```

```
    int deq(); // восстановление
};

// Инициализация
void q_type::init()
{
    head = tail = 0;
}

// Помещение значения в очередь
void q_type::q(int num)
{
    if (tail + 1==head || (tail + 1==SIZE && !head)) {
        cout << "Очередь полна";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // круговой цикл
    queue[tail] = num;
}

// Удаление значения из очереди
int q_type::deq()
{
    if (head==tail) {
        cout << "Очередь пуста";
        return 0;
    }
    head++;
    if(head==SIZE) head = 0; // круговой цикл
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    q1.init();
    q2.init();

    for(i=1; i<=10; i++) {
        q1.q(i);
        q2.q(i * i);
    }

    for(i=1; i<=10; i++) {
        cout << "Элемент из очереди 1: " << q1.deq() << "\n";
        cout << "Элемент из очереди 2: " << q2.deq() << "\n";
    }
```

```
    return 0;
}
```

1.6

1. У функции f() нет прототипа.

1.7

1.
```
#include <iostream>
#include <cmath>
using namespace std;

// Перегрузка функции sroot() для integers, longs и doubles
int sroot(int i);
long sroot(long i) ;
double sroot(double i);

int main()
{
    cout << "Квадратный корень 90.34 равен: " << sroot(90.34);
    cout << "\n";
    cout << "Квадратный корень 90L равен: " << sroot(90L);
    cout << "\n";
    cout << "Квадратный корень 90 равен: " << sroot (90);

    return 0;
}

// Возвращает квадратный корень целого
int sroot(int i)
{
    cout << "расчет корня целого\n";
    return (int) sqrt((double)i);
}

// Возвращает квадратный корень длинного целого
long sroot(long i)
{
    cout << "расчет корня длинного целого\n";
    return (long) sqrt((double)i);
}

// Возвращает квадратный корень вещественного
double sroot(double i)
{
    cout << "расчет корня вещественного\n";
    return sqrt(i);
}
```

2. Функций **atof(), atoi() и atol()** нельзя перегружать потому, что они отличаются только типом возвращаемого значения. Перегрузка функции требует, чтобы было отличие либо в типе, либо в числе аргументов.

3.
```
// Перегрузка функции min ( )
#include <iostream>
#include <cctype>
using namespace std;

char min (char a, char b) ;
int min(int a, int b) ;
double min (double a, double b) ;

int main ( )
{
    cout << "Минимум равен: " << min('x', 'a') << "\n";
    cout << "Минимум равен: " << min (10, 20) << "\n";
    cout << "Минимум равен: " << min(0.2234, 99.2) << "\n";

    return 0;
}

// Минимум для chars
char min(char a, char b)
{
    return tolower(a) < tolower(b) ? a: b;
}

// Минимум для ints
int min (int a, int b)
{
    return a < b ? a: b;
}

// Минимум для doubles
double min (double a, double b)
{
    return a < b ? a: b;
}
```

4.
```
tinclude <iostream>
using namespace std;

// Перегрузка функции sleep() для вызова с целым либо со строкой
void sleep(int n) ;
void sleep (char *n) ;

// Измените эту величину
// в соответствии с быстродействием вашего процессора
tfdefine DELAY 100000
```

```
int main()
{
    cout << '.';
    sleep(3);
    cout << '.';
    sleep("2");
    cout << '.';

    return 0;
}

// Функция sleep() с целым аргументом
void sleep(int n)
{
    long i;

    for(; n; n--)
        for(i=0; i<DELAY; i++);
}

// Функция sleep() с аргументом типа char *
void sleep(char *n)
{
    long i;
    int j;

    j = atoi(n);

    for(; j; j--)
        for(i=0; i<DELAY; i++);
}
```

Проверка усвоения материала главы 1

1. Полиморфизм — это механизм, посредством которого можно использовать один общий интерфейс для доступа к разным реализациям задачи. Инкапсуляция обеспечивает защищенную связь инструкций и данных, с которыми работает программа. Доступ к таким скрытым частям программы может быть затруднен и этим предотвращается несанкционированный доступ к ним. Наследование — это процесс, посредством которого один объект может приобрести свойства другого. Наследование используется для поддержки иерархии классов.
2. Комментарии могут включаться в программу C++ либо как обычные комментарии в стиле C, либо как однострочные комментарии, характерные для C++.
3.
```
#include <iostream>
using namespace std;
```

```
int main ()
{
    int b, e, r;
    cout << "Введите основание степени: ";
    cin >>b;
    cout << "Введите показатель степени: ";
    cin>>e;
    r = 1;
    for(; e; e --) r = r * b;
    cout << "Итог: " << r;
    return 0;
}
```

4.
```
linclude <iostream>
#include <cstring>
using namespace std;

// Перегрузка функции реверса строки
void rev_str(char *s); // реверс строки по адресу s
void rev_str(char *in, char *out); // реверс строки и пересылка ее
                                   // по адресу out

int main()
{
    char s1[80], s2[80];

    strcpy(sl, "Это проверка");

    rev_str(sl, s2) ;
    cout << s2 << "\n";

    rev_str (s1);
    cout << s1 << "\n";

    return 0;
}
// Реверс строки и передача результата по адресу s
void rev_str(char *s)
{
    char temp[80];
    int i, j;

    for(i=strlen(s)-1, j=0; i>=0; i--, j++)
    temp [ j ] = s [ i ];

    temp[ j ] = '\0'; // нуль завершает строку
    strcpy(s, temp);
}
```

```
// Реверс строки и передача результата по адресу out
void rev_str(char *in, char *out)
{
    int i, j;

    for(i=strlen(in)-1, j=0; i>=0; i--, j++)
    out[ j ] = in[ i ];

    out[ j ] = '\0'; // нуль завершает строку
}
```

5.
```
#include <iostream.h>

int f(int a);

int main()
{
    cout << f(10);

    return 0;
}

int f(int a)
{
    return a * 3.1416;
}
```

6. Тип данных **bool** предназначен для хранения значений булева типа. Значениями булева типа являются только два значения — это **true и false.**

ГЛАВА 2

Повторение пройденного •

1.
```
linclude <iostream>
#include <cstring>
using namespace std;

int main()
{
    char s[80];

    cout << "Введите строку: ";
    cin >> s;

    cout << "Длина строки равна: " << strlen(s) << "\n";

    return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class addr {
    char name[40];
    char street[40];
    char city[30];
    char state[3];
    char zip[10];
public:
    void store(char *n, char *s, char *c, char *t, char *z);
    voiddisplay();
};

void addr::store(char *n, char *s, char *c, char *t, char *z)
{
    strcpy(name,n);
    strcpy(street, s);
    strcpy(city,c);
    strcpy(state,t);
    strcpy(zip,z);
}

void addr::display()
{
    cout << name << "\n";
    cout << street << "\n";
    cout << city << "\n";
    cout << state << "\n";
    cout << zip << "\n\n";
}

int main()
{
    addr a;

    a.store("И. И. Иванов", "Невский проспект", "С.-Петербург",
                                                "Рос","46576");

    a.display();

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

int rotate(int i);
long rotate(long i);
```

```
int main()
{
    int a;
    long b;

    a = 0x8000;
    b = 8;

    cout << rotate(a);
    cout << "\n";
    cout << rotate(b);

    return 0;
}

int rotate(int i)
{
    int x;

    if(i & 0x8000) x = 1;
    else x = 0;

    i = i << 1;
    i += x;

    return i;
}

long rotate(long i)
{
    int x;

    if(i & 0x80000000) x = 1;
    else x = 0;

    i = i << 1;
    i += x;

    return i;
}
```

4. Целое i является закрытым для класса **myclass,** и к нему нет доступа из функции **main()**.

Упражнения

2.1

1.
```
#include <iostream>
using namespace std;
```

```
#define SIZE 100

class q_type {
    int queue[SIZE]; // содержит очередь
    int head, tail; // индексы вершины и хвоста
public:
    q_type(); // конструктор
    void q(int num) ; // запоминание
    int deq(); // извлечение из памяти
};

// Конструктор
q_type::q_type()
{
    head = tail = 0;
}

// Постановка значения в очередь
void q_type::q(int num)
{
    if (tail + 1==head || (tail + 1==SIZE && !head)) {
        cout << "Очередь полна";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // круговой цикл
    queue[tail] = num;
}

// Выталкивание значения из очереди
int q_type::deq()
{
    if (head==tail) {
        cout << "Очередь пуста";
        return 0;
    }
    head++;
    if(head==SIZE) head = 0; // круговой цикл
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    for(i=1; i<=10; i++) {
        q1.q(i);
        q2.q(i * i);
    }
```

```
    for(i=1; i<=10; i++) {
        cout << "Элемент из очереди 1: " << q1.deq() << "\n";
        cout << "Элемент из очереди 2: " << q2.deq() << "\n";
    }
    return 0;
}
```

2.
```
// Имитация секундомера
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
    double begin, end;
public:
    stopwatch();
    ~stopwatch();
    void start();
    void stop();
    void show();
};

stopwatch::stopwatch()
{
    begin = end = 0.0;
}

stopwatch::~stopwatch()
{
    cout << "Удаление объекта stopwatch ...";
    show();
}

void stopwatch::start()
{
    begin = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::stop()
{
    end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
    cout << "Затраченное время: " << end - begin;
    cout << "\n";
}
```

```
int main()
{
    stopwatch watch;
    long i;

    watch.start();
    for(i=0; i<320000; i++); // время цикла
    watch.stop();

    watch.show();

    return 0;
}
```

3. У конструктора не может быть возвращаемого значения.

2.2

1.
```
// Динамическое выделение памяти для стека
#include <iostream>
#include <cstdlib>
using namespace std;

// Объявление класса stack для символов
class stack {
    char *stck; // содержит стек
    int tos; // индекс вершины стека
    int size; // размер стека
public:
    stack(int s); // конструктор
    ~stack(); // деструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
};

// Инициализация стека
stack::stack(int s)
{
    cout << "Работа конструктора стека\n";
    tos = 0;
    stck = (char *) malloc(s);
    if(!stck) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    size = s;
}
```

```
stack::~stack()
{
    free(stck);
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==size) {
        cout << "Стек полон \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // образование двух, автоматически инициализируемых, стеков
    stack s1(10), s2(10);
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "символ из стека s1:" << s1.pop()
<< "\n";
    for(i=0; i<3; i++) cout << "символ из стека s2:" << s2.pop()
<< "\n";

    return 0;
}
```

2.
```
#include <iostream>
#include <ctime>
using namespace std;
```

```
class t_and_d {
    time_t systime;
public:
    t_and_d(time_t t); // конструктор
    void show();
};

t_and_d::t_and_d(time_t t)
{
    systime = t;
}

void t_and_d::show()
{
    cout << ctime(&systime);
}

int main()
{
    time_t x;

    x = time (NULL);

    t_and_d ob(x);

    ob.show();

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class box {
    double 1, w, h;
    double volume;
public:
    box (double a, double b, double c) ;
    voidvol ( ) ;
};

box: : box (double a, double b, double c)
{
    1 = a;
    w = b;
    h = c;

    volume = 1 * w * h;
}
```

```
void box::vol()
{
    cout << "Объем равен: " << volume << "\n";
}

int main()
{
    box x(2.2, 3.97, 8.09), y(1.0, 2.0, 3.0);

    x.vol();
    y.vol();

    return 0;
}
```

2.3

1.
```
ttinclude <iostream>
using namespace std;

class area_cl {
public:
    double height;
    double width;
};

class rectangle: public area_cl {
public:
    rectangle(double h, double w);
    double area();
};

class isosceles: public area_cl {
public:
    isosceles(double h, double w);
    double area();
};

rectangle:: rectangle(double h, double w)
{
    height = h;
    width = w;
}

isosceles: : isosceles(double h, double w)
{
    height = h;
    width = w;
}
```

```
double rectangle::area()
{
    return width * height;
}

double isosceles::area()
{
    return 0.5 * width * height;
}

int main()
{
    rectangle b(10.0, 5.0);
    isosceles i(4.0, 6.0);

    cout << "Прямоугольник: " << b.area() << "\n";
    cout << "Треугольник: " << i.area() << "\n";

    return 0;
}
```

2.5

```
1. // Класс стек, образуемый с помощью структуры
#include <iostream>
using namespace std;

ttdefine SIZE 10

// Объявление класса stack для символов
struct stack {
    stack(); // конструктор
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
private:
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека \n";
    tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
```

```
{
    if (tos==SIZE) {
        cout << "Стек полон \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos=0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

int main()
{
    // образование двух, автоматически инициализируемых, стеков
    stack s1, s2;
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "символ из стека s1:" << s1.pop() <<
"\n";
    for(i=0; i<3; i++) cout << "символ из стека s2:" << s2.pop() <<
"\n";

    return 0;
}
```

2.
```
ttinclude <iostream>
using namespace std;

union swapbytes {
    unsigned char c[2];
    unsigned i;
    swapbytes(unsigned x);
```

```
    void swp();
};

swapbytes:: swapbytes (unsigned x)
{
    i = x;
}

void swapbytes::swp()
{
    unsigned char temp;

    temp = c[0];
    c[0] = c[1];
    c[1] = temp;
}

int main()
{
    swapbytes ob(1);

    ob.swp();
    cout << ob.i;

    return 0;
}
```

3. Анонимное объединение представляет собой особый синтаксический механизм, который позволяет двум переменным совместно использовать одну и ту же область памяти. Доступ к членам анонимного объединения можно реализовать непосредственно, без ссылки на объект. Члены анонимного объединения находятся в той же области видимости, что и само объединение.

2.6

1.
```
#include <iostream>
using namespace std;

// Перегрузка функции abs() тремя способами

// Функция abs() для целых
inline int abs(int n)
{
    cout << "В функции abs() для int\n";
    return n<0 ? -n: n;
}

// Функция abs() для длинных целых
inline long abs(long n)
```

```
{
    cout << "В функции abs ( ) для long\n";
    return n<0 ? -n: n;
}

// Функция abs () для вещественных двойной точности
inline double abs (double n)
{
    cout << "В функции abs() для double\n";
    return n<0 ? -n: n;
}

int main ( )
{
    cout << "Абсолютная величина -10:" << abs (-10) << "\n";
    cout << "Абсолютная величина -10L:" << abs(-10L) << "\n";
    cout << "Абсолютная величина -10.01:" << abs(-10.01) << "\n";

    return 0;
}
```

2. Функция не может быть встраиваемой, поскольку содержит цикл for. Большинство компиляторов не поддерживает встраиваемые функции с циклами.

2.7

```
1. ttinclude <iostream>
using namespace std;

#define SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack() { tos = 0; }
    void push(char ch)
    {
        if (tos==SIZE) {
            cout << "Стек полон";
            return;
        }
    stck[tos]=ch;
    tos++;
    }
```

```
    char pop()
    {
        if (tos==0) {
            cout << "Стек пуст";
            return 0; // возврат нуля при пустом стеке
        }
        tos--;
        return stck[tos];
    }
};

int main()
{
    // образование двух, автоматически инициализируемых, стеков
    stack s1, s2;
    int i;

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "символ из стека s1:" << s1.pop() <<
"\n";
    for(i=0; i<3; i++) cout << "символ из стека s2:" << s2.pop() <<
"\n";

    return 0;
}
```

2.
```
#include <iostream>
ttinclude <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr)
    {
        len = strlen(ptr);
        p=(char *) malloc(len + 1);
        if(!p) {
            cout << "Ошибка выделения памяти\n";
            exit(1);
        }
```

```
        strcpy(p, ptr);
    }
    ~strtype() { cout << "Освобождение p\n"; free(p); }

    void show()
    {
        cout << p << " — длина: " << len;
        cout << "\n";
    }
};

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

Проверка усвоения материала главы 2

1. Конструктор — это функция, которая вызывается при создании объекта. Деструктор — это функция, которая вызывается при удалении объекта.

2.
```
#include <iostream>
using namespace std;

class line {
    int len;
public:
    line(int l);
};

line::line(int l)
{
    len = l;

    int i;
    for(i=0; i<len; i++) cout << '*';
}

int main()
{
    line l(10);

    return 0;
}
```

3. 10 1000000 -0.0009

4.
```
#include <iostream>
using namespace std;

class area_cl {
public:
    double height;
    double width;
};

class rectangle: public area_cl {
public:
    rectangle(double h, double w) { height = h; width = w; }
    double area() { return height * width; }
};

class isosceles: public area_cl {
public:
    isosceles(double h, double w) { height = h; width = w; )
    double area() { return 0.5 * height * width; }
};

class cylinder: public area_cl {
public:
    cylinder(double h, double w) { height = h; width = w; }
    double area()
    {
        return (2 * 3.1416 * (width/2) * (width/2)) + (3.1416 *
height * width);
    }
};

int main()
{
    rectangle b(10.0, 5.0);
    isosceles i(4.0, 6.0);
    cylinder c(3.0, 4.0);

    cout << "Прямоугольник: " << b.area() << "\n";
    cout << "Треугольник: " << i.area() << "\n";
    cout << "Цилиндр: " << c.area() << "\n";

    return 0;
}
```

5. **Тело встраиваемой функции встраивается в программу. Это означает, что реально функция не вызывается, что позволяет избежать потерь производительности, связанных с вызовом функции и возвращением функцией своего**

значения. Преимуществом встраиваемых функций является увеличение скорости выполнения программы, а их недостатком — увеличение ее объема.

6.
```
#include <iostream>
using namespace std;

class myclass {
    int i, j;
public:
    myclass(int x, int y) { i = x; j = y; }
    void show() { cout << i << " " << j ; }
};

int main()
{
    myclass count(2, 3);

    count.show();

    return 0;
}
```

7. Члены класса по умолчанию являются закрытыми. Члены структуры — открытыми.

8. Да. Это определение анонимного объединения.

Проверка усвоения материала в целом

1.
```
#include <iostream>
using namespace std;

class prompt {
    int count;
public:
    prompt(char *s) { cout << s; cin >> count; }
    ~prompt();
};

prompt::~prompt() {
    int i, j;
    for(i=0; i<count; i++) {
       cout << '\a';
       for(j=0; j<32000; j++) ; // пауза
    }
}
```

```
int main()
{
    prompt ob("Введите число: ");

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class ftoi {
    double feet;
    double inches;
public:
    ftoi(double f);
};

ftoi::ftoi(double f)
{
    feet = f;
    inches = feet * 12;
    cout << feet << "футов равно " << inches << "дюймам.\n";
}

int main()
{
    ftoi a(12.0), b(99.0);

    return 0;
}
```

3.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dice {
    int val;
public:
    void roll();
};

void dice::roll()
{
    val = (rand() % 6) +1; // генерация чисел от 1 до 6
    cout << val << "\n";
}

int main()
{
    dice one, two;
```

```
    one.roll();
    two.roll();
    one.roll();
    two.roll();
    one.roll();
    two.roll();

    return 0;
}
```

ГЛАВА 3

Повторение пройденного

1. Конструктор называется **widgit()**, а деструктор — **~widgit()**.
2. Конструктор вызывается при создании объекта (т. е., когда объект начинает существовать). Деструктор вызывается при удалении объекта.

```
3. class Mars: public planet {
       // ...
   };
```

4. Функцию можно сделать встраиваемой, если перед ее определением поставить спецификатор **inline,** или если ее определение разместить внутри объявления класса.
5. Встраиваемая функция должна быть определена перед ее первым использованием. В ней не должно быть циклов. Она не должна быть рекурсивной. В ней не может быть инструкций **goto и switch.** И наконец, она не должна содержать статических переменных.
6. sample `ob(100, 'X');`

Упражнения

3.1

1. Инструкция присваивания x = y неправильна, поскольку **cl1** и **cl2** — это два разных класса, а объекты разных типов присваивать нельзя.

2. ttinclude <iostream>
   using namespace std;

   ttdefine SIZE 100

```
class q_type {
    int queue[SIZE]; // содержит очередь
    int head, tail; // индексы вершины и хвоста
public:
    q_type(); // конструктор
    void q(int num) ; // запоминание
    int deq(); // удаление из начала очереди
};

// Конструктор
q_type::q_type()
{
    head = tail = 0;
}

// Помещение значения в очередь
void q_type::q(int num)
{
    if (tail + 1==head || (tail + 1==SIZE && !head)) {
        cout << "Очередь полна";
        return;
    }
    tail++;
    if(tail==SIZE) tail = 0; // круговой цикл
    queue[tail] = num;
}

// Выталкивание значения из очереди
int q_type::deq()
{
    if (head==tail) {
        cout << "Очередь пуста";
        return 0;
    }
    head++;
    if(head==SIZE) head = 0; // круговой цикл
    return queue[head];
}

int main()
{
    q_type q1, q2;
    int i;

    for(i=1; i<=10; i++) {
        q1.q(i);
    }

    // присваивание одного объекта очередь — другому
    q2 = q1;
```

```
    // демонстрация того факта,
    // что обе очереди имеют одинаковое содержимое
    for(i=1; i<=10; i++)
        cout << "Элемент очереди 1: " << q1.deq() << "\n";
    for(i=1; i<=10; i++)
        cout << "Элемент очереди 2: " << q2.deq() << "\n";

    return 0;
}
```

3. Если для хранения очереди память выделяется динамически, тогда, после присваивания одной очереди другой, когда объекты удаляются, для очереди, стоящей в инструкции присваивания слева, память освобождена не будет, а для очереди, стоящей в инструкции присваивания справа, память будет освобождена дважды. Оба этих условия неприемлемы, что и вызывает ошибку.

3.2

1.
```
#include <iostream>
using namespace std;

ttdefine SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); // конструктор
    void push(char ch); // помещает символ в стек
    char pop O; // выталкивает символ из стека
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека \n";
    tos = 0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон \n";
        return;
    }
```

```
    stck[tos] = ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

void showstack(stack o);

int main()
{
    stack s1;
    int i;

    s1.push('a');
    s1.push('b');
    s1.push('c');

    showstack(s1);

    // объект s1 в функции main() по-прежнему существует
    cout << "Стек s1 все еще содержит следующее: \n";
    for(i=0; i<3; i++) cout << s1.pop() << "\n";

    return 0;
}

// Вывод содержимого стека
void showstack(stack o)
{
    char c;

    // когда выполнение этой инструкции завершится,
    // стек о опустеет
    while(c=o.pop()) cout << c << "\n";
    cout << "\n";
}
```

Эта программа выводит на экран следующее:

```
Работа конструктора стека
c
b
a
```

```
Стек пуст
Стек s1 все еще содержит следующее:
c
b
a
```

2. Память для хранения целого, на которую указывает указатель p в объекте o, который используется при вызове функции **neg(),** освобождается, когда при завершении функцией **neg()** своей работы копия объекта o удаляется; однако эта память все еще необходима для объекта o в функции **main()**.

3.3

1.
```
#include <iostream>
using namespace std;

class who {
    char name;
public:
    who(char c) {
        name = c;
        cout << "Создание объекта who #";
        cout << name << "\n";
    }
    ~who() { cout << "Удаление объекта who #" << name << "\n"; }
};

who makewho()
{
    who temp('B');
    return temp;
}

int main()
{
    who ob('A');

    makewho();

    return 0;
}
```

2. Имеется несколько ситуаций, в которых возвращать объект было бы неправильно. Вот одна из таких ситуаций: если при создании объекта файл открывается, а при удалении объекта файл закрывается, тогда к закрытию файла приведет удаление временного объекта, которое происходит, когда функция возвращает объект.

3.4

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

   class pr2; // предварительное объявление

   class pr1 {
       int printing;
       //...
   public:
       pr1() { printing = 0; }
       void set_print(int status) { printing = status; }
       //...
       friend int inuse(pr1 o1, pr2 o2);
   };

   class pr2 {
       int printing;
       //...
   public:
       pr2() { printing = 0; }
       void set_print (int status) { printing = status; }
       //. ..
       friend int inuse (pr1 o1, pr2 o2) ;
   };

   // Возвращает истину при занятом принтере
   int inuse (pr1 o1, pr2 o2)
   {
       if(o1.printing | | o2.printing) return 1;
       else return 0;
   }

   int main ()
   {
       pr1 p1;
       pr2 p2;

       if(!inuse(p1, p2) ) cout << "Принтер свободен\n";

       cout << "Установка для печати принтера p1 ...\n";
       p1.set_print(1);
       if(inuse(p1, p2) ) cout << "Теперь принтер занят\n";

       cout << "Отключение принтера p1 ... \n";
       p1.set_print(0);
       if ( !inuse(p1, p2) ) cout << "Принтер свободен\n";
```

```
    cout << "Подключение принтера p2 ... \n";
    p2.set_print(1);
    if(inuse(p1, p2)) cout << "Теперь принтер занят\n";

    return 0;
}
```

Проверка усвоения материала главы 3

1. Для того чтобы присвоить один объект другому, необходимо, чтобы тип обоих объектов был одинаков.
2. Присваивать **ob1** и **ob2** неправильно, так как память, на которую в начале указывал указатель p объекта **ob2**, теряется, поскольку это значение указателя p при присваивании переписывается. Таким образом, такую память становится невозможно освободить, и наоборот, память, на которую в начале указывал указатель p объекта oBI, при удалении этого объекта освобождается дважды, что может привести к повреждению системы динамического выделения памяти.

3.
```
int light(planet p)
{
    return p.get_miles() / 186000;
}
```

4. Да.

5.
```
// Загрузка алфавита в стек
#include <iostream>
using namespace std;

#define SIZE 27

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); // конструктор
    void push(char ch); // помещает символ в стек
    char pop(); // выталкивает символ из стека
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека \n";
```

```
    tos = 0;
}

// Запись символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

void showstack(stack o);
stack loadstack();

int main()
{
    stack s1;

    s1 = loadstack();
    showstack(s1);

    return 0;
}

// Вывод на экран содержимого стека
void showstack(stack o)
{
    char c;

    // когда выполнение этой инструкции завершится, стек о опустеет
    while(c=o.pop()) cout << c << "\n";
    cout << "\n";
}

// Загрузка стека символами алфавита
stack loadstack()
```

```
{
    stack t;
    char c;
    for(c='a'; c<='z'; c++) t.push(c);
    return t;
}
```

6. При передаче объекта функции в качестве аргумента и при возвращении объекта из функции в качестве возвращаемого значения, создается временный объект, который удаляется при завершении работы функции. При удалении временного объекта деструктор может удалить в программе нечто такое, что может еще пригодиться.

7. Дружественная функция — это функция, не являющаяся членом класса, но обеспечивающая доступ к закрытой части класса, для которого она дружественна. Еще раз, дружественная функция имеет доступ к закрытой части класса, для которого она дружественна, но не является членом этого класса.

Проверка усвоения материала в целом

1.
```
// Загрузка алфавита в стек
#include <iostream>
ttinclude<cctype>
using namespace std;

#define SIZE 27

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack(); // конструктор
    void push(char ch) ; // помещает символ в стек
    char pop(); // выталкивает символ из стека
};

// Инициализация стека
stack::stack()
{
    cout << "Работа конструктора стека \n";
    tos = 0;
}

// Запись символа в стек
void stack::push(char ch)
```

```
{
    if (tos==SIZE) {
        cout << "Стек полон \n";
        return;
    }
    stck[tos] = ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст \n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

void showstack(stack o);
stack loadstack();
stack loadstack(int upper);

int main()
{
    stack s1, s2, s3;

    s1 = loadstack();
    showstack(s1);

    // используется верхний регистр
    s2 = loadstack(1);
    showstack(s2);

    // используется нижний регистр
    s3 = loadstack(0);
    showstack(s3);

    return 0;
}

// Вывод на экран содержимого стека
void showstack(stack o)
{
    char c;

    // когда выполнение этой инструкции завершится, стек о опустеет
    while(c=o.pop()) cout << c << "\n";
```

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

// Загрузка стека символами алфавита
stack loadstack()
{
    stack t;
    char c;

    for(c='a'; c<='z'; c++) t.push(c);
    return t;
}

/* Загрузка стека символами алфавита. Символами верхнего регистра,
если переменная upper равна I, в противном случае символами нижнего
регистра */
stack loadstack(int upper)
{
    stack t;
    char c;

    if(upper) c = 'A';
    else c = 'a';

    for(; toupper(c)<='Z'; c++) t.push(c);
    return t;
}
```

2.

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

class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype();
    void show();
    friend char *get_string(strtype *ob);
};

strtype::strtype(char*ptr)
{
    len=strlen(ptr);
    p=(char *) malloc(len+1);
    if(!p) {
```

```
            cout << "Ошибка выделения памяти\n";
            exit(1);
        }
        strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    free(p);
}

void strtype::show()
{
    cout << p << " - длина: " << len;
    cout << "\n";
}

char *get_string(strtype *ob)
{
    return ob->p;
}

int main()
{
    strtype s1("Это проверка");

    char *s;

    s1.show();

    // получение указателя на строку
    s = get_string(&s1);
    cout << "Это строка, содержащаяся в объекте s1: ";
    cout << s << "\n";

    return 0;
}
```

3. Итог эксперимента следующий: Да, если один объект производного класса присваивается другому, то данные базового класса также копируются. Далее приводится программа для демонстрации этого факта.

```
ttinclude<iostream>
using namespace std;

class base {
    int a;
public:
```

```
    void load_a(int n) { a = n; }
    int get_a() { return a; }
};

class derived: public base {
    int b;
public:
    void load_b (int n) { b = n; }
    int get_b() { return b; }
};

int main()
{
    derived ob1, ob2;

    ob1.load_a(5);
    ob1.load_b(10);

    // Объект ob1 присваивается объекту ob2
    ob2 = ob1;

    cout << "Это переменные a и b объекта ob1: ";
    cout << ob1.get_a() << ' ' << ob1.get_b() << "\n";

    cout << "Это переменные a и b объекта ob2: ";
    cout << ob2.get_a() << ' ' << ob2.get_b() << "\n";
    /* Как вы, вероятно, догадались, выводимые на экран значения
одинаковы для обоих объектов
*/
    return 0;
}
```

ГЛАВА 4

Повторение пройденного

1. При присваивании одного объекта другому объекту того же типа, текущие значения данных всех членов объекта, стоящего справа от оператора присваивания, передаются соответствующим членам объекта, стоящего слева от оператора присваивания.
2. При присваивании одного объекта другому, ошибка может случиться тогда, когда эта операция перезаписывает важные данные, уже существующие в целевом объекте. Например, если значение указателя на динамически выделенную область памяти или на открытый файл будет перезаписано, то очевидно, что тогда теряется исходное значение указателя.

3. Если объект передается функции в качестве аргумента, создается его копия. Однако конструктор копии не вызывается. При завершении работы функции, когда объект удаляется, вызывается деструктор копии.
4. Нарушение принципа независимости объекта и его копии при передаче параметра может возникать в нескольких ситуациях. Например, если деструктор освобождает динамическую память, тогда эта память для аргумента будет потеряна. В общем случае, если деструктор удаляет что-то, что требуется исходному аргументу, аргумент будет испорчен.

5.
```
#include <iostream>
using namespace std;

class summation {
    int num;
    long sum; // суммирование чисел num
public:
    void set_sum(int n) ;
    void show_sum() {
cout << " сумма чисел " << num << " равна " << sum << "\n";
    }
};

void summation::set_sum(int n)
{
    int i;
    num = n;
    sum = 0;
    for(i=1; i<=n; i++)
        sum += i;
}

summation make_sum()
{
    int i;
    summation temp;
    cout << "Введите число: ";
    cin >> i;
    temp.set_sum(i);
    return temp;
}

int main()
{
    summation s;
```

```
    s=make_sum();

    s.show_sum();

    return 0;
}
```

6. Для некоторых компиляторов требуется, чтобы во встраиваемых функциях не было циклов.

7.
```
#include <iostream>
using namespace std;

class myclass {
    int num;
public:
    myclass(int x) { num = x; }
    friend int isneg(myclass ob);
};

int isneg(myclass ob)
{
    return (ob.num < 0) ? 1: 0;
}

int main()
{
    myclass a(-1), b(2);

    cout << isneg(a) << ' ' << isneg(b);
    cout << "\n";

    return 0;
}
```

8. Да, дружественная функция может быть дружественной более чем одному классу.

Упражнения

4.1

1.
```
#include <iostream>
using namespace std;

class letters {
    char ch;
```

```
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};

int main()
{
    letters ob[10] = { 'a', 'b', 'c', 'd', 'e',
                       'f', 'g', 'h', 'i', 'j' };

    int i;

    for(i=0; i<10; i++)
        cout << ob[i].get_ch() << ' ';

    cout << "\n";

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class squares {
    int num, sqr;
public:
    squares(int a, int b) { num = a; sqr = b; }
    void show() {cout << num << ' ' << sqr << "\n"; }
};

int main()
{
    squares ob[10] = {
        squares(1, 1),
        squares(2, 4),
        squares(3, 9),
        squares(4, 16),
        squares(5,25),
        squares(6, 36),
        squares(7,49),
        squares(8,64),
        squares(9, 81),
        squares(10, 100),
    };
    int i;

    for(i=0; i<10; i++) ob[i].show();

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class letters {
    char ch;
public:
    letters(char c) { ch = c; }
    char get_ch() { return ch; }
};

int main()
{
    letters ob[10] = {
        letters('a'),
        letters('b'),
        letters('c'),
        letters('d'),
        letters('e'),
        letters('f'),
        letters('g'),
        letters('h'),
        letters('i'),
        letters('j')
     };

    int i;

    for(i=0; i<10; i++)
        cout << ob[ i ].get_ch() << ' ';

    cout << "\n";

    return 0;
}
```

4.2

1.
```
// Вывод содержимого массива в обратном порядке
#include <iostream>
using namespace std;

class samp {
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};
```

```
int main()
{
    samp ob[4] = {
        samp(1, 2),
        samp(3,4),
        samp(5, 6),
        samp(7, 8)
    };
    int i;

    samp *p;

    p = &ob[3]; // получение адреса последнего элемента массива

    for(i=0; i<4; i++) {
        cout << p -> get_a() << ' ';
        cout << p -> get_b() << "\n";
        p--; // переход к предыдущему объекту
    }

    cout << "\n";

    return 0;
}
```

2.
```
/* Создание двумерного массива объектов с доступом к элементам через
указатель
*/
#include <iostream>
using namespace std;

class samp {
    int a;
public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob[4] [2] = {
        1, 2,
        3, 4,
        5, 6,
        7, 8
    };
    int i;
```

```
    samp *p;

    p = (samp *) ob;

    for(i=0; i<4; i++) {
        cout <<p->get_a() << ' ' ;
        P++;
        cout << p->get_a() << "\n";
        P++;
    }

    cout << "\n";

    return 0;
}
```

4.3

1.
```
// Использование указателя this
#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    myclass(int n, int m) { this->a = n; this->b = m; }
    int add() { return this->a + this->b; }
    void show();
};

void myclass::show()
{
    int t;

    t = this->add(); // вызов функции-члена
    cout << t <<"\n";
}

int main()
{
    myclass ob(10, 14);

    ob.show();

    return 0;
}
```

4.4

1.
```
#include <iostream>
using namespace std;

int main()
{
    float *f;
    long *l;
    char *c;

    f = new float;
    l = new long;
    c = new char;

    if(!f|| !l I I !c) {
        cout << "Ошибка выделения памяти.";
        return 1;
    }

    *f = 10.102;
    *l = 100000;
    *c = 'A';

    cout << *f << ' ' << *l << ' ' << *c;
    cout <<'\n';

    delete f; delete l; delete c;

    return 0;
}
```

2.
```
#include <iostream>
#include <cstring>
using namespace std;

class phone {
    char name[40];
    char number[14];
public:
    void store(char *n, char *num);
    void show();
};

void phone::store(char *n, char *num)
{
    strcpy(name,n);
    strcpy(number,num);
}
```

```
void phone::show()
{
    cout << name << ": " << number;
    cout << "\n";
}

int main()
{
    phone *p;

    p = new phone;

    if(!p) {
        cout << "Ошибка выделения памяти.";
        return 1;
    }

    p->store("Исаак Ньютон", "111 555-2323");

    p->show();

    delete p;

    return 0;
}
```

3. В случае неудачной попытки выделения памяти, оператор **new** может либо возвратить нулевой указатель, либо возбудить исключительную ситуацию. Чтобы выяснить, какой механизм используется в вашем компиляторе, необходимо просмотреть соответствующую техническую документацию. В соответствии с требованиями стандарта по C++, оператор new по умолчанию возбуждает исключительную ситуацию.

4.5

1.
```
char *p;
p = new char[100];
// ....
strcpy(p, "Это проверка");
```

2.
```
ttinclude <iostream>
using namespace std;

int main()
{
    double *p;

    p = new double(-123.0987);
```

```
    cout << *p << '\n';

    return 0;
}
```

4.6

1.
```
#include <iostream>
using namespace std;

void rneg(int &i); // версия функции со ссылкой
void pneg(int *i); // версия функции с указателем

int main()
{
    int i = 10;
    int j = 20;

    rneg(i);
    pneg(&j);

    cout << i << ' ' << j << '\n';

    return 0;
}

// использование параметра-ссылки
void rneg(int &i)
{
    i = -i;
}

// использование параметра-указателя
void pneg(int *i)
{
    *i = -*i;
}
```

2. При вызове функции **triple()** адрес d получен явно, посредством оператора &. Это ненужно и неправильно. При использовании ссылки в качестве параметра перед аргументом не ставится оператор &.

3. Адрес параметра-ссылки передается в функцию автоматически. Для получения адреса нет необходимости производить какие бы то ни было действия. Передача по ссылке быстрее, чем передача по значению. При передаче по ссылке не делается копии аргумента и поэтому не бывает сторонних эффектов, связанных с вызовом деструктора копии.

4.7

1. В исходной программе объект передается в функцию **show()** по значению. Поэтому делается его копия. Когда функция **show()** возвращает свое значение, копия удаляется и при этом вызывается деструктор копии. Это приводит к освобождению памяти, на которую указывает указатель p, но освобожденная память все еще необходима аргументам функции **show()**. Здесь представлена правильная версия программы, в которой, для того чтобы предотвратить появление копии при вызове функции, в качестве параметра используется ссылка:

```
// Теперь программа исправлена
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s) + 1;

    p = new char [l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    strcpy(p, s);
}

// Проблема решена с помощью параметра-ссылки
void show(strtype &x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Привет"), b("Здесь");
```

```
    show(a);
    show(b);

    return 0;
}
```

4.8

1.
```
// Пример защищенного двумерного массива
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
    int isize, jsize;
    int *p;
public:
    array(int i, int j);
    int&put(int i, int j);
    int get(int i, int j);
};

array::array(int i, int j)
{
    p = new int [ i * j ];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    isize = i;
    jsize = j;
}

// Запись значения в массив
int &array::put(int i, int j)
{
    if(i<0 || i>=isize || j<0 || j>=jsize) {
        cout << "Ошибка, нарушены границы массива!!!\n";
        exit(1);
    }
    return p[i * jsize + j]; // возврат ссылки на p[ i ]
}

// Получение значения из массива
int array::get(int i, int j)
{
    if(i<0 || i>=isize || j<0 || j>=jsize) {
        cout << "Ошибка, нарушены границы массива!!!\n";
        exit(1);
    }
```

```
    return p[i * jsize + j]; // возврат символа
}

int main()
{
    array a(2, 3);
    int i, j;

    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            a.put(i, j) = i + j ;
    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            cout << a.get(i, j) << ' ';

    // генерация ошибки нарушения границ массива
    a.put(10, 10);

    return 0;
}
```

2. Нет. Возвращаемую функцией ссылку нельзя присвоить указателю.

Проверка усвоения материала главы 4

1.
```
#include <iostream>
using namespace std;

class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    a_type ob[2][5] = {
        a_type(1, 1), a_type(2, 2),
        a_type(3, 3), a_type(4, 4),
        a_type(5, 5), a_type(6, 6),
        a_type(7, 7), a_type(8, 8),
        a_type(9, 9), a_type(10, 10)
    };
```

```
    int i, j;

    for(i=0; i<2; i++)
        for(j=0; j<5; j++)
            ob[i][j].show();

    cout<<'\n';

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class a_type {
    double a, b;
public:
    a_type(double x, double y) {
        a = x;
        b = y;
    }
    void show() { cout << a << ' ' << b << "\n"; }
};

int main()
{
    a_type ob[2][5] = {
        a_type(1, 1), a_type(2, 2),
        a_type(3, 3), a_type(4, 4),
        a_type(5, 5), a_type(6, 6),
        a_type(7, 7), a_type(8, 8),
        a_type(9, 9), a_type(10, 10)
    };

    a_type *p;

    p = (a_type *) ob;

    int i, j;

    for(i=0; i<2; i++)
        for(j=0; j<5; j++) {
            p->show();
            p++;
        }

    cout<<'\n';

    return 0;
}
```

3. Указатель **this** — это указатель, который автоматически передается функции-члену и который указывает на объект, вызвавший функцию.

4. Основными формами операторов **new** и **delete** являются следующие:

```
p-var = new type;
delete p-var;
```

При использовании оператора **new** нет необходимости в приведении типов. Размер объекта определяется автоматически, поэтому не нужен оператор sizeof. Кроме этого, незачем включать в программу заголовок **<cstdlib>.**

5. Ссылка по существу является скрытым константным указателем и просто играет роль другого имени переменной или аргумента. Преимущество от использования параметра-ссылки в том, что никакой копии аргумента не делается.

6.
```
ttinclude <iostream>
using namespace std;

void recip(double &d);

int main()
{
    double x = 100.0;

    cout << "x равно " << x << '\n';

    recip(x);

    cout << "Обратная величина равна " << x << '\n';

    return 0;
}

void recip(double &d)
{
    d = 1/d;
}
```

Проверка усвоения материала в целом

1. Для доступа к члену объекта с помощью указателя используется оператор стрелка **(->)**.

2.
```
#include <iostream>
#include <cstring>
ttinclude <cstdlib>
using namespace std;
```

```
class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype();
    void show();
};

strtype::strtype(char*ptr)
{
    len = strlen(ptr);
    p = new char[len+1];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}

strtype::~strtype()
{
    cout << "Освобождение памяти по адресу p\n";
    delete [] p;
}

void strtype::show()
{
    cout << p << " — длина: " << len;
    cout << "\n";
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    s1.show();
    s2.show();

    return 0;
}
```

ГЛАВА 5

Повторение пройденного

1. Ссылка — это особый тип указателя, который разыменовывается автоматически. Ссылка в инструкциях может использоваться точно так же, как объект,

на который она указывает. Имеются три вида ссылок: ссылка может быть параметром, ссылка может быть возвращаемым значением функции и, кроме этого, ссылка может быть независимой. Самыми важными являются ссылка в качестве параметра и ссылка в качестве возвращаемого значения функции.

2.
```
#include <iostream>
using namespace std;

int main()
{
    float *f;
    int *i;

    f = new float;
    i = new int;

    if(!f || !i) {
        cout << "Ошибка выделения памяти\n";
        return 1;
    }

    *f = 10.101;
    *i = 100;

    cout << *f << ' ' << *i << '\n';

    delete f;
    delete i;

    return 0;
}
```

3. Здесь показана основная форма оператора new, используемая для инициализации динамических переменных:

указатель_на_переменную = new ***тип (инициализирующее_значение);***

Например, в следующем фрагменте выделяется память для целого и этому целому присваивается значение 10:

```
int *p;
p = new int(10);
```

4.
```
#include <iostream>
using namespace std;

class samp {
    int x;
public:
    samp(int n) { x = n; }
```

```
    int getx() { return x; }
};

int main()
{
    samp A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int i;

    for(i=0; i<10; i++)     cout << A[ i ].getx() << ' ';

    cout << "\n";

    return 0;
}
```

5. *Достоинства:* Ссылка в качестве параметра при вызове функции не приводит к появлению копии объекта. Передача параметра по ссылке часто быстрее, чем его передача по значению. Параметр-ссылка упрощает синтаксис и процедуру вызова функции по ссылке, снижая вероятность ошибки.

   Недостатки: Изменения в параметре-ссылке меняют и используемую в вызове исходную переменную, следовательно параметр-ссылка открывает возможность сторонних эффектов в вызывающей программе.

6. Нет.

7.
```
#include <iostream>
using namespace std;

void mag (long &num, long order);

int main()
{
    long n = 4;
    long o = 2;

    cout << "Значение 4, увеличенное на два порядка равно ";
    mag(n, o);
    cout << n << '\n';

    return 0;
}

void mag (long &num, long order)
{
    for(; order; order --) num = num * 10;
}
```

Упражнения

5.1

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype();
    strtype(char *s, int l);
    char *getstring() { return p; }
    int getlength() { return len; }
};

strtype::strtype()
{
    p = new char[255];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    *p = '\0'; // нулевая строка
    len = 255;
}

strtype::strtype(char *s, int l)
{
    if(strlen(s) >= l) {
        cout << "Выделено слишком мало памяти!\n";
        exit(1);
    }
    p = new char[ l ];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, s);
    len = l;
}

int main()
```

```
{
    strtype s1;
    strtype s2("Это проверка", 100);

    cout << "Строка s1: " << s1.getstring() << "- Длиной: ";
    cout << s1.getlength() << "\n";

    cout << "Строка s2: " << s2.getstring() << "- Длиной: ";
    cout << s2.getlength() << "\n";

    return 0;
}
```

2.
```
// Имитация секундомера
#include <iostream>
#include <ctime>
using namespace std;

class stopwatch {
    double begin, end;
public:
    stopwatch();
    stopwatch (clock_t t);
    ~stopwatch();
    void start();
    void stop();
    void show();
};

stopwatch::stopwatch()
{
    begin = end = 0.0;
}

stopwatch::stopwatch(clock_tt)
{
    begin = (double) t / CLOCKS_PER_SEC;
    end = 0.0;
}

stopwatch::~stopwatch()
{
    cout << "Удаление объекта stopwatch ...";
    show();
}

void stopwatch::start()
{
    begin = (double) clock() / CLOCKS_PER_SEC;
}
```

```
void stopwatch::stop()
{
    end = (double) clock() / CLOCKS_PER_SEC;
}

void stopwatch::show()
{
    cout << "Истекшее время: " << end - begin;
    cout << "\n";
}

int main()
{
    stopwatch watch;
    long i;

    watch.start();
    for(i=0; i<3200000; i++); // время цикла
    watch.stop();
    watch.show();

    // Создание объекта с использованием его начального значения
    stopwatch s2(clock());
    for(i=0; i<250000; i++); // время цикла
    s2.stop();
    s2.show();

    return 0;
}
```

5.2

1. Для объектов **obj** и **temp** вызывается обычный конструктор. Однако, когда объект **temp** становится возвращаемым значением функции **f(),** создается временный объект, который генерирует вызов конструктора копий.

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

class myclass {
public:
    myclass();
    myclass(const myclass &o);
    myclass f();
};

// Обычный конструктор
```

```
myclass::myclass()
{
    cout << "Работа обычного конструктора\n";
}

// Конструктор копий
myclass::myclass(const myclass &o)
{
    cout << "Работа конструктора копий\n";
}

// Возвращение объекта
myclass myclass::f()
{
    myclass temp;

    return temp;
}

int main()
{
    myclass obj;

    obj=obj.f();

    return 0;
}
```

2. В соответствии с программой, когда объект передается в функцию **getval()**, создается его поразрядная копия. Когда функция **getval()** возвращает свое значение и копия удаляется, выделенная для этого объекта память (на которую указывает указатель p) освобождается. Однако эта память по-прежнему требуется исходному объекту, который использовался при вызове функции **getval().** Далее представлена исправленная версия программы. Для решения проблемы потребовался конструктор копий.

```
// Исправленная версия программы
#include <iostream>
#include <cstdlib>
using namespace std;

class myclass {
    int *p;
public:
    myclass(int i);
    myclass(const myclass &o); // конструктор копий
    ~myclass() {delete p;}
    friend int getval(myclass o);
};
```

```
myclass::myclass(int i)
{
    p=new int;

    if(!p) {
    cout << "Ошибка выделения памяти\n";
    exit(1);
    }
    *p=i;
}

// Конструктор копий
myclass::myclass(const myclass &o)
{
    p=new int;    // выделение памяти для копии

    if(!p) {
    cout << "Ошибка выделения памяти\n";
    exit(1);
    }
    *p=*o.p;
}

int getval(myclass o)
{
    return *o.p; // получение значения
}

int main()
{
    myclass a(1), b(2);

    cout << getval(a) << " " << getval(b);
    cout << "\n";
    cout << getval(a) << " " << getval(b);

    return 0;
}
```

3. Конструктор копий вызывается, когда один объект используется для инициализации другого. Обычный конструктор вызывается, когда объект создается.

5.4

1.
```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;

long mystrtol (const char *s, char **end, int base = 10)
{
    return strtol(s, end, base);
}

int main()
{
    long x;
    char *s1 = "100234";
    char *p;

    x = mystrtol (s1, &p, 16);
    cout << "Основание системы счисления 16: " << x << '\n';

    x = mystrtol (s1, &p, 10);
    cout << "Основание системы счисления 10: " << x << '\n';

    x = mystrtol (s1, &p) ; // основание системы счисления,
                            // заданное по умолчанию
    cout << "Основание системы счисления по умолчанию 10: "
                                                    << x << '\n';

    return 0 ;
}
```

2. Все параметры, которые получают аргументы по умолчанию, должны находиться правее параметров, которые этих аргументов не получают. Таким образом, после того как вы начали присваивать значения аргументам по умолчанию, все последующие параметры также должны получать свои значения по умолчанию. Что касается приведенного в вопросе прототипа функции, то в нем параметр q значения по умолчанию не получает.

3. Поскольку функции управления позиционированием курсора у различных компиляторов и сред программирования разные, показано только одно из возможных решений задачи. Следующая программа предназначена для работы в среде программирования Borland C++.

```
/* Эта программа работает только в среде программирования Borland C++
*/
#include <iostream>
#include <conio.h>
using namespace std;

void myclreol (int len = -1);

int main()
{
    int i;
```

```
    gotoxy(1, 1);
    for(i=0; i<24; i++)
        cout << "abcdefghijklmnopqrstuvwxyz1234567890\n";

    gotoxy(1, 2);
    myclreol();
    gotoxy(1, 4);
    myclreol(20);



// Стирание строки на величину, заданную параметром len
void myclreol (int len)
{
    int x, y;

    x = wherex(); // получение положения по x
    y = wherey(); // получение положения по y

    if (len == -1) len = 80 - x;

    int i = x;

    for(; i<=len; i++) cout << ' ';

    gotoxy(x, y) ; // установка курсора
}
```

4. Аргумент по умолчанию не может быть другим параметром или локальной переменной.

5.6

1.
```
#include <iostream>
using namespace std;

int dif(int a, int b)
{
    return a — b;
}

float dif(float a, float b)
{
    return a — b;
}

int main()
{
    int (*p1)(int, int);
```

```
    float (*p2)(float, float);

    p1 = dif; // адрес функции dif(int, int)
    p2 = dif; // адрес функции dif(float, float)

    cout << p1(10, 5) << ' ';
    cout << p2(10.5, 8.9) << '\n';

    return 0;
}
```

Проверка усвоения материала главы 5

1.
```
// Перегрузка конструктора date() для параметра типа time_t
#include <iostream>
#include <cstdio> // заголовок включен для функции sscanf()
#include <ctime>
using namespace std;

class date {
    int day, month, year;
public:
    date(char*str);
    date (int m, int d, int y) {
        day = d;
        month = m;
        year = y;
    }
    // Перегрузка конструктора для параметра типа time_t
    date(time_t t);
    void show() {
        cout << month << '/' << day << '/';
        cout << year << '\n';
    }
};

date::date(char*str)
{
    sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

date::date(time_t t)
{
    struct tm *p;

    p = localtime(&t);
    day = p -> tm_mday;
    month = p -> tm_mon;
```

```
    year = p -> tm_year;
}

int main()
{
    // Образование даты с помощью строки
    date sdate("11/1/92");

    // Образование даты с помощью трех целых
    date idate(11, 1, 92);

    /* Образование даты с помощью параметра типа time_t, что ведет к
       созданию объекта, использующего системную дату
    */
    date tdate(time(NULL));

    sdate.show();
    idate.show();
    tdate.show();

    return 0;
}
```

2. В классе **samp** определен только один конструктор — конструктор с параметром. Поэтому нельзя объявлять объект типа **samp** без параметра. (То есть инструкция **samp x** — это неправильное объявление.)

3. Первый довод в пользу перегрузки конструктора состоит в том, что такая перегрузка обеспечивает гибкость, позволяя вам выбрать в каждом конкретном случае наиболее подходящий конструктор. Другой довод в том, что перегрузка позволяет объявлять как инициализируемые, так и не инициализируемые объекты. Вам может потребоваться перегрузить конструктор для динамического выделения памяти под массив.

4. Ниже представлена основная форма конструктора копий:

```
имя_класса (const имя_класса &объект) {
    тело_конструктора
}
```

5. Конструктор копий вызывается, когда имеет место инициализация, а именно: когда один объект явно используется для инициализации другого, когда объект передается в функцию в качестве параметра, когда создается временный объект при возвращении объекта функцией.

6. Ключевое слово **overload** является устаревшим. В ранних версиях C++ оно информировало компилятор о том, что функция будет перегружена. В современных компиляторах это ключевое слово не поддерживается.

7. Аргумент по умолчанию — это значение, которое присваивается параметру функции при ее вызове, если при этом соответствующий аргумент функции не указан.

8.
```
#include <iostream>
#include <cstring>
using namespace std;

void reverse(char *str, int count = 0);

int main()
{
    char *s1 = "Это проверка";
    char *s2 = "Мне нравится C++";

    reverse(s1); // Реверс всей строки
    reverse(s2, 7); // Реверс первых семи знаков

    cout << s1 << '\n';
    cout << s2 << '\n';

    return 0;
}

void reverse(char *str, int count)
{
    int i, j;
    char temp;

    if(!count) count = strlen(str) - 1;

    for(i=0, j=count; i<j; i++, j--) {
        temp = str[ i ];
        str[ i ] = str[ j ];
        str[ j ] = temp;
    }
}
```

9. Все параметры, получающие аргументы по умолчанию, должны находиться правее параметров, не получающих таких аргументов.

10. Неоднозначность может возникнуть, когда по умолчанию происходит преобразование типа, а также при использовании параметра-ссылки или аргумента по умолчанию.

11. Пример неоднозначен, поскольку компилятор не может определить, какую версию функции **compute()** следует вызвать. Вызвать ли первую версию с аргументом по умолчанию **divisor,** или вторую — в которой функция получает только один параметр?

12. При получении адреса перегруженной функции с помощью указателя, конкретную ее версию определяет способ объявления указателя. То есть для всех перегруженных версий функции, адреса которых мы хотели бы получить, должна объявляться своя версия указателя.

Проверка усвоения материала в целом

1.
```
#include <iostream>
using namespace std;

void order(int &a, int &b)
{
    int t;

    if(a<b) return;
    else { // a и Ь меняются местами
        t = a;
        a = b;
        b - t;
    }
}

int main()
{
    int x = 10, y = 5;

    cout << "x: " << x << ", y: " << y << '\n';

    order(x,y);
    cout << "x: " << x << ", y: " << y << '\n';

    return 0;
}
```

2. Синтаксис вызова функции, параметр которой передается по ссылке, идентичен синтаксису вызова функции, параметр которой передается по значению.

3. Аргумент по умолчанию, фактически, является компактной записью перегрузки функции, поскольку приводит к тому же результату. Например, инструкция

```
int f(int a, int b = 0);
```

идентична следующим двум перегруженным функциям:

```
int f(inta);

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

4.
```
ttinclude <iostream>
using namespace std;
class samp {
    int a;
public:
    samp() .{ a = 0; }
    samp(int n) { a = n; }
    int get_a() { return a; }
};

int main()
{
    samp ob(88);
    samp obarray[10];

    // ...
}
```

5. Конструкторы копий необходимы, если программист хотел бы точно управлять процессом создания копий объекта. Это важно только в том случае, если создаваемые по умолчанию поразрядные копии по каким-либо причинам нежелательны.

ГЛАВА 6

Повторение пройденного

1.
```
class myclass {
    int x, y;
public:
    myclass(int i, int j) { x = i; y = j; }
    myclass() { x = 0; y = 0; }
};
```

2.
```
class myclass {
    int x, y;
public:
    myclass(int i = 0, int j = 0) { x = i; y = j; }
};
```

3. В объявлении функции после появления аргумента по умолчанию не должно быть обычных аргументов.

4. Функции нельзя перегружать, если их отличие только в том, что одна получает параметр по значению, а вторая — по ссылке. (Компилятор не в состоянии их отличить.)
5. Аргументы по умолчанию лучше использовать тогда, когда можно с уверенностью предположить, что при работе приложения одно или более значений будут встречаться чаще других. Аргументы по умолчанию лучше не использовать, когда такой уверенности нет.
6. Нет, поскольку нельзя инициализировать динамический массив. В этом классе имеется только один конструктор, которому требуется инициализация.
7. Конструктор копий — это особый конструктор, который вызывается при инициализации одного объекта другим. Такая инициализация имеет место в следующих трех случаях: когда один объект явно используется для инициализации другого, когда объект передается в функцию в качестве параметра, и когда в качестве возвращаемого значения функции создается временный объект.

Упражнения

6.2

1.
```
// Перегрузка операторов * и / относительно класса coord
ttinclude<iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int sj) { i = x; j = y; }
    coord operator* (coord ob2);
    coord operator/ (coord ob2);
};

// Перегрузка оператора * относительно класса coord
coord coord::operator*(coord ob2)
{
    coord temp;

    temp.x = x * ob2.x;
    temp.y = y * ob2.y;

    return temp;
}
```

```
// Перегрузка оператора / относительно класса coord
coord coord: : operator/ (coordob2)
{
    coord temp;

    temp.x = x / ob2.x;
    temp.y = y / ob2.y;

    return temp;
}

int main()
(
    coord o1 (10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 * o2;
    o3.get_xy(x, y) ;
    cout << "(o1 * o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 / o2;
    o3.get_xy(x, y) ;
    cout << "(o1 / o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

2. Так перегружать оператор % нежелательно, поскольку перегруженная версия оператора не связана с его традиционным использованием.

6.3

1.
```
// Перегрузка операторов < и > относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    int operator<(coord ob2);
    int operator>(coord ob2);
};

// Перегрузка оператора < для класса coord
int coord::operator<(coord ob2)
```

```
{
    return x<ob2.x && y<ob2.y;
}

// Перегрузка оператора > для класса coord
int coord::operator>(coord ob2)
{
    return x>ob2.x && y>ob2.y;
}

int main()
{
    coord o1(10, 10), o2(5, 3);

    if(o1>o2) cout << "o1 > o2\n";
    else cout << "o1 <= o2\n";

    if(o1<o2) cout << "o1 < o2\n";
    else cout << "o1 >= o2\n";

    return 0;
}
```

6.4

1.
```
// Перегрузка оператора -- относительно класса coord
ttinclude<iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x=i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator--(); // префиксная форма
    coord operator--(int notused) ; // постфиксная форма
};

// Перегрузка префиксной формы оператора — для класса coord
coordcoord::operator--()
{
    x--;
    y--;

    return *this;
}

// Перегрузка постфиксной формы оператора — для класса coord
coord coord::operator--(int notused)
```

```
{
    x--;
    y--;

    return *this;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    o1--; // декремент объекта
    o1.get_xy(x, y);
    cout << "(o1--) X: " << x << ", Y: " << y << "\n";

    --o1; // декремент объекта
    o1.get_xy(x, y);
    cout << "(--o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

2.
```
// Перегрузка оператора + относительно класса coord
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int sj) { i = x; j = y; }
    coord operator+(coord ob2); // бинарный плюс
    coord operator+(); // унарный плюс
};

// Перегрузка бинарного оператора + для класса coord
coord coord::operator+(coord ob2)
{
    coord temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
}

// Перегрузка унарного оператора + для класса coord
coord coord::operator+()
```

```
{
    if(x<0) x = -x;
    if(y<0) y = -y;

    return *this;
}

int main()
{
    coord o1(10, 10), o2(-2, -2);
    int x, y;

    o1 = o1 + o2; // сложение
    o1.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    o2 = +o2; // абсолютное значение
    o2.get_xy(x, y);
    cout << "(+o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

6.5

1.
```
/* Перегрузка операторов – и / относительно класса coord с
использованием дружественных функций
*/
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y=j; }
    void get_xy(int &i, int Sj) { i = x; j = y; }
    friend coord operator-(coord ob1, coord ob2);
    friend coord operator/(coord ob1, coord ob2);
};

// Перегрузка оператора – для класса coord
// с использованием дружественной функции .
coord operator-(coord ob1, coord ob2)
{
    coord temp;

    temp.x = ob1.x – ob2.x;
```

```
    temp.y= ob1.y — ob2.y;

    return temp;
}

// Перегрузка оператора / для класса coord
// с использованием дружественной функции
coord operator/ (coord ob1,\ coord ob2)
{
    coord temp;

    temp.x = ob1.x / ob2.x;
    temp.y = ob1.y / ob2.y;

    return temp;
}

int main()
{
    coord o1 (10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 - o2;
    o3.get_xy(x, y);
    cout << "(o1 - o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 / o2;
    o3.get_xy(x, y);
    cout << "(o1 / o2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

2.
```
// Перегрузка оператора * для операций ob*int и int*ob
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y= 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int Si, int &j) { i = x; j = y; }
    friend coord operator* (coord ob1, int i);
    friend coord operator* (int i, coord ob2);
};

// Перегрузка оператора * первым способом
coord operator* (coord ob1, int i)
```

```
{
    coord temp;

    temp.x = ob1.x * i;
    temp.y = ob1.y * i;

    return temp;
}

// Перегрузка оператора * вторым способом
coord operator* (int i, coord ob2)
{
    coord temp;

    temp.x = ob2.x * i;
    temp.y = ob2.y * i;

    return temp;
}

int main()
{
    coord o1(10, 10), o2;
    int x, y;

    o2 = o1 * 2; // ob * int
    o2.get_xy(x, y);
    cout << " (o1 * 2) X: " << x << ", Y: " << y << "\n";

    o2 = 3 * o1; // int * ob
    o2.get_xy(x, y);
    cout << "(3 * o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

3. Благодаря использованию дружественных оператор-функций, стало возможным получить, в качестве левого операнда, встроенный тип данных. При использовании функций-членов, левый операнд должен быть объектом класса, для которого определяется оператор.

4.
```
/* Перегрузка оператора -- относительно класса coord
с использованием дружественных функций
*/
#include <iostream>
using namespace std;

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y = 0; }
```

```
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    // префиксная форма
    friend coord operator--(coord &ob);

    // постфиксная форма
    friend coord operator--(coord &ob, int notused);
};

// Перегрузка префиксной формы оператора - для класса coord
// с использованием дружественной функции
coord operator--(coord Sob)
{
    ob.x--;
    ob.y --;
    return ob;
}

// Перегрузка постфиксной формы оператора - для класса coord
// с использованием дружественной функции
coord operator--(coord &ob, int notused)
{
    ob.x--;
    ob.y --;
    return ob;
}

int main()
{
    coord o1(10, 10);
    int x, y;

    --o1; // декремент объекта o1
    o1.get_xy(x, y);
    cout << "(--o1) X: " << x << ", Y: " << y << "\n";

    o1--; // декремент объекта o1
    o1.get_xy(x, y);
    cout << "(o1--) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

6.6

1.
```
#include <iostream>
ttinclude<cstdlib>
using namespace std;
```

```
class dynarray {
    int *p;
    int size;
public:
    dynarray(int s);
    int &put(int i);
    int get(int i);
    dynarray &operator=(dynarray &ob);
};

// Конструктор
dynarray::dynarray(int s)
{
    p = new int[s];
    if(!p) <
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    size = s;
}

// Запоминание элемента
int &dynarray::put(int i)
{
    if(i<0 I I i>=size) {
        cout << "Ошибка нарушения границ массива!\n";
        exit(1);
    }

    return p[i];
}

// Получение элемента
int dynarray::get(int i)
{
    if(i<0 I I i>=size) {
        cout << "Ошибка нарушения границ массива!\n";
        exit(1);
    }

    return p[i];
}

// Перегрузка оператора = для класса dynarray
dynarray &dynarray::operator=(dynarray &ob)
{
    int i;
```

```
    if(size!=ob.size) {
        cout << "Нельзя копировать массивы разных размеров!\n";
        exit(1);
    }

    for(i=0; i<size; i++) p[i] = ob.p[i];
    return *this;
}

int main()
{
    int i;

    dynarray ob1(10), ob2(10), ob3(100);

    ob1.put(3) = 10;
    i = ob1.get(3);
    cout << i << "\n";

    ob2 = ob1;

    i = ob2.get(3);
    cout << i << "\n";

    // Выполнение следующей инструкции ведет к ошибке
    ob1 = ob3; // !!!
    return 0;
}
```

6.7

1.
```
#include <iostream>
#include <cstring>
tfinclude<cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *s);
    ~strtype() {
        cout << "Освобождение памяти по адресу " <<
                                        (unsigned) p << '\n';
        delete []p;
    }
    char *get() { return p; }
    strtype &operator=(strtype &ob);
```

```
    char &operator[](int i);
};

strtype::strtype(char *s)
{
    int l;

    l = strlen(s) + 1;

    p = new char [l];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    len = l;
    strcpy(p, s);
}

// Присваивание объекта
strtype &strtype::operator=(strtype &ob)
{
    // Выяснение необходимости дополнительной памяти
    if(len< ob.len) .{ // требуется выделение
                       // дополнительной памяти
        delete [] p;
        p = new char [ob.len];
        if(!p) {
            cout << "Ошибка выделения памяти\n";
            exit(1);
        }
    }
    len = ob.len;
    strcpy(p, ob.p);
    return *this;
}

// Индексируемые символы в строке
char &strtype::operator[](int i)
{
    if(i<0 || i>len-1) {
        cout << "\nЗначение индекса ";
        cout << i << " выходит за границы массива\n";
        exit(1);
    }

    return p[i];
}
```

```
int main()
{
    strtype a("Привет"), b("Здесь");

    cout << a.get() << ' \n';
    cout << b.get() << ' \n';

    a = b; // теперь указатель p не перезаписывается

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    // Доступ к символам посредством индексирования массива
    cout << a[0] << a[1] << a[2] << "\n";

    // Присваивание символов посредством индексирования массива
    a[0] = 'X';
    a[1] = 'Y';
    a[2] = 'Z';

    cout << a.get() << "\n";

    return 0;
}
```

2.
```
#include <iostream>
#include <cstdlib>
using namespace std;

class dynarray {
    int *p;
    int size;
public:
    dynarray(int s);
    dynarray &operator=(dynarray &ob);
    int &operator[](int i);
};

// Конструктор
dynarray::dynarray(int s)
{
    p = new int[s];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    size = s;
}
```

```
// Перегрузка оператора = для класса dynarray
dynarray &dynarray::operator=(dynarray &ob)
{
    int i;

    if(size!=ob.size) {
        cout << "Нельзя копировать массивы разных размеров! \n";
        exit(1);
    }

    for(i=0; i<size; i++) p[i] = ob.p[i];
    return *this;
}

// Перегрузка оператора []
int &dynarray::operator[](int i)
{
    if(i<0 I I i>size) (
        cout << "\nЗначение индекса ";
        cout << i << " выходит за границы массива\n";
        exit(1);
    }
    return p[i];
}

int main()
{
    int i;

    dynarray ob1(10), ob2(10), ob3(100);

    ob1[3] = 10;
    i = ob1[3];
    cout << i << "\n";

    ob2 = ob1;

    i = ob2[3];
    cout << i << "\n";

    // Выполнение следующей инструкции ведет к ошибке
    ob1 = ob3; // присваивание массивов разных размеров
    return 0;
}
```

Проверка усвоения материала главы 6

1. // Перегрузка операторов << и >>

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

class coord {
    int x, y; // значения координат
public:
    coord() { x = 0; y = 0; }
    coord(int i, int j) { x = i; y = j; }
    void get_xy(int &i, int &j) { i = x; j = y; }
    coord operator<<(int i) ;
    coord operator>>(int i) ;
};

// Перегрузка оператора <<
coord coord::operator<<(int i)
{
    coord temp;

    temp.x = x << i;
    temp.y = y << i;

    return temp;
}

// Перегрузка оператора >>
coord coord::operator>>(int i)
{
    coord temp;

    temp.x = x >> i;
    temp.y = y >> i;

    return temp;
}

int main()
{
    coord o1(4, 4), o2;
    int x, y;

    o2 = o1 << 2; // ob << int
    o2.get_xy(x, y);
    cout << "(o1 << 2) X: " << x << ", Y: " << y << "\n";

    o2 = o1 >> 2; // ob >> int
    o2.get_xy(x, y);
    cout << "(o1 >> 2) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x = 0; y = 0; z = 0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    three_d operator+(three_d ob2);
    three_d operator-(three_d ob2);
    three_d operator++();
    three_d operator--();
};

three_d three_d::operator+(three_d ob2)
{
    three_d temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    temp.z = z + ob2.z;

    return temp;
}

three_d three_d::operator-(three_d ob2)
{
    three_d temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;
    temp.z = z - ob2.z;

    return temp;
}

three_d three_d::operator++()
{
    x++;
    y++;
    z++;

    return *this;
}
```

```
three_d three_d::operator--()
{
    x--;
    y--;
    z--;

    return *this;
}

int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3;
    int x, y, z;

    o3 = o1 + o2;
    o3.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    o3 = o1 - o2;
    o3.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    ++o1;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    --o1;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x = 0; y = 0; z = 0; }
```

```
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    three_d operator+ (three_d &ob2);
    three_d operator- (three_d&ob2) ;
    friend three_d operator++(three_d &ob) ;
    friend three_d operator-- (three_d &ob);
};

three_d three_d::operator+(three_d&ob2)
{
    three_d temp;

    temp.x = x + ob2.x;
    temp.y = y + ob2.y;
    temp.z = z + ob2.z;

    return temp;
}

three_d three_d::operator-(three_dsob2)
{
    three_d temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;
    temp.z = z - ob2.z;

    return temp;
}

three_d operator++(three_d &ob)
{
    ob.x++;
    ob.y++;
    ob.z++;

    return ob;
}

three_d operator--(three_d &ob)
{
    ob.x--;
    ob.y--;
    ob.z--;

    return ob;
}
```

```
int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3;
    int x, y, z;

    o3 = o1 + o2;
    o3.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    o3 = o1 - o2;
    o3.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    ++o1;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    --o1;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    return 0;
}
```

4. Бинарная оператор-функция — член класса получает левый операнд неявно, через указатель **this.** Дружественная бинарная оператор-функция явно получает оба операнда. У унарной оператор-функции — члена класса нет явно заданных параметров. У дружественной унарной оператор-функции есть один параметр.

5. Перегрузка оператора = может потребоваться, когда применяемого по умолчанию поразрядного копирования недостаточно. Например, вам может понадобиться объект, в котором была бы копия только части данных исходного объекта.

6. Нет.

7.
```
#include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
```

```
    {
        x = i; y= j; z = k;
    }
    three_d() { x = 0; y = 0; z = 0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    friend three_d operator+(three_d ob, int i);
    friend three_d operator+(int i, three_d ob) ;
};

three_d operator+(three_d ob, int i)
{
    three_d temp;

    temp.x = ob.x + i;
    temp.y = ob.y + i;
    temp.z = ob.z + i ;

    return temp;
}

three_d operator+(int i, three_d ob)
{
    three_d temp;

    temp.x = ob.x + i;
    temp.y = ob.y + i;
    temp.z = ob.z + i;

    return temp;
}

int main()
{
    three_d o1(10, 10, 10);
    int x, y, z;

    61 = o1 + 10;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    o1 = -20 + o1;
    o1.get(x, y, z);
    cout << " X: " << x << ", Y: " << y;
    cout << Z: " << z << "\n";

    return 0;
}
```

8.
```
#include <iostream>
using namespace std;

class three_d {
    int x, y, z;
public:
    three_d(int i, int j, int k)
    {
        x = i; y = j; z = k;
    }
    three_d() { x = 0; y = 0; z = 0; }
    void get(int &i, int &j, int &k)
    {
        i = x; j = y; k = z;
    }
    int operator==(three_d ob2) ;
    int operator!=(three_d ob2);
    int operator||(three_d ob2);
};

int three_d::operator==(three_d ob2)
{
    return x==ob2.x && y==ob2.y && z==ob2.z;
}

int three_d::operator!=(three_d ob2)
{
    return x!=ob2.x && y!=ob2.y && z!=ob2.z;
}

int three_d::operator||(three_d ob2)
{
    return x||ob2.x && y||ob2.y && z||ob2.z;
}

int main()
{
    three_d o1(10, 10, 10), o2(2, 3, 4), o3(0, 0, 0);

   if(o1==o1) cout << "o1 == o1\n";

   if(o1!=o2) cout << "o1 != o2\n";

   if(o3||o1) cout << "o1 или o3 равняется истина\n";

    return 0;
}
```

9. Оператор [] обычно перегружается для того, чтобы использовать привычный индексный синтаксис для доступа к элементам инкапсулированного в классе массива.

Проверка усвоения материала в целом

1.
```
/* В данной программе для простоты не был реализован контроль
ошибок. Однако в реально функционирующем приложении такой контроль
обязателен
*/
#include <iostream>
ttinclude<cstring>
using namespace std;

class strtype {
    char s[80];
public:
    strtype() { *s = '\0'; }
    strtype(char *p) { strcpy(s, p); }
    char *get() { return s; }
    strtype operator+(strtype s2);
    strtype operator=(strtype s2);
    int operator<(strtype s2);
    int operator>(strtype s2);
    int operator==(strtype s2);
};

strtype strtype::operator+(strtype s2)
{
    strtype temp;

    strcpy(temp.s, s);
    strcpy(temp.s, s2.s);

    return temp;
}

strtype strtype::operator=(strtype s2)
{
    strcpy(s, s2.s);

    return *this;
}

int strtype::operator<(strtype s2)
{
    return strcmp(s, s2.s) < 0;
}
```

```
int strtype::operator>(strtype s2)
{
    return strcmp(s, s2.s) > 0;
}

int strtype::operator==(strtype s2)
{
    return strcmp(s, s2.s) == 0;
}

int main()
{
    strtype o1("Привет"), o2(" Здесь"), o3;

    o3 = o1 + o2;
    cout << o3.get() << "\n";

    o3 = o1;
    if(o1==o3) cout << "o1 равно o3\n";

    if(o1>o2) cout << "o1 > o2\n";

    if(o1<o2) cout << "o1 < o2\n";

    return 0;
}
```

ГЛАВА 7

Повторение пройденного

1. Нет. Перегрузка оператора просто увеличивает количество типов данных, с которыми оператор может работать, но не влияет на его исходные свойства.
2. Да. Не нужно перегружать операторы только для встроенных типов данных C++.
3. Нет, приоритет оператора изменить нельзя. Нет, нельзя изменить и число операндов.
4.
```
#include <iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
```

```
    void set(int n[10]);
    void show();
    array operator+(array ob2);
    array operator-(array ob2);
    int operator==(array ob2);
};

array::array()
{
    int i;
    for(i = 0; i < 10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i = 0; i < 10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i = 0; i < 10; i++)
        cout << nums[i] << ' ';

    cout << "\n";
}

array array::operator+(array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++) temp.nums[i] = nums[i] + ob2.nums[i];

    return temp;
}

array array::operator-(array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++) temp.nums[i] = nums[i] - ob2.nums[i];

    return temp;
}

int array::operator==(array ob2)
```

```
{
    int i;

    for(i=0; i<10; i++) if(nums[i] != ob2.nums[i]) return 0;

    return 1;
}

int main()
{
    array o1, o2, o3;

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

    o1.set(i);
    o2.set(i);

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o3;
    o3.show();

    if(o1==o2) cout << "o1 равно o2\n";
    else cout << "o1 не равно o2\n";

    if(o1==o3) cout << "o1 равно o3\n";
    else cout << "o1 не равно o3\n";

    return 0;
}
```

5.
```
#include <iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    friend array operator+(array ob1, array ob2);
    friend array operator-(array ob1, array ob2);
    friend int operator==(array ob1, array ob2);
};

array::array()
{
    int i;
```

```
    for(i = 0; i < 10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i = 0; i < 10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;

    for(i = 0; i < 10; i++) cout << nums[i] << ' ';

    cout << "\n";
}

array operator+(array ob1, array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++) temp.nums[i] = ob1.nums[i] + ob2.nums[i];

    return temp;
}

array operator-(array ob1, array ob2)
{
    int i;
    array temp;

    for(i=0; i<10; i++) temp.nums[i] = ob1.nums[i] — ob2.nums[i];

    return temp;
}

int operator==(array ob1, array ob2)
{
    int i;

    for(i=0; i<10; i++) if(ob1.nums[i] != ob2.nums[i]) return 0;

    return 1;
}

int main()
{
    array o1, o2, o3;
```

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

    o1.set(i);
    o2.set(i);

    o3 = o1 + o2;
    o3.show();

    o3 = o1 - o3;
    o3.show();

    if(o1==o2) cout << "o1 равно o2\n";
    else cout << "o1 не равно o2\n";

    if(o1==o3) cout << "o1 равно o3\n";
    else cout << "o1 не равно o3\n";

    return 0;
}
```

6.
```
ttinclude <iostream>
using namespace std;

class array {
    int nums[10];
public:
    array();
    void set(int n[10]);
    void show();
    array operator++ ();
    friend array operator--(array &ob);
};

array::array()
{
    int i;

    for(i = 0; i < 10; i++) nums[i] = 0;
}

void array::set(int *n)
{
    int i;

    for(i = 0; i < 10; i++) nums[i] = n[i];
}

void array::show()
{
    int i;
```

```
    for(i = 0; i < 10; i++)
        cout << nums[i] << ' ';

    cout << "\n";
}

// Перегрузка унарного оператора с использованием функции-члена
array array::operator++()
{
    int i;

    for(i=0; i<10; i++) nums[i]++;

    return *this;
}

// Перегрузка унарного оператора
// с использованием дружественной функции
array operator--(array &ob)
{
    int i;

    for(i=0; i<10; i++) ob.nums[i]--;

    return ob;
}

int main()
{
    array o1, o2, o3;

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

    o1.set(i);
    o2.set(i);

    o3 = ++o1;
    o3.show();

    o3 = --o1;
    o3.show();

    return 0;
}
```

7. Нет. Оператор присваивания может перегружаться только с использованием функции-члена.

Упражнения

7.1

1. Правильными являются инструкции A и C.
2. Когда открытые члены базового класса наследуются как открытые, они становятся открытыми членами производного класса. Когда открытые члены базового класса наследуются как закрытые, они становятся закрытыми членами производного класса.

7.2

1. Когда защищенные члены базового класса наследуются как открытые, они становятся защищенными членами производного класса. Когда защищенные члены базового класса наследуются как закрытые, они становятся закрытыми членами производного класса.
2. Категория защищенности позволяет сохранить закрытость определенных членов базового класса, оставляя, тем не менее, возможность доступа к ним из производного класса.
3. Нет.

7.3

1.
```
ttinclude <iostream>
#include <cstring>
using namespace std;

class mybase {
    char str[80];
public:
    mybase(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

class myderived: public mybase {
    int len;
public:
    myderived(char *s): mybase(s) {
        len = strlen(s);
    }
    int getlen() { return len; }
    void show() { cout << get() << ' \n'; }
};
```

```
int main()
{
    myderived ob("привет");

    ob.show();
    cout << ob.getlen() << '\n';

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

// Базовый класс для автомобилей разных типов
class vehicle {
    int num_wheels;
    int range;
public:
    vehicle(int w, int r)
    {
        num_wheels = w; range = r;
    }
    void showv()
    {
        cout << "Число колес: " << num_wheels << '\n';
        cout << "Грузоподъемность: " << range << '\n';
    }
};

class car: public vehicle {
    int passengers;
public:
    car(int p, int w, int r) : vehicle(w, r)
    {
        passengers = p;
    }

    void show()
        {
            showv();
            cout << "Пассажироемкость: " << passengers << '\n';
        }
};

class truck: public vehicle {
    int loadlimit;
public:
    truck(int l, int w, int r) : vehicle(w, r)
```

```
    {
        loadlimit = 1;
    }

    void show()
        {
            showv();
            cout << "Пробег: " << loadlimit << '\n';
        }
};

int main()
{
    car c(5, 4, 500) ;
    truck t(30000, 12, 1200);

    cout << "Легковушка:\n";
    c.show();
    cout << "\nГрузовик:\n";
    t.show();

    return 0;
}
```

7.4

1.
```
Работа конструктора A
Работа конструктора B
Работа конструктора C
Работа деструктора C
Работа деструктора B
Работа деструктора A
```

2.
```
#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int a) { i = a; }
};

class B {
    int j;
public:
    B(int a) { j = a; }
};
```

```
class C: public A, public B {
    int k;
public:
    C(int c, int b, int a): A(a), B(b) {
        k = c;
    }
};
```

7.5

2. Виртуальный базовый класс нужен тогда, когда производный класс наследует два (или более) класса, каждый из которых сам унаследовал один и тот же базовый класс. Без виртуального базового класса в последнем производном классе существовало бы две (или более) копии общего базового класса. Однако благодаря тому, что исходный базовый класс делается виртуальным, в последнем производном классе представлена только одна копия базового.

Проверка усвоения материала главы 7

1.
```
#include <iostream>
using namespace std;

class building {
protected:
    int floors;
    int rooms;
    double footage;
};

class house: public building {
    int bedrooms;
    int bathrooms;
public:
    house(int f, int r, double ft, int br, int bth) {
        floors = f; rooms = r; footage = ft;
        bedrooms = br; bathrooms = bth;
    }
    void show() {
        cout << "этажей: " << floors << '\n';
        cout << "комнат: " << rooms << '\n';
        cout << "метраж: " << footage << '\n';
        cout << "спален: " << bedrooms << '\n';
        cout << "ванн: " << bathrooms << '\n';
    }
};
```

```
class office: public building {
    int phones;
    int extinguishers;
public:
    office(int f, int r, double ft, int p, int ext) {
        floors = f; rooms = r; footage = ft;
        phones = p; extinguishers = ext;
    }
    void show() {
        cout << "этажей: " << floors << '\n';
        cout << "комнат: " << rooms << '\n';
        cout << "метраж: " << footage << '\n';
        cout << "телефонов: " << phones << '\n';
        cout << "огнетушителей: " << extinguishers << '\n';
    }
};

int main()
{
    house h_ob(2, 12, 5000, 6, 4);
    office o_ob(4, 25, 12000, 30, 8);

    cout << "Жилой дом: \n";
    h_ob.show();

    cout << "\nОфис: \n";
    o_ob.show();

    return 0;
}
```

2. Если базовый класс наследуется как открытый, открытые члены базового класса становятся открытыми членами производного класса, а закрытые члены базового класса остаются закрытыми. Если базовый класс наследуется как закрытый, все члены базового класса становятся закрытыми членами производного.

3. Члены, объявленные как защищенные (**protected**), являются закрытыми членами базового класса, но могут наследоваться (и к ним можно иметь доступ) любым производным классом. При наследовании со спецификатором доступа **protected** все открытые и защищенные члены базового класса становятся защищенными членами производного класса.

4. Конструкторы вызываются в порядке наследования. Деструкторы — в обратном порядке.

5. ttinclude <iostream>
using namespace std;

```
class planet {
protected:
    double distance; // расстояние в милях от Солнца
    int revolve;     // полный оборот в днях
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth: public planet {
    double circumference; // окружность орбиты
public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2 * distance * 3.1416;
    }
    void show() {
        cout << "Расстояние от Солнца: " << distance << '\n';
        cout << "Полный оборот в днях: " << revolve << '\n';
        cout << "Окружность орбиты: " << circumference << '\n';
    }
};

int main()
{
    earth ob(93000000, 365);

    ob.show();

    return 0;
}
```

6. Для того чтобы программа стала верной, классы **motorized и road_use** должны наследовать базовый класс **vehicle** как виртуальный.

Проверка усвоения материала в целом

1. Некоторые компиляторы не допускают использования инструкции **switch** во встраиваемых функциях. Такие функции автоматически трактуются компилятором как "обычные" функции.

2. Оператор присваивания является единственным оператором, который не наследуется. Причину этого понять легко. Поскольку в производном классе появятся члены, которых нет в базовом, то при перегрузке оператора = относительно базового класса в базовом классе ничего не будет известно о членах, появившихся в производном классе, и поэтому невозможно правильно копировать эти новые члены.

ГЛАВА 8

Повторение пройденного

1.
```
#include <iostream>
using namespace std;

class airship {
protected:
    int passegers;
    double cargo;
};

class airplane: public airship {
    char engine; // p для винтового, j для реактивного
    double range;
public:
    airplane(int p, double c, char e, double r)
    {
        passegers = p;
        cargo = c;
        engine = e;
        range = r;
    }
    void show();
};

class balloon: public airship {
    char gas; // h для водорода, e для гелия
    double altitude;
public:
    balloon(int p, double c, char g, double a)
    {
        passegers = p;
        cargo = c;
        gas = g;
        altitude = a;
    }
    void show();
};

void airplane::show()
{
    cout << "Пассажировместимость: " << passegers << '\n';
    cout << "Грузоподъемность: " << cargo << '\n';
    cout << "Двигатель: ";
```

```
    if(engine=='p') cout << "Винтовой\n";
    else cout << "Реактивный\n";
    cout << "Дальность: " << range << '\n';
}

void balloon::show()
{
    cout << "Пассажировместимость: " << passegers << '\n';
    cout << "Грузоподъемность: " << cargo << '\n';
    cout << "Наполнитель: ";
    if(gas=='h') cout << "Водород\n";
    else cout << "Гелий\n";
    cout << "Потолок: " << altitude << '\n';
}

int main()
{
    balloon b(2, 500.0, 'h', 12000.0);
    airplane b727(100, 40000.0, 'j', 40000.0);

    b.show();
    cout << '\n';
    b727.show();

    return 0;
}
```

2. Спецификатор доступа **protected** оставляет члены базового класса закрытыми, но при этом позволяет получить к ним доступ из любого производного класса.

3. Выводимый на экран результат работы программы иллюстрирует очередность вызова конструкторов и деструкторов.

```
Работа конструктора A
Работа конструктора B
Работа конструктора C
Работа деструктора C
Работа деструктора B
Работа деструктора A
```

4. Порядок вызова конструкторов — ABC, порядок вызова деструкторов — CBA.

5.
```
#include <iostream>
using namespace std;

class base {
    int i, j;
public:
    base(int x, int y) {i = x; j = y; }
```

```
    void showij() { cout << i << ' ' << j << '\n'; }
};

class derived: public base {
    int k;
public:
    derived(int a, int b, int c) : base(b, c) {
        k = a;
    }
    void show() { cout << k << ' '; showij(); }
};

int main()
{
    derived ob(1, 2, 3);

    ob.show();

    return 0;
}
```

6. Пропущены слова "общего" и "специальным".

Упражнения

8.2

1.
```
ttinclude <iostream>
using namespace std;

int main()
{
    cout.setf(ios::showpos);

    cout << -10 << ' ' << 10 << '\n';

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{

    cout.setf(ios::showpoint | ios::uppercase I ios::scientific);

    cout << 100.0;
```

```
    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

int main()
{
    ios::fmtflagsf;

    f = cout.flags(); // сохранение флагов

    cout.unsetf(ios::dec);
    cout.setf(ios::showbase | ios::hex);
    cout << 100 << '\n';

    cout.flags(f); // возврат флагов в исходное состояние

    return 0;
}
```

8.3

1.
```
/* Создание таблицы натуральных и десятичных логарифмов чисел
от 2 до 100
*/
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double x;

    cout.precision(5);
    cout << "x      lg x      ln x\n\n";

    for(x = 2.0; x <= 100.0; x++) {
        cout.width(10);
        cout << x << " ";
        cout.width(10);
        cout << log10(x) << " ";
        cout.width(10);
        cout << log(x) << "\n";
    }

    return 0;
}
```

2.
```
tinclude <iostream>
tinclude <cstring>
using namespace std;

void center(char *s) ;

int main()
{
    center("Это здесь!");
    center("Мне нравится C++.");

    return 0;
}

void center(char *s)
{
    int len;

    len = 40 + (strlen(s)/2);

    cout.width(len);
    cout << s << "\n";
}
```

8.4

1a.
```
// Таблица натуральных и десятичных логарифмов чисел от 2 до 100
tinclude <iostream>
tinclude <iomanip>
tinclude <cmath>
using namespace std;

int main()
{
    double x;

    cout << setprecision(5);
    cout << "x      lg x       ln x\n\n";

    for(x = 2.0; x <= 100.0; x++) {
        cout << setw(10) << x << " ";
        cout << setw(10) << log10(x) << " ";
        cout << setw(10) << log(x) << "\n";
    }

    return 0;
}
```

1b.
```
#include <iostream>
#include <iomanip>
```

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

void center(char *s);

int main()
{
    center("Это здесь!");
    center("Мне нравится C++.");

    return 0;
}

void center(char *s)
{
    int len;

    len = 40 + (strlen(s)/2);

    cout << setw(len) << s << "\n";
}
```

2. `cout << setiosflags(ios::showbase | ios::hex) << 100;`

3. Установка флага **boolalpha** для потока вывода приводит к тому, что значения булева типа выводятся на экран в виде слова **true** или **false.** Установка флага **boolalpha** для потока ввода позволяет вводить значения булева типа с помощью слова **true или false.**

8.5

1.
```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char*ptr);
    ~strtype() {delete [] p; }
    friend ostream &operator<<(ostream &stream, strtype &ob);
};

strtype::strtype(char*ptr)
{
    len = strlen(ptr) + 1;
```

```
    p = new char [len];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr) ;
}

ostream &operator<<(ostream &stream, strtype &ob)
{
    stream << ob.p;

    return stream;
}

int main()
{
    strtype s1("Это проверка"), s2("Мне нравится C++");

    cout << s1;
    cout << endl << s2 << endl;

    return 0;
}
```

2.
```
ttinclude <iostream>
using namespace std;

class planet {
protected:
    double distance; // расстояние в милях от Солнца
    int revolve;     // полный оборот в днях
public:
    planet(double d, int r) { distance = d; revolve = r; }
};

class earth: public planet {
    double circumference; // окружность орбиты
public:
    earth(double d, int r) : planet(d, r) {
        circumference = 2 * distance * 3.1416;
    }

    friend ostream &operator<<(ostream &stream, earth &ob) ;
};

ostream &operator<<(ostream &stream, earth Sob)
{
    stream << "Расстояние от Солнца: " << ob.distance << '\n';
```

```
    stream << "Оборот вокруг Солнца: " << ob.revolve << '\n';
    stream << "Окружность орбиты: " << ob.circumference;
    stream << '\n';

    return stream;
}

int main()
{
    earth ob(93000000, 365);

    cout << ob;

    return 0;
}
```

3. Пользовательская функция вывода не может быть функцией-членом потому, что объект, который вызывает функцию, не является объектом определенного пользователем класса.

8.6

1.
```
finclude <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
    char *p;
    int len;
public:
    strtype(char *ptr);
    ~strtype() { delete [] p; }
    friend ostream &operator<<(ostream &stream, strtype &ob);
    friend istream &operator>>(istream &stream, strtype &ob);
};

strtype::strtype(char *ptr)
{
    len = strlen(ptr)+1;
    p = new char [len];
    if(!p) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    strcpy(p, ptr);
}
```

```
ostream &operator<< (ostream &stream, strtype &ob)
{
    stream << ob.p;

    return stream;
}

istream &operator>> (istream &stream, strtype &ob)
{
    char temp[255];

    stream >> temp;

    if (strlen(temp) >= ob.len) {
        delete [] ob.p;
        ob.len = strlen(temp) + 1;
        ob.p = new char [ob.len];
        if(!ob.p) {
            cout << "Ошибка выделения памяти\n";
            exit(1);
        }
    }
    strcpy(ob.p, temp);

    return stream;
}

int main()
{
    strtype s1 ("Это проверка"), s2("Мне нравится C++");

    cout << s1;
    cout << '\n' << s2;

    cout << "\nВведите строку: ";
    cin >> s1;
    cout << s1;

    return 0;
}
```

2.
```
ttinclude <iostream>
using namespace std;

class factor {
    int num;    // число
    int lfact; // наименьший делитель
public:
    factor(int i);
```

```
    friend ostream &operator<<(ostream &stream, factor ob);
    friend istream &operator>>(istream &stream, factor &ob);
};

factor::factor(int i)
{
    int n;

    num = i;

    for(n=2; n<(i/2); n++)
        if(!(i%n)) break;

    if(n<(i/2)) lfact = n;
    else lfact = 1;
}

istream &operator>>(istream &stream, factor &ob)
{
    stream >> ob.num;

    int n;

    for(n=2;n<(ob.num/2); n++)
        if(!(ob.num%n)) break;

    if(n<(ob.num/2)) ob.lfact = n;
    else ob.lfact = 1;

    return stream;
}

ostream &operator<<(ostream &stream, factor ob)
{
    stream << ob.lfact << " Это наименьший делитель числа ";
    stream << ob.num << '\n';

    return stream;
}

int main()
{
    factor o(32);

    cout << o;

    cin >> o;
    cout << o;

    return 0;
}
```

Проверка усвоения материала главы 8

1.
```
#include <iostream>
using namespace std;

int main()
{
    cout << 100 << ' ' ;

    cout.unsetf(ios::dec); // сброс флага dec
    cout.setf(ios::hex);
    cout << 100 << ' ' ;

    cout.unsetf(ios::hex); // сброс флага hex
    cout.setf(ios::oct);
    cout << 100 << '\n';

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
    cout.setf(ios::left);
    cout.precision(2);
    cout.fill('*');
    cout.width(20);

    cout << 1000.5354 << '\n';

    return 0;
}
```

3а.
```
ttinclude <iostream>
using namespace std;

int main()
{
    cout << 100 << ' ';

    cout << hex << 100 << ' ';

    cout << oct << 100 << '\n';

    return 0;
}
```

3b.
```
ttinclude <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setiosflags(ios::left);
    cout << setprecision(2);
    cout << setfill('*');
    cout << setw(20);

    cout << 1000.5354 << '\n';

    return 0;
}
```

4.
```
ios::fmtflagsf;
f = cout.flags(); // сохранение

// ...

cout.flags(); // отображение
```

5.
```
finclude <iostream>
using namespace std;

class pwr {
    int base;
    int exponent;
    double result; // результат возведения в степень
public:
    pwr(int b, int e);
    friend ostream &operator<<(ostream &stream, pwr ob);
    friend istream &operator>>(istream &stream, pwr &ob);
};

pwr::pwr(int b, int e)
{
    base = b;
    exponent = e;

    result = 1;
    for(; e; e--) result = result * base;
}

ostream &operator<<(ostream &stream, pwr ob)
{
    stream << ob.base << " в степени " << ob.exponent;
    stream << " равно " << ob.result << '\n';
```

```
    return stream;
}

istream &operator>>(istream &stream, pwr &ob)
{
    int b, e;

    cout << "Введите основание и показатель степени: ";
    stream >> b >> e;

    pwr temp(b, e);

    ob = temp;

    return stream;
}

int main()
{
    pwr ob(10, 2);

    cout << ob;

    cin>>ob;
    cout << ob;

    return 0;
}
```

6.
```
// Эта программа рисует квадраты
#include <iostream>
using namespace std;

class box {
    int len;
public:
    box(int l) { len = l; }
    friend ostream &operator<<(ostream &stream, box ob);
};

// рисование квадрата
ostream &operator<<(ostream &stream, box ob)
{
    int i, j;

    for(i=0; i<ob.len; i++) stream << '*';
    stream << '\n';
    for(i=0; i<ob.len-2; i++) {
        stream<<'*';
        for(j=0; j<ob.len-2; j++) stream << ' ';
```

```
            stream << "*\n";
        }
        for(i=0; i<ob.len; i++) stream << '*';
        stream<< ' \n';

        return stream;
}

int main()
{
    box b1(4), b2(7);

    cout << b1 << endl << b2;

    return 0;
}
```

Проверка усвоения материала в целом

1.
```
#include <iostream>
using namespace std;

#define SIZE 10

// Объявление класса stack для символов
class stack {
    char stck[SIZE]; // содержит стек
    int tos; // индекс вершины стека
public:
    stack();
    void push(char ch); // помещает в стек символ
    char pop(); // выталкивает из стека символ
    friend ostream &operator<<(ostream &stream, stack ob);
};

// Инициализация стека
stack::stack()
{
    tos=0;
}

// Помещение символа в стек
void stack::push(char ch)
{
    if (tos==SIZE) {
        cout << "Стек полон";
        return;
    }
```

```
    stck[tos]=ch;
    tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
    if (tos==0) {
        cout << "Стек пуст";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}

ostream &operator<<(ostream &stream, stack ob)
{
    char ch;

    while(ch=ob.pop()) stream << ch;
    stream << endl;

    return stream;
}

int main() .
{
    stack s;

    s.push('a');
    s.push('b');
    s.push('c');

    cout << s;
    cout << s;

    return 0;
}
```

2.
```
#include <iostream>
ttinclude<ctime>
using namespace std;

class watch {
    time_t t;
public:
    watch() { t = time(NULL); }
    friend ostream &operator<<(ostream &stream, watch ob);
};
```

```
ostream &operator<<(ostream &stream, watch ob)
{
    struct tm *localt;

    localt = localtime(&ob.t);
    stream << asctime(localt) << endl;

    return stream;
}

int main()
{
    watch w;

    cout << w;

    return 0;
}
```

3.
```
ttinclude <iostream>
using namespace std;

class ft_to_inches {
    double feet;
    double inches;
public:
    void set(double f) {
        feet = f;
        inches = f * 12;
    }
   friend istream &operator>>(istream &stream, ft_to_inches &ob);
   friend ostream &operator<<(ostream &stream, ft_to_inches ob);
   };
istream &operator>>(istream &stream, ft_to_inches &ob)
{
    double f;

    cout << "Введите число футов: ";
    stream >> f;
    ob.set(f);

    return stream;
}

ostream &operator<<(ostream &stream, ft_to_inches ob)
{
    stream << ob.feet << " футов равно " << ob.inches;
    stream << " дюймам\n";
```

```
    return stream;
}

int main()
{
    ft_to_inches x;
    cin>>x;
    cout << x;
    return 0;
}
```

ГЛАВА 9

Повторение пройденного

1.
```
#include <iostream>
using namespace std;

int main()
{
    cout.width(40);
    cout.fill(':');
    cout << "C++ прекрасен" << '\n';
    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int main()
{
    cout.precision(4);
    cout << 10.0/3.0 << '\n';
    return 0;
}
```

3.
```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setprecision(4) << 10.0/3.0 << '\n';
```

```
    return 0;
}
```

4. Пользовательская функция вывода — это перегруженная оператор-функция **operator<<()**, которая передает данные класса в поток вывода. Пользовательская функция ввода — это перегруженная оператор-функция **operator>>()**, которая принимает данные класса из потока ввода.

5.
```
#include <iostream>
using namespace std;

class date {
    char d[9]; // дата хранится в виде строки: mm/dd/yy
public:
    friend ostream &operator<<(ostream &stream, date ob);
    friend istream &operator>>(istream &stream, date &ob);
};

ostream &operator<<(ostream &stream, date ob)
{
    stream << ob.d << '\n';
    return stream;
}

istream &operator>>(istream &stream, date &ob)
{
    cout << "Введите дату (mm/dd/yy): ";
    stream >> ob.d;
    return stream;
}

int main()
{
    date ob;
    cin>>ob;
    cout << ob;
    return 0;
}
```

6. Для использования манипуляторов с параметрами необходимо включить в программу заголовок **<iomanip>**.

7. Встроенными потоками являются потоки:

cin
cout
cerr
clog

Упражнения

9.1

1.
```
// Представление времени и даты
ttinclude<iostream>
#include <ctime>
using namespace std;

// Манипулятор вывода времени и даты
ostream &td (ostream &stream)
{
    struct tm *localt;
    time_t t;

    t = time(NULL);
    localt = localtime (&t);
    stream <<asctime (localt) << endl;

    return stream;
}

intmain ()
{
    cout << td << '\n';

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

// Установка шестнадцатеричного вывода с символом X
// в верхнем регистре
ostream &sethex (ostream &stream)
{
    stream.unsetf (ios::dec | ios::oct);
    stream.setf (ios::hex| ios::uppercase | ios: :showbase);

    return stream;
}

// Сброс флагов
ostream &reset (ostream &stream)
{
    stream.unsetf (ios::hex | ios::uppercase | ios: : showbase);
    stream.setf (ios::dec);

    return stream;
}
```

```
int main ()
{
    cout << sethex << 100 << '\n';
    cout << reset << 100 << '\n';

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

// Пропуск 10 символов
istream &skipchar (istream &stream)
{
    int i;
    char c;

    for(i=0; i<10; i++) stream >> c;

    return stream;
}

int main()
{
    char str[80];

    cout << "Введите несколько символов: ";
    cin >> skipchar >> str;

    cout << str << '\n';

    return 0;
}
```

9.2

1.
```
// Копирование файла и вывод числа скопированных символов
#include <iostream>
ttinclude<fstream>
using namespace std;

int main(int argc, char *argv[ ])
{
    if(argc!=3) {
        cout << "Копирование <файл_ввода> <файл_вывода>\n";
        return 1;
    }

    ifstream fin(argv[1]) ;  // открытие файла для ввода
    ofstream fout(argv[2]) ; // создание файла для вывода
```

```
    if(!fin) {
        cout << "Файл для ввода открыть невозможно\n";
        return 1;
    }

    if(!fout) {
        cout << "Файл для вывода открыть невозможно\n";
        return 1;
    }

    char ch;
    unsigned count = 0;

    fin.unsetf(ios::skipws); // не пропускать пробелы
    while(!fin.eof()) {
        fin >>ch;
        if(!fin.eof()) {
            fout << ch;
            count++;
        }
    }

    cout << "Число скопированных байтов: " << count << '\n';

    fin.close();
    fout.close();

    return 0;
}
```

Результат, выводимый этой программой, может отличаться от того, который выводится в каталоге, поскольку некоторые символы могут интерпретироваться иначе. Уточним, когда считывается последовательность *возврат каретки/перевод строки,* она преобразуется в символ новой строки. При выводе новая строка считается одним символом, но опять преобразуется в последовательность символов *возврат каретки/перевод строки.*

2.
```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream pout("phone");

    if(!pout) {
        cout << "Файл PHONE открыть невозможно\n";
        return 1;
    }
```

```
    pout << "Иссак Ньютон, 415 555-3423\n";
    pout << "Роберт Годдард, 213 555-2312\n";
    pout << "Энрико Ферми, 202 555-1111\n";

    pout.close();

    return 0;
}
```

3.
```
// Подсчет числа слов
#include <iostream>
ttinclude<fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Подсчет слов: <файл_ввода>\n";
        return 1;
    }

    ifstream in(argv[ 1 ]);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    int count = 0;
    char ch;

    in >> ch; // нахождение первого символа — не пробела

    // Теперь пробелы пропускать нельзя
    in.unsetf(ios: : skipws) ; // не пропускать пробелы

    while (!in.eof()) {
        in>>ch;
        if(isspace(ch)) {
            count++;
            while (isspace(ch) && !in.eof()) in >> ch;
        }
    }

    cout << "Число слов: " << count << '\n';

    in.close();

    return 0;
}
```

(Если между словами имеется более одного пробела, то общее число слов подсчитывается программой неправильно. Кроме этого необходимо, чтобы последним символом в файле был пробел. — *Примеч. пер.)*

4. Функция **is_open** возвращает истину, если вызывающий поток связан с открытым файлом.

9.3

1a.
```
// Копирование файла и вывод числа скопированных символов
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Копирование: <файл_ввода> <файл_вывода>\n";
        return 1;
    }

ifstream fin(argv[1], ios::in | ios::binary) ;   // открытие
                                                 // файла ввода
ofstream fout(argv[2], ios::out | ios::binary); // создание
                                                // файла вывода

    if(!fin) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    if(!fout) {
        cout << "Файл вывода открыть невозможно\n";
        return 1;
    }

    char ch;
    unsigned count = 0;

    while(!fin.eof()) {
        fin.get(ch);
        if(!fin.eof())
            fout.put(ch);
            count++;
        }
    }

    cout << "Число скопированных байтов: " << count << '\n';

    fin.close();
    fout.close();
```

```
    return 0;
}
```

1b.
```
// Подсчет числа слов
#include <iostream>
#include <fstream>
ttinclude<cctype>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Подсчет: <файл_ввода>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
       , return 1;
    }

    int count = 0;
    char ch;

    // нахождение первого символа — не пробела
    do {
        in.get(ch);
    ) while(isspace(ch));

    while(!in.eof ()) {
        in.get(ch);
        if(isspace(ch)) {
            count++;
            // Поиск следующего слова
            while (isspace(ch) && !in.eof()) in.get(ch);
        }
    }

    cout << "Число слов: " << count << ' \n ' ;

    in.close();

    return 0;
}
```

2.
```
// Вывод содержимого класса account в файл с помощью
// пользовательской функции вывода
#include <iostream>
```

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

class account {
    int custnum;
    char name[80];
    double balance;
public:
    account(int c, char *n, double b)
    {
        custnum = c;
        strcpy(name, n);
        balance = b;
    }
    friend ostream &operator<<(ostream &stream, account ob);
};

ostream &operator<<(ostream &stream, account ob)
{
    stream << ob.custnum << ' ';
    stream << ob.name << ' ' << ob.balance;
    stream << '\n';

    return stream;
}

int main()
{
    account Rex(1011, "Ralph Rex", 12323.34);
    ofstream out("accounts", ios::out | ios::binary);

    if(!out) {
        cout << "Файл вывода открыть невозможно\n";
        return 1;
    }

    out << Rex;

    out.close();

    return 0;
}
```

9.4

1. // Использование функции get() для считывания строки с пробелами

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

```
int main()
{
    char str[80];

    cout << "Введите Ваше имя: ";
    cin.get(str, 79);

    cout << str << '\n';

    return 0;
}
```

Программа работает одинаково, независимо от того, какая функция используется — **get()** или **getline()**.

2. // Использование функции getline() для вывода файла на экран

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Считывание: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    char str[255];

    while(!in.eof()) {
        in.getline(str, 254);
        cout << str << '\n';
    }

    in.close();

    return 0;
}
```

9.5

1. // Вывод содержимого файла на экран в обратном порядке

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

```
int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Реверс: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    char ch;
    long i;

    // переход в конец файла (до символа eof)
    in.seekg(0, ios::end);
    i = (long) in.tellg(); // выяснение количества байтов в файле
    i -= 2;

    for(; i>=0; i--) {
        in.seekg(i, ios::beg);
        in.get(ch);
        cout << ch;
    }

    in.close();

    return 0;
}
```

2.
```
// Перестановка местами символов в файле
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Перестановка: <имя_файла>\n";
        return 1;
    }

    // Открытие файла для ввода/вывода
    fstream io(argv[ 1 ], ios::in | ios::out| ios::binary);
    if(!io) {
        cout << "Файл открыть невозможно\n";
```

```
        return 1;
    }

    char ch1, ch2;
    long i;

    for(i=0; !io.eof(); i+=2) {
        io.seekg(i, ios::beg);
        io.get(ch1);
        if(io.eof()) continue;
        io.get(ch2);
        if(io.eof()) continue;
        io.seekg(i, ios::beg);
        io.put(ch2);
        io.put(ch1);
    }

    io.close();

    return 0;
}
```

9.6

1a. /* Вывод содержимого файла на экран в обратном порядке с контролем ошибок
*/

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

int main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Реверс: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    char ch;
    long i;

    // Переход в конец файла (до символа eof)
    in.seekg(0, ios::end);
```

```
    if(!in.good()) return 1;
    i = (long) in.tellg(); // выяснение количества байтов в файле
    if(!in.good()) return 1;
    i -= 2;

    for(; i>=0; i--) {
        in.seekg(i, ios::beg);
        if(!in.good()) return 1;
        in.get(ch);
        if(!in.good()) return 1;
        cout << ch;
    }

    in.close();
    if(!in.good()) return 1;

    return 0;
}
```

1b.

```
// Перестановка местами символов в файле с контролем ошибок
ttinclude<iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
'{
    if(argc!=2) {
        cout << "Перестановка: <имя_файла>\n";
        return 1;
    }

    // Открытие файла для ввода/вывода
    fstream io(argv[ 1 ], ios::in | ios: :out | ios: :binary);

    if(!io) {
        cout << "Файл открыть невозможно\n";
        return 1;
    }

    char ch1, ch2;
    long i;

    for(i=0; !io.eof(); i+=2) {
        io.seekg(i, ios::beg);
        if(!io.good()) return 1;
        io.get(ch1);
        if(io.eof()) continue;
        io.get(ch2);
        if(!io.good()) return 1;
```

```
            if(io.eof()) continue;
            io.seekg(i, ios::beg);
            if(!io.good()) return 1;
            io.put(ch2);
            if(!io.good()) return 1;
            io.put(ch1);
            if(!io.good()) return 1;
        }

        io.close();
        if(!io.good()) return 1;

        return 0;
    }
```

Проверка усвоения материала главы 9

1.
```
ttinclude <iostream>
using namespace std;

ostream &tabs(ostream &stream)
{
    stream << '\t' << '\t' << '\t';
    stream.width(20);

    return stream;
}

int main()
{
    cout << tabs << "Проверка\n";

    return 0;
}
```

2.
```
#include <iostream>
#include <cctype>
using namespace std;

istream &findalpha(istream &stream)
{
    char ch;

    do {
        stream.get(ch);
    } while(!isalpha(ch));
    return stream;
}
```

```
int main()
{
    char str[80];

    cin >> findalpha >> str;
    cout << str << "\n";

    return 0;
}
```

3. // Копирование файла и смена регистра букв

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

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Копирование <источник> <целевой_файл>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    ofstream out(argv[2]);

    if(!out) {
        cout << "Файл вывода открыть невозможно\n";
        return 1;
    }

    while(!in.eof()) {
        ch = in.get();
        if(!in.eof()) {
            if(islower(ch)) ch = toupper(ch);
            else ch = tolower(ch);
            out.put(ch);
        }
    }

    in.close();
    out.close();
```

```
    return 0;
}
```

4.
```
// Подсчет букв
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int alpha[26];

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Подсчет: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    // Инициализация массива alpha[]
    int i;
    for(i=0; i<26; i++) alpha[i] = 0;

    while(!in.eof()) {
        ch = in.get();
        if(isalpha(ch)) { // если найдена буква, считаем ее
            ch = toupper(ch);
            alpha[ch- 'A']++; // 'A' -- 'A' == 0,
                              // 'B' - 'A' == 1, и т. д.
        }
    };

    // Вывод на экран результата
    for(i=0; i<26; i++) {
        cout << (char) ('A' + i) << ": " << alpha[i] << '\n';
    }

    in.close();

    return 0;
}
```

5a.
```
// Копирование файла и смена регистра букв с контролем ошибок
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Копирование <откуда> <куда>\n";
        return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    ofstream out(argv[2]);
    if(!out) {
        cout << " Файл вывода открыть невозможно\n";
        return 1;
    }

    while(!in.eof()) {
        ch = in.get();
        if(!in.good() && !in.eof()) return 1;
        if(!in.eof()) {
            if(islower(ch)) ch = toupper(ch);
            else ch = tolower(ch);
            out.put(ch);
            if(!out.good()) return 1;
        }
    };

    in.close();
    out.close();
    if(!in.good() && !out.good()) return 1;

    return 0;
}
```

5b.
```
// Подсчет букв с контролем ошибок
#include <iostream>
#include <fstream>
```

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

int alpha[26];

int main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Подсчет: <имя_файла>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    // Инициализация массива alpha[]
    int i;
    for(i=0; i<26; i++) alpha[i] = 0;

    while(!in.eof()) {
        ch = in.get();
        if( !in.good() && !in.eof()) return 1;
        if(isalpha(ch)) { // если найдена буква, подсчитаем ее
            ch = toupper(ch);
            alpha[ch - 'A']++; // 'A' - 'A' == 0,
                               // 'B' - 'A' == 1, и т. д.
        }
    };

    // Вывод на экран результата
    for(i=0; i<26; i++) {
        cout << (char) ('A' + i) << ": " << alpha[i] << '\n';
    }

    in.close();
    if(!in.good()) return 1;

    return 0;
}
```

6. Для установки указателя чтения (get) используется функция seekg(). Для установки указателя записи (put) используется функция **seekp()**.

Проверка усвоения материала в целом

1.
```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

#define SIZE 40

class inventory {
    char item[SIZE]; // название предмета
    int onhand;  // количество экземпляров, выданных на руки
    double cost; // цена экземпляра
public:
    inventory(char *i, int o, double c)
    {
        strcpy(item, i);
        onhand = o;
        cost = c;
    }
    void store(fstream &stream) ;
    void retrieve(fstream &stream);
    friend ostream &operator<<(ostream &stream, inventory ob);
    friend istream &operator>>(istream &stream, inventory &ob);
};

ostream &operator<<(ostream &stream, inventory ob)
{
    stream << ob.item << ": " << ob.onhand;
    stream << " $" << ob.cost << '\n';

    return stream;
}

istream &operator>>(istream Sstream, inventory &ob)
{
    cout << "Введите название предмета: ";
    stream >> ob.item;
    cout << "Введите число выданных экземпляров: ";
    stream >> ob.onhand;
    cout << "Введите стоимость экземпляра: ";
    stream >> ob.cost;

    return stream;
}

void inventory::store(fstream &stream)
{
    stream.write(item, SIZE);
```

```
    stream.write ( (char*) &onhand, sizeof(int));
    stream.write ( (char*) &cost, sizeof(double));
}

void inventory::retrieve(fstream &stream)
{
    stream.read(item, SIZE);
    stream.read ( (char*} &onhand, sizeof(int));
    stream.read ( (char*) &cost, sizeof(double));
}

int main()
{
    fstream inv("inv", ios::out | ios::binary);
    int i;

    inventory pliers ("плоскогубцы", 12, 4.95);
    inventory hammers ("молотки", 5, 9.45);
    inventory wrenches ("ключи", 22, 13.90);
    inventory temp("", 0, 0.0);

    if(!inv) {
        cout << "Файл для вывода открыть невозможно\n";
        return 1;
    }
    // Запись в файл
    pliers.store(inv);
    hammers.store(inv);
    wrenches.store(inv);

    inv.close();

    // Открытие файла для ввода
    inv.open("inv", ios::in | ios::binary);

    if(!inv) {
        cout << "Файл для ввода открыть невозможно\n";
        return 1;
    }

    do {
        cout << "Запись # (для выхода введите -1) : ";
        cin>>i;
        if(i == -1) break;
        inv.seekg(i*(SIZE+sizeof(int)+sizeof(double)) , ios::beg);
        temp.retrieve(inv);
        cout << temp;
    } while (inv.good());
```

```
    inv.close();

    return 0;
}
```

ГЛАВА 10

Повторение пройденного

1.
```
ttinclude <iostream>
using namespace std;

ostream &setsci (ostream &stream)
{
    stream.setf(ios::scientific I ios::uppercase);

    return stream;
}

int main()
{
    double f = 123.23;

    cout << setsci << f;
    cout << '\n';

    return 0;
}
```

2.
```
// Копирование файла и преобразование табуляций в пробелы
#include <iostream>
#include <fstream>
using namespace std;

int main (int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Копирование: <ввод> <вывод>\n";
        return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }
```

```
    ofstream out(argv[2]);
    if(!out) {
        cout << "Файл вывода открыть невозможно\n";
        return 1;
    }

    char ch;
    int i = 8;

    while(!in.eof()) {
        in.get(ch);
        if(ch=='\t') for(; i>0; i --) out.put(' ');
        else out.put(ch);
        if(i==-1 || ch=='\n') i = 8;
        i --;
    }

    in.close();
    out.close();

    return 0;
}
```

3.
```
// Поиск слова в файле
tfinclude<iostream>
#include <fstream>
#include <cstring>
using namespace std;

int main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Поиск: <файл> <слово>\n";
        return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
        cout << "Файл ввода открыть невозможно\n";
        return 1;
    }

    char str[255];
    int count = 0;

    while(!in.eof()) Ч
        in >> str;
        if( !strcmp(str,argv[2]) ) count++;
    }
```

```
    cout << argv[2] << " найдено " << count;
    cout << " раз. \n";

    in.close();

    return 0;
}
```

4. Искомой инструкцией является следующая:

```
out.seekp(234, ios::beg);
```

5. Такими функциями являются rdstate(), **good(), eof(), fail() и bad().**

6. Ввод/вывод C++ обеспечивает возможность работы с создаваемыми вами классами.

Упражнения

10.2

1.
```
#include <iostream>
using namespace std;

class num {
public:
    int i;
    num(int x) {i = x; }
    virtual void shownum() { cout << i << '\n'; }
};

class outhex: public num {
public:
    outhex(int n) : num(n) {}
    void shownum() { cout << hex << i << '\n'; }
};

class outoct: public num {
public:
    outoct (int n) : num(n) { }
    void shownum() { cout << oct << i << '\n'; }
};

int main()
{
    outoct o(10);
    outhex h(20);

    o.shownum();
    h.shownum();
```

```
    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

class dist {
public:
    double d;
    dist(double f) {d = f; }
    virtual void trav_time()
    {
        cout << "Время движения со скоростью 60 миль/час: " ;
        cout << d / 60 << '\n';
    }
};

class metric: public dist {
public:
    metric(doublef) : dist(f) {}
    void trav_time()
    {
        cout << "Время движения со скоростью 100 км/час:";
        cout << d / 100 << '\n';
    }
};

int main()
{
    dist *p, mph(88.0);
    metric kph(88);

    p = &mph;
    p->trav_time();

    p = &kph;
    p -> trav_time();

    return 0;
}
```

10.3

2. По определению в абстрактном классе содержится, по крайней мере, одна чистая виртуальная функция. Это означает, что в классе отсутствует тело этой функции. Таким образом, создание объекта абстрактного класса невозможно, поскольку определение класса неполно.

3. При вызове функции **func()** относительно класса **derived1** используется функция **func()** из класса **base.** Так происходит потому, что виртуальные функции имеют иерархическую структуру.

10.4

1.
```
// Демонстрация возможностей виртуальных функций
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
    list *head; // указатель на начало списка
    list *tail; // указатель на конец списка
    list *next; // указатель на следующий элемент списка
    int num;    // число для хранения

    list () { head = tail = next = NULL; }
    virtual void store(int i) = 0;
    virtual int retrieve() = 0;
};

// Создание списка типа очередь
class queue: public list {
public:
    void store(int i);
    int retrieve();
};

void queue::store(int i)
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;

    // Добавление элемента в конец списка
    if(tail) tail -> next = item;
    tail = item;
    item -> next = NULL;
    if(!head) head = tail;
}
```

```
int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }

    // Удаление элемента из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;

    return i;
}

// Создание списка типа стек
class stack: public list {
public:
    void store(int i);
    int retrieve();
};

void stack::store(int i)
{
    list *item;

    item = new stack;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;

    // Внесение элемента в начало списка
    if(head) item -> next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
```

```
        return 0;
    }

    // Удаление элемента из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;

    return i;
}

// Создание отсортированного списка
class sorted: public list {
public:
    void store(int i);
    int retrieve();
};

void sorted::store(int i)
{
    list *item;
    list *p, *p2;

    item = new sorted;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;

    // Поиск места внесения следующего элемента списка
    p = head;
    p2 = NULL;
    while(p) { // идем в середину
        if(p->num> i) {
            item -> next = p;
            if(p2) p2 -> next = item; // не первый элемент
            if(p==head) head = item;  // новый первый элемент
            break;
        }
        p2 = p;
        p = p -> next;
    }
    if(!p) { // идем в конец
        if(tail) tail -> next = item;
        tail = item;
        item -> next = NULL;
    }
```

```
    if(!head) // первый элемент
        head = item;
}

int sorted::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }

    // Удаление элемента из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;

    return i;
}

int main()
{
    list *p;

    // Демонстрация очереди
    queue q_ob;
    p = &q_ob; // указывает на очередь

    p -> store(1);
    p -> store(2);
    p -> store(3);

    cout << "Очередь: ";
    cout << p -> retrieve();
    cout << p -> retrieve();
    cout << p -> retrieve();

    cout << '\n';

    // Демонстрация стека
    stack s_ob;
    p = &s_ob; // указывает на стек

    p -> store(1);
    p -> store(2);
    p -> store(3);

    cout << "Стек: ";
    cout << p -> retrieve();
```

```
    cout << p -> retrieve();
    cout << p -> retrieve();

    cout << '\n';

    // Демонстрация отсортированного списка
    sorted sorted_ob;
    p = &sorted_ob;

    p -> store(4);
    p -> store(1);
    p -> store(3);
    p -> store(9);
    p -> store (5);

    cout << "Отсортированный список: ";
    cout << p -> retrieve();
    cout << p -> retrieve();
    cout << p -> retrieve();
    cout << p -> retrieve();
    cout << p -> retrieve();

    cout << '\n';

    return 0;
}
```

Проверка усвоения материала главы 10

1. Виртуальная функция — это функция, обязательно являющаяся членом класса, объявляемая в базовом классе и переопределяемая в производном от базового классе. Процесс переопределения называют подменой (overriding).
2. Виртуальными не могут быть функции, не являющиеся членами класса, а также конструкторы.
3. Виртуальные функции поддерживают динамический полиморфизм путем использования указателей базового класса. Если указатель базового класса указывает на объект производного класса, содержащего виртуальную функцию, то выбор конкретной версии вызываемой функции определяется типом объекта, на который он указывает.
4. Чистая виртуальная функция — это функция, у которой нет определения в базовом классе.
5. Абстрактный класс — это базовый класс, в котором содержится по крайней мере, одна чистая виртуальная функция. Полиморфный класс — это класс, в котором содержится, по крайней мере, одна виртуальная функция.
6. Этот фрагмент неправилен, поскольку переопределение виртуальной функции должно обеспечивать тот же тип возвращаемого значения, тот же тип

параметров и то же их количество, что и исходная функция. В данном случае, переопределенная функция f() отличается числом своих параметров.

7. Да.

Проверка усвоения материала в целом

1.
```
// Демонстрация возможностей виртуальных функций
#include <iostream>
#include <cstdlib>
using namespace std;

class list {
public:
    list *head; // указатель на начало списка
    list *tail; // указатель на конец списка
    list *next; // указатель на следующий элемент списка
    int num;    // число для хранения

    list() { head = tail = next = NULL; }
    virtual void store(int i) = 0;
    virtual int retrieve() = 0;
};

// Создание списка типа очередь
class queue: public list {
public:
    void store(int i);
    int retrieve();
    queue operator+(int i) { store(i); return *this; }
    int operator --(int unused) { return retrieve(); }
};

void queue::store(int i)
{
    list *item;

    item = new queue;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;

    // Добавление элемента в конец списка
    if(tail) tail -> next = item;
    tail = item;
    item -> next = NULL;
    if(!head) head = tail;
}
```

```
int queue::retrieve()
{
    int i;
    list *p;

    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }

  , // Удаление элемента из начала списка
    i = head -> num;
    p = head;
  . head = head -> next;
    delete p;

    return i;
}

// Создание списка типа стек
class stack: public list {
public:
    void store(int i);
    int retrieve();
    stack operator+(int i) { store(i); return *this; }
    int operator --(int unused) { return retrieve(); }
};

void stack::store(int i)
{
    list *item;

    item =' new stack;
    if(!item) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;

    // Добавление элемента в начало списка
    if(head) item -> next = head;
    head = item;
    if(!tail) tail = head;
}

int stack::retrieve()
{
    int i;
    list *p;
```

```
    if(!head) {
        cout << "Список пуст\n";
        return 0;
    }
    // Удаление элемента из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;
    return i;
}

int main()
{
    // Демонстрация очереди
    queue q_ob;

    q_ob + 1;
    q_ob + 2;
    q_ob + 3;

    cout << "Очередь: " ;
    cout << q_ob --;
    cout << q_ob --;
    cout << q_ob --;

    cout << '\n';

    // Демонстрация стека
    stack s_ob;

    s_ob + 1;
    s_ob + 2;
    s_ob + 3;

    cout << "Стек: ";
    cout << s_ob --;
    cout << s_ob --;
    cout << s_ob --;

    cout << '\n';

    return 0;
}
```

2. Отличие виртуальных функций от перегружаемых в том, что перегружаемые функции *должны* отличаться либо числом, либо типом своих параметров. Подменяемая виртуальная функция должна иметь точно такой же прототип (поэтому такой же тип возвращаемого значения, такой же тип параметров и то же их число), как и исходная функция.

ГЛАВА 11

Повторение пройденного

1. Виртуальная функция — это функция, которая в базовом классе объявляется с ключевым словом **virtual** и затем в производном классе подменяется другой функцией.
2. Чистая виртуальная функция — это функция, которая в базовом классе не имеет определения функции. Это означает, что функция обязательно должна подменяться соответствующей ей функцией в производном классе. Базовый класс, который содержит, по крайней мере, одну чистую виртуальную функцию, называется абстрактным.
3. Динамический полиморфизм достигается посредством использования виртуальных функций и указателей базового класса.
4. Если в производном классе отсутствует подмена не чистой виртуальной функции, то производный класс будет использовать версию виртуальной функции из базового класса.
5. Главным преимуществом динамического полиморфизма является гибкость. Главным его недостатком является некоторое снижение быстродействия.

Упражнения

11.1

2.
```
#include <iostream>
using namespace std;

template <class X> X min(X a, X b)
{
    if(a<=b) return a;
    else return b;
}

int main()
{
    cout << min(12.2, 2.0);
    cout << endl;
    cout << min(3, 4);
    cout << endl;
    cout << min('c', 'a');
    return 0;
}
```

3.
```
#include <iostream>
#include <cstring>
using namespace std;

template <class X> int find(X object, X *list, int size)
{
    int i;

    for(i=0; i<size; i++)
        if(object == list[i]) return i;
    return -1;
}

int main()
{
    int a[]={1, 2, 3, 4};
    char *c="это проверка";
    double d[]={1.1, 2.2, 3.3};

    cout << find(3, a, 4);
    cout << endl;
    cout << find('a', c, (int) strlen(c));
    cout << endl;
    cout << find(0.0, d, 3);

    return 0;
}
```

4. Ценность родовых функций в том, что они позволяют определить общий алгоритм, который можно применить к разным типам данных. (Следовательно, не нужно явно задавать конкретные версии алгоритма.) Кроме этого, родовые функции помогают реализовать основную идею программирования на C++, а именно "один интерфейс — множество методов".

11.2

2.
```
// Создание родовой очереди
#include <iostream>
using namespace std;

ttdefine SIZE 100

template <class Qtype> class q_type {
    Qtype queue[SIZE]; // содержит очередь
    int head, tail;    // индекс вершины и хвоста очереди
public:
    q_type() { head = tail = 0; }
    void q(Qtype num); // помещает объект в очередь
```

```
    Qtype deq();        // извлекает объект из очереди
};

// Размещение значения в очередь
template <class Qtype> void q_type<Qtype>::q(Qtype num)
{
    if (tail+1==head || (tail+1==SIZE && !head)) {
        cout << "Очередь полна\n";
        return;
}
    tail++;
    if (tail==SIZE) tail = 0; // замыкание цикла
    queue[tail] = num;
}

// Удаление значения из очереди
template <class Qtype> Qtype q_type<Qtype>::deq()
{
    if (head == tail) {
        cout << "Очередь пуста";
        return 0; // очередь пуста или какая-то иная ошибка
}
    head++;
    if (head==SIZE) head = 0;  // замыкание цикла
    return queue[head];
}

int main()
{
    q_type<int> q1;
    q_type<char> q2;

    int i;

    for(i=1; i<10; i++) {
        q1.q(i);
        q2.q(i-1+'A');
}

    for(i=1; i<10; i++) {
        cout << "Первая очередь " << q1.deq() << "\n";
        cout << "Вторая очередь " << q2.deq() << "\n";
}

    return 0;
}
```

3.
```
linclude <iostream>
using namespace std;
```

```
template <class X> class input {
    X data;
public:
    input(char *s, X min, X max);
    // ...
};

template <class X>
input<X>::input(char *s, X min, X max)
{
    do {
        cout << s << ": ";
        cin >> data;
    }while(data<min || data>max);
}

int main()
{
    input<int> i("ввод целых", 0, 10);
    input<char> c("ввод символов", 'A', 'Z');

    return 0;
}
```

11.3

2. Инструкция throw вызывается еще до того, как управление передано в блок try.
3. Вызывается символьная исключительная ситуация, а инструкция catch предназначена только для обработки исключительной ситуации типа указатель на символ. (То есть для обработки символьной исключительной ситуации нет соответствующей инструкции catch.)
4. Если возбуждается исключительная ситуация, для которой не задано соответствующей инструкции catch, то вызывается функция **terminate()**, что может привести к аварийному завершению программы.

11.4

2. Для инструкции throw нет соответствующей инструкции catch.
3. Одним из способов решить проблему является создание обработчика catch(int). Другой способ — это перехватывать все исключительные ситуации с помощью обработчика **catch(...)**.
4. Все типы исключительных ситуаций перехватываются инструкцией **catch(...)**.

5.
```
finclude <iostream>
#include <cstdlib>
using namespace std;
```

```
double divide(double a, double b)
{
    try {
        if(!b) throw(b);
    }
    catch(double) {
        cout << "На ноль делить нельзя\n";
        exit(1);
    }
    return a/b;
}

int main()
{
    cout << divide(10.0, 2.5) << endl;
    cout << divide(10.0, 0.0);

    return 0;
}
```

11.5

1. По умолчанию оператор new возбуждает исключительную ситуацию при появлении ошибки выделения памяти. Оператор **new(nothrow)** при невозможности выделить память возвращает нулевой указатель.

2.
```
p = new(nothrow) (int);
if(!p) {
        cout << "Ошибка выделения памяти\n";
        // ...
    }

try {
    p = new int;
} catch(bad_alloc ba) {
    cout << "Ошибка выделения памяти\n";
    // ...
}
```

Проверка усвоения материала главы 11

1.
```
#include <iostream>
ttinclude<cstring>
using namespace std;

// Родовая функция для поиска
// наиболее часто встречающегося значения
template <class X> X mode (X *data, int size)
```

```
{
    register int t, w;
    X md, oldmd;
    int count, oldcount;

    oldmd = 0;
    oldcount = 0;
    for(t=0; t<size; t++) {
        md=data[t];
        count = 1;
        for(w = t+1; w < size; w++)
            if(md==data[w]) count++;
        if(count > oldcount) {
            oldmd = md;
            oldcount = count;
        }
    }
    return oldmd;
}

int main()
{
    int i[] = {1, 2, 3, 4, 2, 3, 2, 2, 1, 5};
    char *p = "Это проверка";

    cout << "Значение для массива целых: " << mode(i, 10) << endl;
    cout << "Значение для массива символов: " << mode(p, (int)
    strlen(p));

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

template <class X> X sum(X *data, int size)
{
    int i;
    X result = 0;

    for(i=0; i<size; i++) result += data[i];

    return result;
}

int main()
{
    int i[] = {1, 2, 3, 4};
    double d[] = {1.1, 2.2, 3.3, 4.4};
```

```
    cout << sum(i, 4) << endl;
    cout << sum(d, 4) << endl;

    return 0;
}
```

3.
```
#include <iostream>
using namespace std;

// Родовой класс для пузырьковой сортировки
template <class X> void bubble(X *data, int size)
{
    register int a, b;
    X t;

    for(a=1; a<size; a++)
        for(b=size-1; b>=a; b--)
            if(data[b-1] > data[b]) {
                t = data[b-1];
                data[b-1] = data[b];
                data[b] = t;
            }
}

int main()
{
    int i[] = {3, 2, 5, 6, 1, 8, 9, 3, 6, 9};
    double d[] = {1.2, 5.5, 2.2, 3.3};
    int j;

    bubble(i, 10);   // сортировка данных типа int
    bubble(d, 4);    // сортировка данных типа double

    for(j=0; j<10; j++) cout << i[j] << ' ';
    cout << endl;

    for(j=0; j<4; j++) cout << d[j] << ' ';
    cout << endl;

    return 0;
}
```

4.
```
// Здесь показан родовой стек для хранения пар значений
linclude <iostream>
using namespace std;

#define SIZE 10

// Создание родового класса для стека
template <class StackType> class stack {
```

```
    StackType stck[SIZE] [2]; // содержит стек
    int tos;                  // индекс вершины стека

public:
    void init() { tos = 0;}
    void push (StackType ob, StackType ob2) ;
    StackType pop (StackType &ob2);
};

// Размещение объектов в стеке
template <class StackType>
void stack<StackType>: : push (StackType ob, StackType ob2)
{
    if (tos==SIZE) {
        cout << "Стек полон\n";
        return;
    }
    stck[tos][0] = ob;
    stck[tos] [1] = ob2;
    tos++;
}

// Выталкивание объектов из стека
template <class StackType>
StackType stack<StackType>: : pop (StackType Sob2)
{
    if (tos==0) {
        cout << "Стек пуст\n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    ob2 = stck[tos] [1];
    return stck[tos] [0];
}

int main ()
{
    // Демонстрация символьных стеков
    stack<char> s1, s2; // создание двух стеков
    int i;
    char ch;

    // Инициализация стеков
    s1.init();
    s2.init();

    s1.push('a', 'b');
    s2.push('x', 'z');
    s1.push('b', 'd');
```

```
    s2.push('y', 'e');
    s1.push('c', 'a');
    s2.push('z', 'x');

    for(i=0; i<3; i++) {
        cout << "Из стека 1:" << s1.pop(ch);
        cout << ' ' << ch << "\n";
    }

    for(i=0; i<3; i++) {
        cout << "Из стека 2:" << s2.pop(ch);
        cout << ' ' << ch << "\n";
    }

    // Демонстрация стеков со значениями типа double
    stack<double> ds1, ds2; // создание двух стеков
    double d;

    // Инициализация стеков
    ds1.init();
    ds2.init();

    ds1.push(1.1, 2.0);
    ds2.push(2.2, 3.0);
    ds1.push(3.3, 4.0);
    ds2.push(4.4, 5.0);
    ds1.push(5.5, 6.0);
    ds2.push(6.6, 7.0);

    for(i=0; i<3; i++) {
        cout << "Из стека 1:" << ds1.pop(d);
        cout << ' ' << d << "\n";
    }

    for(i=0; i<3; i++) {
        cout << "Из стека 2:" << ds2.pop(d);
        cout << ' ' << d << "\n";
    }

    return 0;
}
```

5. Ниже представлены обычные формы инструкций **try, catch и throw:**

```
try {
    // блок try
    throw исключ_ситуация
}
catch (type arg) {
    // ...
}
```

6.
```
/* В этой программе показан родовой класс stack, в который встроена
   обработка исключительных ситуаций
*/
#include <iostream>
using namespace std;

#define SIZE 10

// Создание родового класса stack
template <class StackType> class stack {
    StackType stck[SIZE]; // содержит стек
    int tos;              // индекс вершины стека

public:
    void init() { tos = 0;} // инициализация стека
    void push(StackType ch); // помещает объект в стек
    StackType pop(); // выталкивает объект из стека
};

// Размещение объекта в стеке
template <class StackType>
void stack<StackType>::push(StackType ob)
{
    try {
        if (tos==SIZE) throw SIZE;
    }
    catch(int) {
        cout << "Стек полон\n";
        return;
    }
    stck[tos] = ob;
    tos++;
}

// Выталкивание объекта из стека
template <class StackType>
StackType stack<StackType>::pop()
{
    try {
        if (tos==0) throw 0;
    }
    catch(int) {
        cout << "Стек пуст\n";
        return 0; // возврат нуля при пустом стеке
    }
    tos--;
    return stck[tos];
}
```

```
int main()
{
    // Демонстрация символьных стеков
    stack<char> s1, s2; // создание двух стеков
    int i;

    // Инициализация стеков
    s1.init();
    s2.init();

    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');

    for(i=0; i<3; i++) cout << "Из стека 1:" << s1.pop() << "\n";
    for(i=0; i<4; i++) cout << "Из стека 2:" << s2.pop() << "\n";

    // Демонстрация стеков со значениями типа double
    stack<double> ds1, ds2; // создание двух стеков

    // Инициализация стеков
    ds1.init();
    ds2.init();

    ds1.push(1.1);
    ds2.push(2.2);
    ds1.push(3.3);
    ds2.push(4.4);
    ds1.push(5.5);
    ds2.push(6.6);

   for(i=0; i<3; i++) cout << "Из стека 1:" << ds1.pop() << "\n";
   for(i=0; i<4; i++) cout << "Из стека 2:" << ds2.pop() << "\n";

    return 0;
}
```

8. Если при ошибке выделения памяти оператор **new** возбуждает исключительную ситуацию, вы можете быть уверены, что эта ошибка будет обработана тем или иным способом (даже путем аварийного завершения программы). И наоборот, если при ошибке выделения памяти оператор **new** возвращает нулевой указатель, а вы забыли организовать контроль этого возвращаемого значения, то такая ошибка может остаться незамеченной и в конечном итоге при попытке использования нулевого указателя привести к краху программы, причем будет чрезвычайно трудно обнаружить причину этого.

ГЛАВА 12

Повторение пройденного

1. В C++ родовая функция определяет общий набор операций для данных разных типов. Она реализуется с помощью ключевого слова **template.** Ниже представлена основная форма родовой функции:

```
template <class Ttype>возвр_значение имя_функции(сп_параметров)
{
    // ...
}
```

2. В C++ родовой класс определяет все операции класса, но действительные данные задаются в качестве параметра при создании объекта этого класса. Ниже представлена основная форма родового класса:

```
template <class Ttype>class имя_класса
{
    // ...
}
```

3.
```
finclude <iostream>
using namespace std;

// Возвращает значение a в степени Ь
template <class X> X gexp (X a, X b)
{
    X i, result = 1;

for(i=0; Kb; i++) result *= a;

    return result;
}

int main()
{
cout << gexp (2, 3) << endl;
cout << gexp (10.0, 2.0);

return 0;
}
```

4.
```
#include <iostream>
#include <fstream>
using namespace std;
```

```
template <class CoordType> class coord {
    CoordType x, y;
public:
    coord(CoordType i, CoordType j) { x = i; y = j ; }
    void show() { cout << x << ", " << y << endl; }
};

int main()
{
    coord<int> o1(1, 2), o2(3, 4);

    o1.show();
    o2.show();

    coord<double> o3 (0.0, 0.23), o4(10.19, 3.098);

    o3.show();
    o4.show();

    return 0;
}
```

5. Совместная работа инструкций **try, catch и throw** происходит следующим образом. Поместите все инструкции, для которых вы хотите обеспечить контроль исключительных ситуаций, внутри блока **try.** При возникновении исключительной ситуации она возбуждается с помощью инструкции **throw** и затем обрабатывается с помощью соответствующей инструкции **catch.**

6. Нет.

7. Вызов функции **terminate()** происходит в том случае, если исключительная ситуация возбуждается той инструкцией **throw,** для которой нет соответствующей инструкции **catch.** Вызов функции **unexpected()** происходит в том случае, если исключительная ситуация не указана в списке типов инструкции **throw.**

8. **catch(...)**

Упражнения

12.1

1. Динамическая идентификация типа необходима потому, что в C++ при компиляции программы не всегда есть возмозможность выяснить, на объект какого типа указывает указатель базового класса или ссылается ссылка базового класса.

2. Если класс **BaseClass** перестает быть полиморфным классом, на экране мы увидим следующее:

```
Тип переменной i — это int
Указатель p указывает на объект типа BaseClass
Указатель p указывает на объект типа BaseClass
Указатель p указывает на объект типа BaseClass
```

3. Да.

4. `if(typeid(*p) == typeid(D2)) ...`

5. Истинным. Несмотря на то, что здесь используется один и тот же класс-шаблон, тип данных в каждой из версий различен.

12.2

1. Оператор **dynamic_cast** позволяет проверить правильность выполнения операции приведения полиморфных типов.

2.
```
#include <iostream>
#include <typeinfo>
using namespace std;

class B {
    virtual void f() {}
};

class D1: public B {
    void f() {}
};

class D2: public B {
    void f() {}
};

int main()
{
    B *p;
    D2 ob;

    p = dynamic_cast<D2 *> (&ob);

    if(p) cout << "Приведение типов прошло успешно";
    else cout << "Приведение типов не произошло";

    return 0;
}
```

3.
```
int main()
{
    int i;;
    Shape *p;
```

```
    for(i=0; i<10; i++) {
        p = generator(); // получение следующего объекта

        cout << typeid(*p).name() << endl;

        // объект изображается на экране только в том случае,
        // если это не объект типа NullShape
        if(!dynamic_cast<NullShape *> (p))
            p->example();
    }

    return 0;
}
```

4. Нет. Указатели Bp и Dp указывают на объекты совершенно разных типов.

12.3

1. Новые операторы приведения типов обеспечивают более безопасные и понятные способы выполнения операции приведения типов.

2.
```
#include <iostream>
using namespace std;

void f(const double &i)
{
    double &v = const_cast<double &> (i) ;

    v = 100.0;
}

int main()
{
    double x = 98.6;

    cout << x << endl;
    f(x);
    cout << x << endl;

    return 0;
}
```

3. Поскольку оператор **const_cast** лишает объекты атрибута **const,** его использование может привести к неожиданной и нежелательной модификации объектов.

Проверка усвоения материала главы 12

1. Оператор **typeid** возвращает ссылку на объект класса **type_info**, содержащего информацию о типе.

2. Для использования оператора **typeid** в программу необходимо включить заголовок **<typeinfo>**.

3. Ниже представлены новые операторы приведения типов.

Оператор	Назначение
dynamic_cast	Выполняет операцию приведения полиморфных типов
reinterpret_cast	Преобразует указатель на один тип в указатель на другой тип
static_cast	Выполняет операцию "обычного" приведения типов
const_cast	Лишает объекты атрибута **const**

4.
```
ttinclude <iostream>
#include <typeinfo>
using namespace std;

class "A {
    virtual void f() {}
};

class B: public A {
};

class C: public B {
};

int main()
{
    A *p, a_ob;
    B b_ob;
    C c_ob;
    int i;

    cout << "Введите 0 для объектов типа A, ";
    cout << "1 для объектов типа B или";
    cout << "2 для объектов типа C.\n";

    cin>>i;

    if(i==1) p = &b_ob;
    else if(i==2) p = &c_ob;
    else p = &a_ob;
```

```
    if(typeid(*p) == typeid(A))
        cout << "Объект A";
    if(typeid(*p) == typeid(B))
        cout << "Объект B";
    if(typeid(*p) == typeid(C))
        cout << "Объект C";

    return 0;
}
```

5. Оператор **typeid** можно заменить оператором **dynamic_cast** в ситуациях, когда оператор **typeid** используется для проверки правильности выполнения операции приведения полиморфных типов.
6. Оператор **typeid** возвращает ссылку на объект типа **type_info**.

Проверка усвоения материала в целом

1. Здесь представлена версия функции **generator()**, в которой для контроля за выделением памяти используется механизм обработки исключительных ситуаций.

```
/* Использование механизма обработки исключительных ситуаций для
отслеживания ошибок выделения памяти
*/  *
Shape *generator()
{
    try {
        switch(rand() % 4) {
            case 0:
                return new Line;
            case 1:
                return new Rectangle;
            case 2:
                return new Triangle;
            case 3:
                return new NullShape;
        }
    }
    catch (bad_alloc ba) {
        return NULL;
    }
    return NULL;
}
```

2. Ниже представлена версия функции **generator()**, в которой используется оператор **new**(no**throw**).

```
// Использование оператора new(nothrow)
Shape *generator()
{
    Shape *temp;

    switch(rand() % 4) {
        case 0:
            return new(nothrow) Line;
        case 1:
            return new(nothrow) Rectangle;
        case 2:
            return new(nothrow) Triangle;
        case 3:
            return new(nothrow) NullShape;
    }

    if(temp) return temp;
    else return NULL;
}
```

ГЛАВА 13

Повторение пройденного

1. Помимо обычных для языка С операторов приведения типов, в C++ поддерживаются дополнительные операторы, которые перечислены ниже:

Оператор	Назначение
dynamic_cast	Выполняет операцию приведения полиморфных типов
reinterpret_cast	Преобразует указатель на один тип в указатель на другой тип
static_cast	Выполняет операцию "обычного" приведения типов
const_cast	Лишает объекты атрибута **const**

2. Класс **type_info** — это класс, который инкапсулирует информацию о типе данных. Ссылка на объект типа **type_info** является возвращаемым значением оператора **typeid**.

3. С помощью оператора **typeid.**

4.
```
if(typeid(Derived) == typeid(p*))
    cout << "Указатель p указывает на объект класса Derived\n";
else
    cout << "Указатель p указывает на объект класса Base\n";
```

5. Пропущено слово "производного".

6. Нет.

Упражнения

13.1

```
1. /* Программа превращения пробелов в вертикальные линии |
   без использования инструкции "using namespace std"
   */
   #include <iostream>
   #include <fstream>

   int main(int argc, char *argv[])
   {
       if(argc!=3) {
         std::cout << "Преобразование <файл_ввода> <файл_вывода>\n";
         return 1;
       }

       std::ifstream fin(argv[1]); // открытие файла для ввода
       std::ofstream fout(argv[2]); // создание файла для вывода

       if(!fout) {
           std::cout << "Файл открыть невозможно\n";
           return 1;
       }
       if(!fin) {
           std::cout << "Файл открыть невозможно\n";
           return 1;
       }

       char ch;

       fin.unsetf(std::ios::skipws); // не пропускать пробелы
       while(!fin.eof()) {
           fin>>ch;
           if(ch==' ')ch = ' | ';
           if(!fin.eof()) fout << ch;
       }

       fin.close();
       fout.close();

       return 0;
   }
```

2. Безымянное пространство имен ограничивает область видимости идентификаторов тем файлом, в котором они объявлены.
3. Представленная ниже форма инструкции **using** вводит в текущее пространство имен только указанный в инструкции член:

   using ***имя_пространства*** **::** ***член*;**

   Следующая форма инструкции **using** делает видимым все пространство имен:

   using namespace ***имя_пространства*;**

4. Поскольку вся стандартная библиотека C++, включая потоки **cin** и **cout,** объявлена в пространстве имен **std,** для совместимости большинство программ нужно вводить в пространство имен **std.** Это дает возможность использовать в программах имена стандартной библиотеки C++ непосредственно, без уточнения, к какому пространству имен они принадлежат. Для большинства программ альтернативой была бы необходимость задания всех ссылок на имена стандартной библиотеки с инструкцией **std::**. Другой альтернативой являются инструкции **using** только для потоков, т. е. **using** std::cin и **using std::cout.**
5. Размещая код библиотеки в собственном пространстве имен, вы снижаете вероятность возникновения конфликтов имен.

13.2

1.
```
// Преобразование строки в целое
#include <iostream>
#include <cstring>
using namespace std;

class strtype {
    char str[80];
    int len;
public:
    strtype(char *s) { strcpy(str, s); len = strlen(s); }
    operator char *() { return str; }
    operator int() { return len; }'
};

int main()
{
    strtype s("Функции преобразования весьма удобны");
    char *p;
    int l;

    l = s; // преобразование строки s в целое — длину строки
    p = s; // преобразование строки s
```

```
    // в char * — указатель на строку

    cout << "Строка: \n";
    cout << p << "\nимеет длину " << l << " символов.\n";

    return 0;
}
```

2.
```
#include <iostream>
using namespace std;

int p(int base, int exp);

class pwr {
    int base;
    int exp;
public:
    pwr(int b, int e) { base = b; exp = e; }
    operator int() { return p(base, exp); }
};

// Возвращает основание base в степени exp
int p(int base, int exp)
{
    int temp;

    for(temp=1; exp; exp--) temp = temp * base;

    return temp;
}

int main()
{
    pwr o1(2, 3), o2(3, 3);
    int result;

    result = o1;
    cout << result << '\n';

    result = o2;
    cout << result << '\n';

    // объекты можно указывать непосредственно в инструкции cout,
    // как это сделано здесь
    cout << o1 + 100 << '\n';

    return 0;
}
```

13.3

1.
```
// Пример разделения ресурса с трассировкой работы
#include <iostream>
```

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

class output {
    static char outbuf[255]; // это разделяемый ресурс
    static int inuse; // если переменная inuse равна 0,
                      // буфер доступен; иначе он занят
    static int oindex; // индекс буфера
    char str[80];
    int i; // индекс следующего символа в str
    int who; // идентификатор объекта, должен быть положительным
public:
    output(int w, char *s) { strcpy(str, s); i = 0; who = w; }

/* Эта функция возвращает -1 при ожидании освобождения буфера, она
возвращает 0 при завершении вывода, и она возвращает who, если буфер
все еще используется
*/
    int putbuf()
    {
        if(!str[i]) { // вывод закончен
            inuse = 0; // освобождение буфера
            return 0; // сигнал завершения
        }
        if(!inuse) inuse = who; // захват буфера
        if(inuse != who) {
            cout << "Процесс" << who << "сейчас блокирован\n";
            return -1; // буфер использует кто-то еще
        }
        if(str[i]) { // символы все еще выводятся
            outbuf[oindex] = str[i];
            cout << "Процесс " << who << " выводит символы\n";
            i++; oindex++;
            outbuf[oindex] = '\0'; // последним всегда идет нуль
            return 1;
        }
        return 0;
    }
    void show() { cout << outbuf << '\n'; }
};

char output::outbuf[255]; // это разделяемый ресурс
int output::inuse = 0; // если переменная inuse равна 0,
                       // буфер доступен; иначе нет
int output::oindex = 0; // индекс буфера

int main()
{
  output o1(1, "Это проверка"), o2(2, " статических переменных");
```

```
    while(o1.putbuf() | o2.putbuf()); // вывод символов
    o1.show();
    return 0;
}
```

2.
```
#include <iostream>
#include <new>
using namespace std;

class test {
    static int count;
public:
    test() { count++; }
    ~test() { count--; }
    int getcount() { return count; }
};

int test::count = 0;

int main()
{
    test o1, o2, o3;

    cout << o1.getcount() << " объектов существует\n";

    test *p;

/* Отслеживание ошибок выделения памяти с помощью старого и нового
механизмов обработки ошибок
*/
    try {
        p = new test; // выделение памяти объекту
        if(!p) { // старый стиль
            cout << "Ошибка выделения памяти\n";
            return 1;
        }

    } catch(bad_alloc ba) { // новый стиль
            cout << "Ошибка выделения памяти\n";
            return 1;
      }

    cout << o1.getcount();
    cout << " объектов существует после выделения памяти\n";

    // удаление объекта
    delete p;

    cout << o1.getcount();
```

```
    cout << " объектов существует после удаления\n";
    return 0;
}
```

13.4

1. Для исправления программы просто сделайте переменную **current** модифицируемой, чтобы ее можно было изменить с помощью постоянной функции-члена **counting()** Решение упражнения представлено ниже:

```
// Теперь программа исправлена
#include <iostream>
using namespace std;

class CountDown {
 int incr;
 int target;
 mutable int current; // делаем переменную current модифицируемой
public:
 CountDown(int delay, int i = 1) {
     target = delay;
     incr = i;
     current = 0;
 }
    bool counting() const {
        current += incr;
        if(current >= target) {
            cout << "\a";
            return false;
        }
        cout << current << " ";
        return true;
    }
};

int main()
{
    CountDown ob(100, 2);
    while(ob.counting());
    return 0;
}
```

2. Нет, не может. Если бы у постоянной функции-члена была возможность вызвать не постоянную функцию-член, то не постоянную функцию-член можно было бы использовать для модификации вызывающего объекта.

13.5

1. Да.
2. Да, поскольку в C++ определено автоматическое преобразование из типа **int** в тип **double.**
3. Одна из проблем неявного преобразования конструкторов состоит в возможности просто забыть, что такое преобразование имело место. Например, неявное преобразование в инструкции присваивания очень напоминает перегруженный оператор присваивания, хотя их действия не обязательно аналогичны. Когда вы создаете классы, предназначенные для широкого использования, чтобы не вводить в заблуждение потенциальных пользователей ваших классов, было бы правильным отказаться от неявного преобразования конструкторов.

13.7

1.
```
/* В этой версии программы на экран выводится число символов,
записанных в буфер
*/
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    char buf[255];

    ostrstream ostr(buf, sizeof buf);

    ostr << "ввод/вывод через массивы работает с потоками\n";
    ostr << "точно так же, как обычный ввод/вывод\n " << 100;
    ostr << '  ' << 123.23 << '\n';

    // можно также использовать манипуляторы
    ostr << hex << 100 << '  ';
    // или флаги формата
    ostr.setf(ios::scientific);
    ostr << dec << 123.23;
    ostr << endl << ends; // гарантия того,
                          // что буфер завершится нулем

    // вывод на экран содержимого буфера
    cout << buf;

    cout << ostr.pcount();

    return 0;
}
```

2.
```
/* Массивы в качестве объектов ввода/вывода при копировании
содержимого одного массива в другой
*/
#include <iostream>
#include <strstream>
using namespace std;

char inbuf[] = "Это проверка ввода/вывода C++, основанного на
использовании символьных массивов";
char outbuf[255];

int main()
{
    istrstream istr(inbuf);
    ostrstream ostr(outbuf, sizeof outbuf);

    char ch;
    while(!istr.eof()) {
        istr.get(ch);
        if(!istr.eof()) ostr.put(ch);
    }
    ostr.put('\0'); // нулевой символ завершения

    cout << "Ввод: " << inbuf << '\n';
    cout << "Вывод: " << outbuf << '\n';

    return 0;
}
```

3.
```
// Преобразование строки в число с плавающей точкой
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
    float f;
    char s[ ] = "1234.564"; // вещественное в виде строки

    istrstream istr(s);

    // Простой способ преобразования строки
    // в ее внутреннее представление
    istr >> f;

    cout << "Преобразованная форма: " << f << '\n';

    return 0;
}
```

Проверка усвоения материала главы 13

1. В отличие от обычных переменных-членов, для которых характерно то, что в каждом объекте класса имеется их копия, у статической переменной-члена существует только одна копия, которую все объекты класса используют совместно.
2. Для использования массивов в качестве объектов ввода/вывода в программу необходимо включить заголовок **<strstream>**.
3. Нет.
4. `extern "C" int counter();`
5. Функция преобразования просто превращает объект в значение, совместимое с другим типом данных. Как правило, этим другим типом данных является встроенный тип данных C++.
6. Ключевое слово **explicit** применяется только к конструкторам. Оно предотвращает неявное преобразование конструкторов.
7. Постоянная функция-член не может модифицировать объект, который ее вызывает.
8. Пространство имен, объявляемое с помощью ключевого слова **namespace,** предназначено для локализации области видимости имен.
9. Ключевое слово **mutable** дает возможность постоянной функции — члену класса изменять данные, являющиеся членами этого класса.

Проверка усвоения материала в целом

1. Да. В ситуациях, в которых неявное преобразование выполняет то же действие, которое в отношении типа параметра конструктора было бы выполнено перегруженным оператором присваивания, перегружать оператор присваивания не нужно.
2. Да.
3. Новые библиотеки можно размещать в собственных пространствах имен, предотвращая тем самым конфликты имен с кодами других программ. Эта полезная возможность оказывается столь же полезной и в отношении старых кодов, которые предполагается использовать с новыми библиотеками.

ГЛАВА 14

Повторение пройденного

1. Пространства имен были добавлены в C++ для локализации имен идентификаторов с целью предотвращения конфликтов имен. Проблема конфлик-

тов имен серьезно заявила о себе в последнее время благодаря постоянному росту числа и объема библиотек классов сторонних фирм.

2. Чтобы функцию-член сделать постоянной, необходимо за списком параметров функции указать ключевое слово **const.** Например:

```
int f(int a) const;
```

3. Нет. Модификатор **mutable** (модифицируемый) позволяет постоянной функции-члену изменить переменную-член.

4.
```
class X {
    int a, b;
public:
    X(int i, int j) { a = i, b = j; }
    operator int() { return a+b; }
};
```

5. Да, это так.

6. Нет. Спецификатор **explicit** запрещает автоматическое преобразование типа **int** в тип **Demo.**

Упражнения

14.1

1. Контейнер — это объект, предназначенный для хранения других объектов. Алгоритм — это процедура, предназначенная для работы с содержимым контейнеров. Итератор по отношению к объектам библиотеки стандартных шаблонов действует аналогично указателю.

2. Предикаты бывают бинарными и унарными.

3. Существуют следующие пять типов итераторов: произвольного доступа, двунаправленный, однонаправленный, ввода и вывода.

14.3

2. Для любого объекта, хранящегося в векторе, должен быть определен конструктор по умолчанию.

3.
```
// Хранение в векторе объектов класса Coord
#include <iostream>
ttinclude<vector>
using namespace std;
```

```
class Coord {
public:
    int x, y;
    Coord() { x = y = 0; }
    Coord(int a, int b) { x = a; y = b; }
};

boll operator<(Coord a, Coord b)
{
    return (a.x + a.y) < (b.x + b.y);
}

boll operator==(Coord a, Coord b)
{
    return (a.x + a.y) == (b.x + b.y);
}

int main()
{
    vector<Coord> v;
    int i;

    for(i=0;i<10; i++)
        v.push_back(Coord(i,i));

    for(i=0; i<v.size(); i++)
        cout << v[i].x << "," << v[i].y << " ";

    cout << endl;

    for(i=0; i<v.size(); i++)
        v[i].x = v[i].x * 2;

    for(i=0; i<v.size(); i++)
        cout << v[i].x << "," << v[i].y << " ";

    return 0;
}
```

14.4

```
2. // Основные операции списка
#include <iostream>
#include <list>
using namespace std;

int main()
{
```

```
    list<char> lst; // создание пустого списка
    int i;

    for(i=0; i<10; i++) lst.push_back('A' + i);

    cout << "Размер = " << lst.size() << endl;

    list<char>::iterator p;

    cout << "Содержимое: ";
    for(i=0; i<lst.size(); i++) {
      p = lst.begin();
      cout << *p;
      lst.pop_front();
      lst.push_back(*p); // размещение элемента в конце списка
    }

    cout << endl;

    if(!lst.empty())
      cout << "Список пустым не является\n";

    return 0;
}
```

После выполнения программы на экране появится следующее:

```
Размер = 10
Содержимое: ABCDEFGHIJ
Список пустым не является
```

В данной программе элементы поочередно извлекаются из начала списка и размещаются в его конце. Таким образом список никогда не остается пустым. В цикле по выводу на экран содержимого списка с помощью функции size() организуется непрерывный контроль его длины.

3.

```
// Слияние двух списков проектов
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class Project {
public:
    char name[40];
    int days_to_completion;
    Project() {
      strcpy(name, " ");
      days_to_completion = 0;
    }
```

```
    Project(char *n, int d) {
        strcpy(name, n);
        days_to_completion = d;
    }

    void add_days(int i) {
        days_to_completion += i;
    }

    void sub_days(int i) {
        days_to_completion -= i;
    }

    bool completed() { return !days_to_completion; }

    void report() {
        cout << name << ": ";
        cout << days_to_completion;
        cout << " дней до завершения\n";
    }
};

bool operator<(const Project &a, const Project &b)
{
    return a.days_to_completion < b.days_to_completion;
}

bool operator>(const Project Sa, const Project &b)
{
    return a.days_to_completion > b.days_to_completion;
}

bool operator==(const Project &a, const Project &b)
{
    return a.days_to_completion == b.days_to_completion;
}

bool operator!=(const Project &a, const Project &b)
{
    return a.days_to_completion != b.days_to_completion;
}

int main()
{
    list<Project>proj;
    list<Project> proj2;

    proj.push_back(Project("Разработка компилятора", 35));
    proj.push_back(Project("Разработка электронной таблицы", 190));
    proj.push_back(Project("Разработка STL", 1000));
```

```
    proj2.push_back(Project("Разработка базы данных", 780));
    proj2.push_back(Project("Разработка стандартных писем", 50));
    proj2.push_back(Project("Разработка объектов COM", 300));

    proj.sort();
    proj2.sort();

    proj.merge(proj2); // слияние списков

    list<Project>::iterator p = proj.begin();

    // вывод проектов на экран
    while(p != proj.end()) {
        p->report();
        p++;
    }

    return 0;
}
```

14.5

2.
```
// Ассоциативный список абонентов и телефонных номеров
#include <iostream>
tinclude <map>
tinclude <cstring>
using namespace std;

class name {
    char str[20];
public:
    name() { strcpy(str, ""); }
    name(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

// Для объектов типа name следует определить оператор < (меньше)
bool operator<(name a, name b)
{
    return strcmp(a.get(), b.get()) < 0;
}

class phonenum {
    char str[20];
public:
    phonenum() { strcmp(str, ""); }
    phonenum(char *s) { strcpy(str, s); }
    char *get() { return str; }
};
```

```
int main()
{
    map<name, phonenum> m;

    // Размещение в ассоциативном списке имен абонентов
    // и их телефонных номеров
    m.insert(pair<name, phonenum>
        (name("Василий"), phonenum("541-85-51")));
    m.insert(pair<name, phonenum>
        (name("Иосиф"), phonenum("550-09-96")));
    m.insert(pair<name, phonenum>
        (name("Михаил"), phonenum("8-3712-41-16-36")));
    m.insert(pair<name, phonenum>
        (name("Никодим"), phonenum("8-095-967-85-85")));

    // Поиск телефонного номера по заданному имени абонента
    char str[80];
    cout << "Введите имя: ";
    cin >> str;

    map<name, phonenum>::iterator p;

    p = m.find(name(str));
    if(p != m.end())
        cout << "Телефонный номер: " << p->second.get();
    else
        cout << "Такого имени в ассоциативном списке нет\n";

    return 0;
}
```

3. Да.

14.6

1.
```
// Сортировка вектора с помощью алгоритма sort
#include <iostream>
#include <vector>
#include <cstdlib>
ttinclude<algoritm>
using namespace std;

int main()
{
    vector<char> v;
    int i;

    // Создание вектора из случайных символов
```

```
    for(i=0; i<10; i++)
        v.push_back( 'A' + (rand()%26) );

    cout << "Исходное содержимое: " ;
    for(i=0; i<v.size(); i++)
        cout << v[i] << " ";

    cout << endl << endl;

    // Сортировка вектора
    sort(v.begin(), v.end());

    cout << "Отсортированное содержимое: " ;
    for(i=0; i<v.size(); i++)
        cout << v[i] << " ";

    return 0;
}
```

2.
```
// Выполнение слияния двух списков с помощью алгоритма merge
#include <iostream>
#include <list>
#include <algoritm>
using namespace std;

int main()
{
    list<char> lst1, lst2, lst3(20);
    int i;

    for(i=0; i<10; i+=2) lst1.push_back('A' +. i);
    for(i=0; i<11; i+=2) lst2.push_back( 'A' + i);

    cout << "Содержимое списка lst1: ";
    list<char>::iterator p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;

    cout << "Содержимое списка lst2: ";
    p = lst2.begin();
    while(p != lst2.end()) {
        cout << *p;
        p++;
    }
    cout << endl << endl;
```

```
    // Теперь выполняем слияние двух списков
    merge(lst1.begin(), lst1.end(),
          lst2.begin(),lst2.end(),
          lst3.begin());

    cout << "Содержимое списка после слияния: ";
    p = lst3.begin();
    while(p != lst3.end()) {
        cout << *p;
        p++;
    }

    return 0;
}
```

14.7

1.
```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> str;

    str.push_back(string("один"));
    str.push_back(string("два"));
    str.push_back(string("три"));
    str.push_back(string("четыре"));
    str.push_back(string("пять"));
    str.push_back(string("шесть"));
    str.push_back(string("семь"));
    str.push_back(string("восемь"));
    str.push_back(string("девять"));
    str.push_back(string("десять"));

    str.sort(); // сортировка списка

    list<string>::iterator p = str.begin();
    while(p != str.end()) {
        cout << *p << " ";
        p++;
    }

    return 0;
}
```

2.
```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
    string str;

    cout << "Введите строку: ";
    cin >> str;

    int i = count(str.begin(), str.end(), 'e');

    cout << i << " символов являются символом e\n";

    return 0;
}
```

3.
```
#include <iostream>
ttinclude <string>
ttinclude <algorithm>
#include <cctype>
using namespace std;

int main()
{
    string str;

    cout << "Введите строку: ";
    cin >> str;

    int i = count_if(str.begin(), str.end(), islower);

    cout << i << " символов введено в нижнем регистре\n";

    return 0;
}
```

4. Класс **string** — это конкретизация класса-шаблона **basic_string**.

Проверка усвоения материала главы 14

1. Библиотека стандартных шаблонов предоставляет многократно опробованные, отлаженные версии многих традиционных структур данных и алгоритмов. Поскольку контейнеры, входящие в библиотеку стандартных шаблонов, являются классами-шаблонами, их можно использовать для хранения данных любых типов.

2. Контейнер — это объект, предназначенный для хранения других объектов. Алгоритм — это процедура, предназначенная для работы с содержимым контейнеров. Итератор по отношению к объектам библиотеки стандартных шаблонов действует аналогично указателю.

3.
```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main()
{
    vector<int> v(10);
    list<int> lst;
    int i;

    for(i=0; i<10; i++) v[i] = i;

    for(i=0; i<10; i++)
        if(!(v[i]%2)) lst.push_back(v[i]);

    list<int>::iterator p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }

    return 0;
}
```

4. Строковый тип данных дает возможность обрабатывать строки с помощью обычных операторов. С другой стороны, работа с данными строкового типа не настолько эффективна, как работа с оканчивающимся нулем символьными массивами.

5. Предикат — это функция, возвращаемым значением которой является истина либо ложь.

Приложение С

Список английских технических терминов

Настоящее приложение содержит список использованных в книге английских технических терминов. В табл. С.1 термины расположены в соответствии с порядком английского алфавита, а в табл. С.2 — русского. В первом столбце указан сам термин, во втором — его аббревиатура (если таковая имеется) или его наиболее общеупотребительное название, и в третьем — русский перевод термина.

Таблица С. 1. *Английские технические термины (A-Z)*

Название	Аббревиатура	Перевод
Abstract Class		Абстрактный класс
Access Specifier		Спецификатор доступа
Algorithm		Алгоритм
Allocator		Распределитель памяти
Ambiguity		Неоднозначность
American National Standards Institute	ANSI	Американский национальный институт стандартов
Anonymous Union		Анонимное объединение
Array-Based I/O		Ввод/вывод с использованием массивов
Associative Container		Ассоциативный контейнер
Automatic Bounds Checking		Автоматический контроль границ
Base Class		Базовый класс
Bidirectional Iterator	Bilter	Двунаправленный итератор
Bitmask		Битовая маска
Bounded Array		Защищенный массив
C string		Строка в стиле С
C++ Standard Library		Стандартная библиотека C++

Таблица С. 1 (продолжение)

Название	Аббревиатура	Перевод
Casting Operator		Оператор приведения типов
Class		Класс
Comparison Function		Функция сравнения
Const Class Member		Постоянный член класса
Constructor		Конструктор
Container		Контейнер
Container Class		Класс-контейнер
Conversion Function		Функция преобразования
Copy Constructor		Конструктор копий
Default Argument		Аргумент по умолчанию
Derived Class		Производный класс
Destructor		Деструктор
Early Binding		Раннее связывание
Encapsulation		Инкапсуляция
Enumeration		Перечисление, перечислимый тип данных
Exception		Исключительная ситуация
Exception Catching		Перехват исключительной ситуации
Exception Handling		Обработка исключительной ситуации
Exception Throwing		Возбуждение исключительной ситуации
Extraction Operator		Оператор извлечения
Extractor		Функция извлечения
Format Flag		Флаг формата
Forward Declaration		Предварительное объявление
Forward Iterator	ForIter	Однонаправленный итератор
Forward Reference		Ссылка вперед
Friend Function		Дружественная функция
Function Object		Объект-функция
Function Overloading		Перегрузка функций
Generated Function		Порожденная функция

Таблица С. 1 (продолжение)

Название	Аббревиатура	Перевод
Generic Class		Родовой класс
Generic Function		Родовая функция
Get Pointer		Указатель считывания
Header		Заголовок
Header File		Заголовочный файл
Heap		Пирамида
Hierarchical Classification		Иерархия классов
I/O Manipulator		Манипулятор ввода/вывода
Independent Reference		Независимая ссылка
Indirect Base Class		Косвенный базовый класс
Inheritance		Наследование
In-Line Function		Встраиваемая функция
Input Iterator	InIter	Итератор ввода
Input/Output	I/O	Ввод/вывод
Inserter		Функция вставки
Inserter Function		Функция вставки
Inserter Operator		Оператор вставки
Insertion		Вставка
Insertion Operation		Операция вставки
Instantiating		Создание экземпляра функции
International Standards Organization	ISO	Международная организация по стандартам
Iterator		Итератор
Join		Соединение
Key		Ключ
Large Character Set		Расширенный набор символов
Late Binding		Позднее связывание
Linkage Specifier		Спецификатор сборки
List		Список
Mangling		Искажение
Manipulator		Манипулятор
Map		Ассоциативный список
Member		Член

Таблица С. 1 (продолжение)

Название	Аббревиатура	Перевод
Member Function •		Функция-член
Merge		Слияние
Microsoft Foundation Classes	MFC	Библиотека классов MFC
Multiple Inheritance		Множественное наследование
Mutable Class Member		Модифицируемый член класса
Name Collision		Конфликт имен
Name Mangling		Искажение имен
Namespace		Пространство имен
Object		Объект
Object Factory		Фабрика объектов
Object Oriented Programming	OOP	Объектно-ориентированное программирование
Operation		Операция
Operator		Оператор
Operator Function •		Оператор-функция
Operator Overloading		Перегрузка операторов
Output Iterator	OutIter	Итератор вывода
Overloading		Перегрузка
Overriding		Подмена
Pass by Reference		Передача по ссылке
Pass by Value		Передача по значению
Permutation		Перестановка
Pointer		Указатель
Polymorphic Class		Полиморфный класс
Polymorphism		Полиморфизм
Predicate		Предикат
Private		Закрытый
Protected		Защищенный
Public		Открытый
Pure Virtual Function		Чистая виртуальная функция
Put Pointer		Указатель записи
Queue		Очередь

Таблица С. 1 (продолжение)

Название	Аббревиатура	Перевод
Random Access		Произвольный доступ
Random Access Iterator	RandIter	Итератор произвольного доступа
Redefinition		Переопределение
Reference		Ссылка
Reverse Iterator		Обратный итератор
Run-Time Error		Динамическая ошибка
Run-Time Polymorphism		Динамический полиморфизм
Run-Time Type Identification	RTTI	Динамическая идентификация типа
Safe Array		Безопасный массив
Scope		Область видимости
Scope Resolution Operator		Оператор расширения области видимости
Sequence		Последовательность
Stack		Стек
Standard Template Library	STL	Библиотека стандартных шаблонов
Statement		Инструкция
Static Class Member		Статический член класса
Stream		Поток ввода/вывода
String Class		Строковый класс
String Data Type		Строковый тип данных
Structure		Структура
Structured Programming Language		Язык структурного программирования
Template		Шаблон
Template Class		Класс-шаблон
Template Function		Функция-шаблон
Type Promotion		Приведение типа
Union		Объединение
Unnamed Namespace		Безымянное пространство имен
Vector		Вектор
Virtual Base Class		Виртуальный базовый класс
Virtual Function		Виртуальная функция

Таблица C.2. *Английские технические термины (А-Я)*

Название	Аббревиатура	Перевод
Abstract Class		Абстрактный класс
Automatic Bounds Checking		Автоматический контроль границ
Algorithm		Алгоритм
American National Standards Institute	ANSI	Американский национальный институт стандартов
Anonymous Union		Анонимное объединение
Default Argument		Аргумент по умолчанию
Associative Container		Ассоциативный контейнер
Map		Ассоциативный список
Base Class		Базовый класс
Safe Array		Безопасный массив
Unnamed Namespace		Безымянное пространство имен
Microsoft Foundation Classes	MFC	Библиотека классов MFC
Standard Template Library	STL	Библиотека стандартных шаблонов
Bitmask		Битовая маска
Input/Output	I/O	Ввод/вывод
Array-Based I/O		Ввод/вывод с использованием массивов
Vector		Вектор
Virtual Function		Виртуальная функция
Virtual Base Class		Виртуальный базовый класс
Exception Throwing		Возбуждение исключительной ситуации
Insertion		Вставка
In-Line Function		Встраиваемая функция
Bidirectional Iterator	Bilter	Двунаправленный итератор
Destructor		Деструктор
Run-Time Type Identification	RTTI	Динамическая идентификация типа
Run-Time Error		Динамическая ошибка
Run-Time Polymorphism		Динамический полиморфизм
Friend Function		Дружественная функция
Header		Заголовок
Header File		Заголовочный файл

***Таблица** С.2 (продолжение)*

Название	Аббревиатура	Перевод
Private		Закрытый
Protected		Защищенный
Bounded Array		Защищенный массив
Hierarchical Classification		Иерархия классов
Encapsulation		Инкапсуляция
Statement		Инструкция
Mangling		Искажение
Name Mangling		Искажение имен
Exception		Исключительная ситуация
Iterator		Итератор
Input Iterator	InIter	Итератор ввода
Output Iterator	OutIter	Итератор вывода
Random Access Iterator	RandIter	Итератор произвольного доступа
Class		Класс
Container Class		Класс-контейнер
Template Class		Класс-шаблон
Key		Ключ
Constructor		Конструктор
Copy Constructor		Конструктор копий
Container		Контейнер
Name Collision		Конфликт имен
Indirect Base Class		Косвенный базовый класс
Manipulator		Манипулятор
I/O Manipulator		Манипулятор ввода/вывода
International Standards Organization	ISO	Международная организация по стандартам
Multiple Inheritance		Множественное наследование
Mutable Class Member		Модифицируемый член класса
Inheritance		Наследование
Independent Reference		Независимая ссылка
Ambiguity		Неоднозначность
Scope		Область видимости

Таблица С.2 (продолжение)

Название	Аббревиатура	Перевод
Exception Handling		Обработка исключительной ситуации
Reverse Iterator		Обратный итератор
Union		Объединение
Object		Объект
Object Oriented Programming	OOP	Объектно-ориентированное программирование
Function Object		Объект-функция
Forward Iterator	ForIter	Однонаправленный итератор
Operator		Оператор
Inserter Operator		Оператор вставки
Extraction Operator		Оператор извлечения
Casting Operator		Оператор приведения типов
Scope Resolution Operator		Оператор расширения области видимости
Operator Function		Оператор-функция
Operation		Операция
Insertion Operation		Операция вставки
Public		Открытый
Queue		Очередь
Overloading		Перегрузка
Operator Overloading		Перегрузка операторов
Function Overloading		Перегрузка функций
Pass by Value		Передача по значению
Pass by Reference		Передача по ссылке
Redefinition		Переопределение
Permutation		Перестановка
Exception Catching		Перехват исключительной ситуации
Enumeration		Перечисление, перечислимый тип данных
Heap		Пирамида
Overriding		Подмена
Late Binding		Позднее связывание

Таблица С.2 (продолжение)

Название	Аббревиатура	Перевод
Polymorphism		Полиморфизм
Polymorphic Class		Полиморфный класс
Generated Function		Порожденная функция
Sequence		Последовательность
Const Class Member		Постоянный член класса
Stream		Поток ввода/вывода
Forward Declaration		Предварительное объявление
Predicate		Предикат
Type Promotion		Приведение типа
Derived Class		Производный класс
Random Access		Произвольный доступ
Namespace		Пространство имен
Early Binding		Раннее связывание
Allocator		Распределитель памяти
Large Character Set		Расширенный набор символов
Generic Function		Родовая функция
Generic Class		Родовой класс
Merge		Слияние
Join		Соединение
Instantiating		Создание экземпляра функции
Access Specifier		Спецификатор доступа
Linkage Specifier		Спецификатор сборки
List		Список
Reference		Ссылка
Forward Reference		Ссылка вперед
C++ Standard Library		Стандартная библиотека C++
Static Class Member		Статический член класса
Stack		Стек
C string		Строка в стиле С
String Class		Строковый класс
String Data Type		Строковый тип данных
Structure		Структура

Таблица С.2 (продолжение)

Название	Аббревиатура	Перевод
Pointer		Указатель
Put Pointer		Указатель записи
Get Pointer		Указатель считывания .
Object Factory		Фабрика объектов
Format Flag		Флаг формата
Inserter		Функция вставки
Inserter Function		Функция вставки
Extractor		Функция извлечения
Conversion Function		Функция преобразования
Comparison Function		Функция сравнения
Member Function		Функция-член
Template Function		Функция-шаблон
Pure Virtual Function		Чистая виртуальная функция
Member		Член
Template		Шаблон
Structured Programming Language		Язык структурного программирования

Приложение D

Описание дискеты

На дискете находятся листинги программ, которые приведены в книге в разделах "Примеры" (каталог Example) и "Упражнения" (каталог Exercise). Листинги из главы с номером N находятся в соответствующем подкаталоге ChapN. Листинги программ — ответы на упражнения, расположенные в начале каждой главы (раздел "Повторение пройденного"), находятся в подкаталоге Review. Листинги профамм — ответы на упражнения, расположенные в конце каждой главы (раздел "Проверка усвоения материала главы"), находятся в подкаталоге Mastery. Листинги профамм — ответы на упражнения, расположенные также в конце каждой главы (раздел "Проверка усвоения материала в целом"), находятся в подкаталоге Cumulative. Тексты профамм содержат символы русского языка и предназначены для компиляции и запуска в MS-DOS (каталог DOS). Если вы работаете в MS Windows, то необходимо компилировать эти программы как консольные приложения (каталог Windows). Для компиляции можно использовать любой современный компилятор C++.

Не огорчайтесь, если записанные на дискете профаммы будут работать не так, как вы ожидаете, или вообще не будут работать. Поиск и исправление синтаксических ошибок — это тоже прекрасный способ изучения языка профаммирования. Желаем удачи!

Предметный указатель

M

H

O

П

Р

С

Т

У

Ф

Ч

Ш

Я

Содержание

Кирьянов Д. Самоучитель Mathcad 11	560 с.
Кирьянов Д. Самоучитель Mathcad 2001	544 с.
Кирьянов Д., Кирьянова Е. Самоучитель Adobe After Effects 6.0	368 с.
Кирьянов Д., Кирьянова Е. Самоучитель Adobe Premiere 6.0	480 с.
Кирьянов Д., Кирьянова Е. Самоучитель Adobe Premiere Pro	448 с.
Кирьянов Д., Кирьянова Е. Самоучитель Adobe Premiere 6.5	480 с.
Клюквин А. Краткий самоучитель работы на ПК	432 с.
Комолова Н. Компьютерная верстка и дизайн	512 с.
Коркин И. Самоучитель Microsoft Internet Explorer 6.0	288 с.
Костромин В. Самоучитель Linux для пользователя	672 с.
Костромин В. Приложение к книге Костромина В. Самоучитель Linux для пользователя 4 CD-ROM " Дистрибутив Red Hat L	
Котеров Д. Самоучитель PHP 4	576 с.
Кузнецов И. Самоучитель видео на ПК (+CD-ROM)	416 с.
Кузнецов М., Симдянов И. PHP 5	560 с.
Кузютина А., Шапошников И. Самоучитель Adobe GoLive 6	352 с.
Культин Н. Delphi 6. Программирование на Object Pascal	528 с.
Культин Н. Основы программирования в Delphi 7 (+дискета)	608 с.
Культин Н. Программирование в TurboPascal 7 и Delphi, 2-е изд. (+дискета)	416 с.
Культин Н. C++ Builder (+прил. на CD-ROM)	320 с.
Леоненков А. Самоучитель UML	304 с.
Леоненков А. Самоучитель UML 2-изд.	432 с.
Матросов А., Чаунин М. Самоучитель Perl	432 с.
Медведев Е., Трусова В. Музыкальная азбука на PC (+дискета)	496 с.
Медников В. Основы комьютерной музыки	336 с.
Мур М. и др. Телекоммуникации. Руководство для начинающих	624 с.
Надеждин Н. Цифровая фотография. Практическое руководство	368 с.
Немнюгин С. Современный Фортран	496 с.
Омельченко Л. Самоучитель Visual Foxpro	688 с.
Омельченко Л., Федоров А. Самоучитель Microsoft Windows XP	560 с.
Омельченко Л., Федоров А. Самоучитель Windows 2000 Professional	528 с.
Омельченко Л. Самоучитель Visual FoxPro 7.0	678 с.
Омельченко Л. Самоучитель Visual FoxPro 8	688 с.
Пекарев Л. Самоучитель 3ds max 5	336 с.
Полещук Н. Самоучитель AutoCAD 2002	608 с.
Полещук Н., Савельева В. Самоучитель AutoCAD 2004	640 с.
Поляк-Брагинский А. Сеть своими руками	320 с.
Поляк-Брагинский А. Сеть своими руками, 2-е изд.	432 с.
Понамарев В. Самоучитель Delphi 7 Studio	512 с.

Понамарев В. Самоучитель JBuilder 6/7	304 с.
Понамарев В. Самоучитель KYLIX	416 с.
Правин О. Правильный самоучитель работы на компьютере 2-е изд.	496 с.
Секунов Н. Самоучитель C#	576 с.
Секунов Н. Самоучитель Visual C++ .NET (+дискета)	738 с.
Секунов Н. Самоучитель Visual C++ 6 (+дискета)	960 с.
Сироткин С., Чалышев И., Воробьев С. Самоучитель WML и WMLScript	240 с.
Соломенчук В. Аппаратные средства персональных компьютеров	512 с.
Тайц А. М., Тайц А. А. Самоучитель Adobe Photoshop 7 (+дискета)	688 с.
Тайц А. М., Тайц А. А. Самоучитель CorelDRAW 11	704 с.
Тихомиров Ю. Самоучитель MFC (+дискета)	640 с.
Токарев С. Самоучитель Macromedia Dreamweaver MX	544 с.
Токарев С. Самоучитель Macromedia Fireworks	448 с.
Трасковский А. Устройство, модернизация, ремонт IBM PC	608 с.
Трасковский А. Сбои и неполадки домашнего ПК	384 с.
Трусова В., Медведев *Е.* Музыкальная азбука на PC (+дискета)	496 с.
Федорова А. Самоучитель Adobe PageMaker 7	736 с.
Хабибуллин И. Самоучитель Java	464 с.
Хабибуллин И. Самоучитель XML	336 с.
Хомоненко А. Самоучитель Microsoft Word 2000	688 с.
Хомоненко А. Самоучитель Microsoft Word 2002	624 с.
Хомоненко А., Хомоненко Н. Самоучитель Microsoft Word 2003	672 с.
Хомоненко А., Гофман В. Самоучитель Delphi	576 с.
Шапошников И. Самоучитель HTML 4	288 с.
Шапошников И. Интернет. Быстрый старт	272 с.
Шапошников И. Самоучитель ASP.NET	368 с.
Шилдт Г. Самоучитель C++, 3-е изд. (+дискета)	688 с.

ИЗУЧАЕМ ВМЕСТЕ С BHV

Г. Шилдт

САМОУЧИТЕЛЬ

C++

Книга построена на коротких, тщательно продуманных уроках. Автор начинает каждый урок с описания определенного программного принципа, далее переходит к примеру, иллюстрирующему этот принцип в действии, и заканчивает урок упражнениями для закрепления изученного материала. Уроки даются строго последовательно. Каждый новый урок строится на основе предыдущего и готовит Вас к следующему. Поскольку каждая глава начинается и заканчивается упражнениями на проверку пройденного материала, Вы всегда будете знать, следует ли вернуться назад или можно двигаться дальше.

Предлагаемые Вам уроки программирования на C++ охватывают следующие темы:

- Принципы объектно-ориентированного программирования
- Классы и объекты
- Конструкторы и деструкторы
- Перегрузка функций и операторов
- Пространства имен
- Виртуальные функции
- Система ввода-вывода C++
- Шаблоны и контейнеры
- Обработка исключительных ситуаций
- Динамическая идентификация типа
- Библиотека стандартных шаблонов

ISBN-5-7791-0086-1

bhv